Filesystem - Inode

Youjip Won

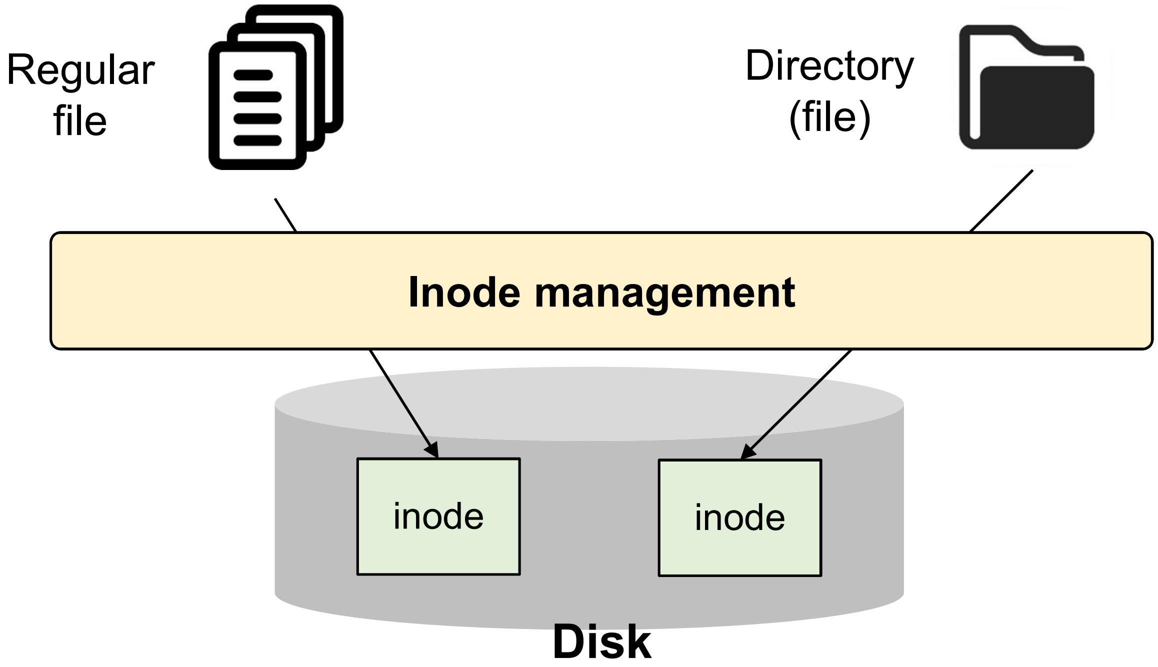
KAIST EE

Contents

- Inode structure: On-disk and in-memory inode.
- Code:
  - `iget()`,`iput()`, `ilock()`, and `iunlock()`
  - `ialloc()`, `iupdate()`, and `itrunc()`
- Reading or writing the data through inode.
- Code:
  - `readi()` and `writei()`
  - `filewrite()`

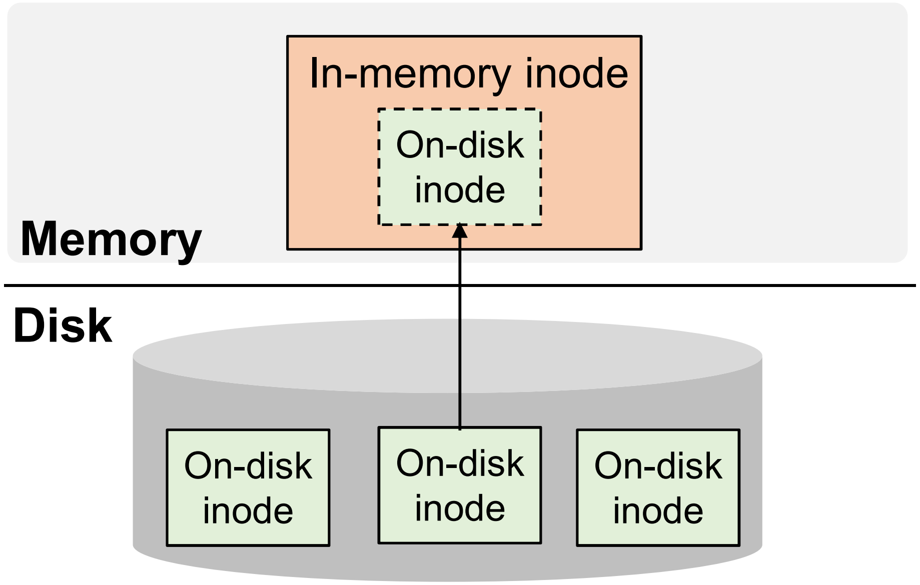
Inode

- 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
  - creation time, modification time, access authority
  - locations of file blocks



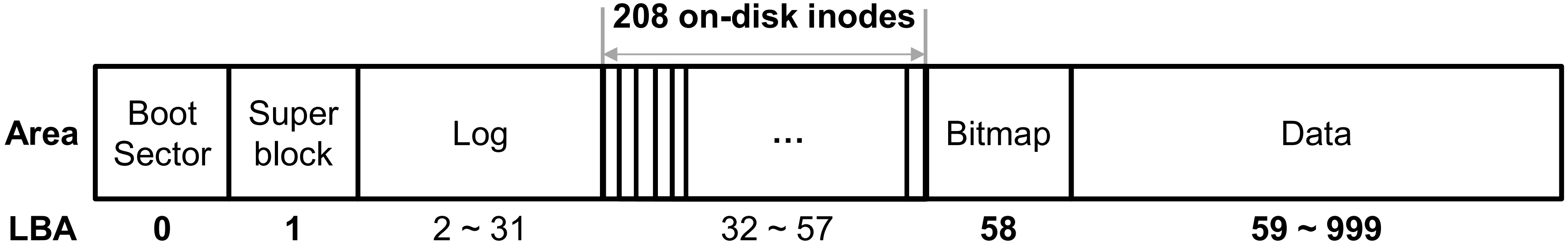
On-disk inode and in-memory inode

- There are on-disk inode and in-memory inode.
    - On-disk inode: inode structure on the disk
    - In-memory inode: inode structure in the memory.
- In-memory inode contains a copy of the on-disk inode and information needed within the kernel.



On-disk inode

- All of the on-disk inodes are stored into the inode blocks of disk.
- Every inode is the same size, 64 Byte.
  - 8 inodes on a single block.
  - 26 inode blocks.
  - There are 208 on-disk inode slots in the disk.
- The content of inode blocks is the array of on-disk inodes.

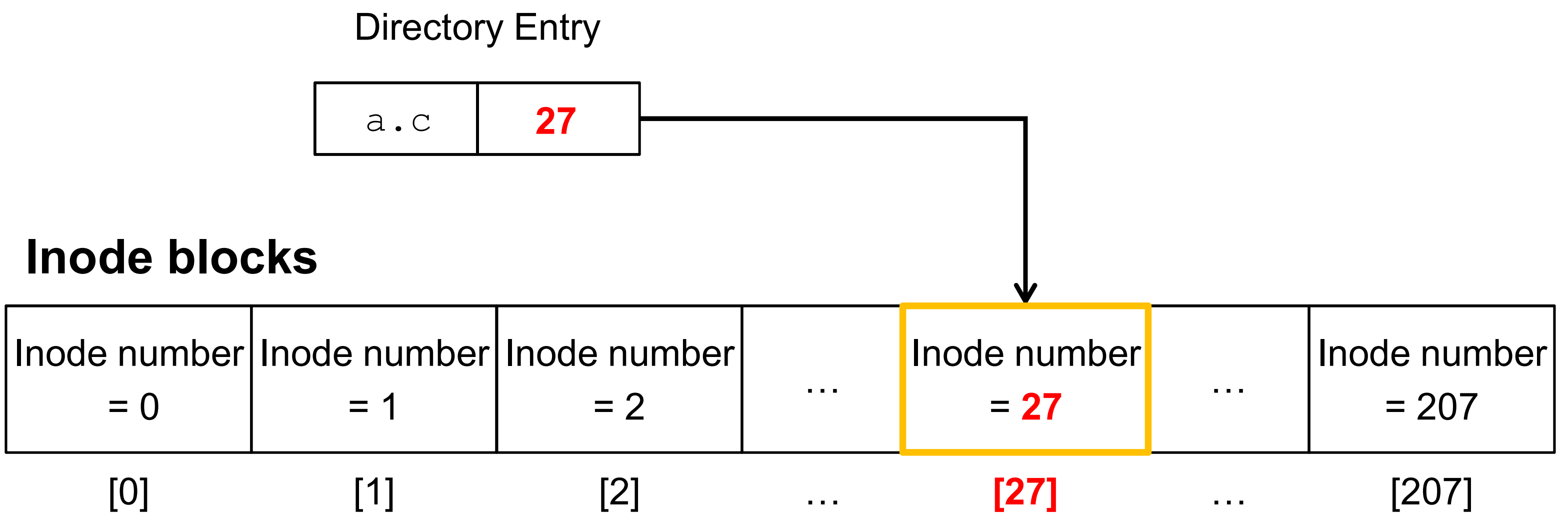


208 on-disk inodes

Area	Boot Sector	Super block	Log	…	Bitmap	Data
LBA	**0**	**1**	2 ~ 31	32 ~ 57	**58**	**59 ~ 999**

Inode number

- The index of on-disk inode is called inode number.
- Inode number is how inodes are identified in the kernel.
- The directory entry stores the inode number.
  - This number represents the location of an on-disk inode.



Directory Entry

a.c	27

Inode blocks

Inode number = 0	Inode number = 1	Inode number = 2	…	Inode number = 27	…	Inode number = 207
[0]	[1]	[2]	…	[27]	…	[207]

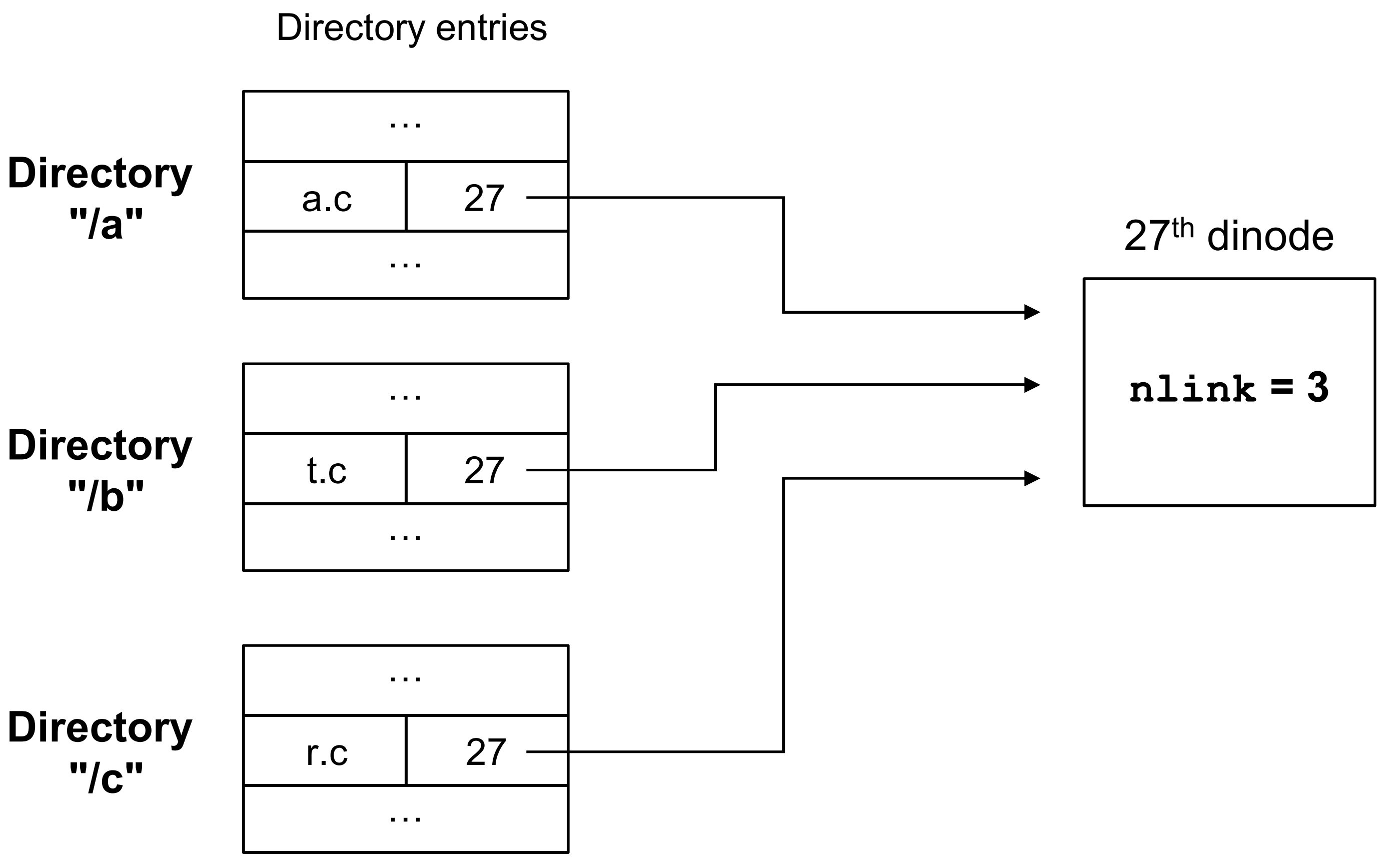
struct dinode

- type: T_FILE (regular file), T_DIR (directory), or T_DEV (device file)
- major and minor (T_DEV only)
  - Driver type and device ID for a driver type.
- nlink: the number of directory entries referring to this inode
- size: file size (byte)
- addrs: file block addresses

```
 struct dinode {
  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

nlink

- The number of directory entries that refer to this inode



Directory entries

Directory "/a"

…	
a.c	27
…	

Directory "/b"

…	
t.c	27
…	

Directory "/c"

…	
r.c	27
…	

27th dinode

nlink = 3

In-memory inode

- Inode structure cached in memory from the disk.
- It contains a copy of the on-disk inode and the information needed within the kernel.
  - Reference count, lock, and so on...

```
struct inode {
  uint dev;
  uint inum;
  int ref;
  struct sleeplock lock;
  int valid;

  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

The content of the on-disk inode.

struct inode

- `dev`: device number
- `inum`: inode number
- `ref`: the number of the processes that currently open the file
- `lock`: sleep lock for the exclusive access of `valid` and the copy of on-disk inode.
- `valid`: indicator that represents whether the copy of on-disk inode is valid.
  - If the value is 1, the content of on-disk inode is valid.

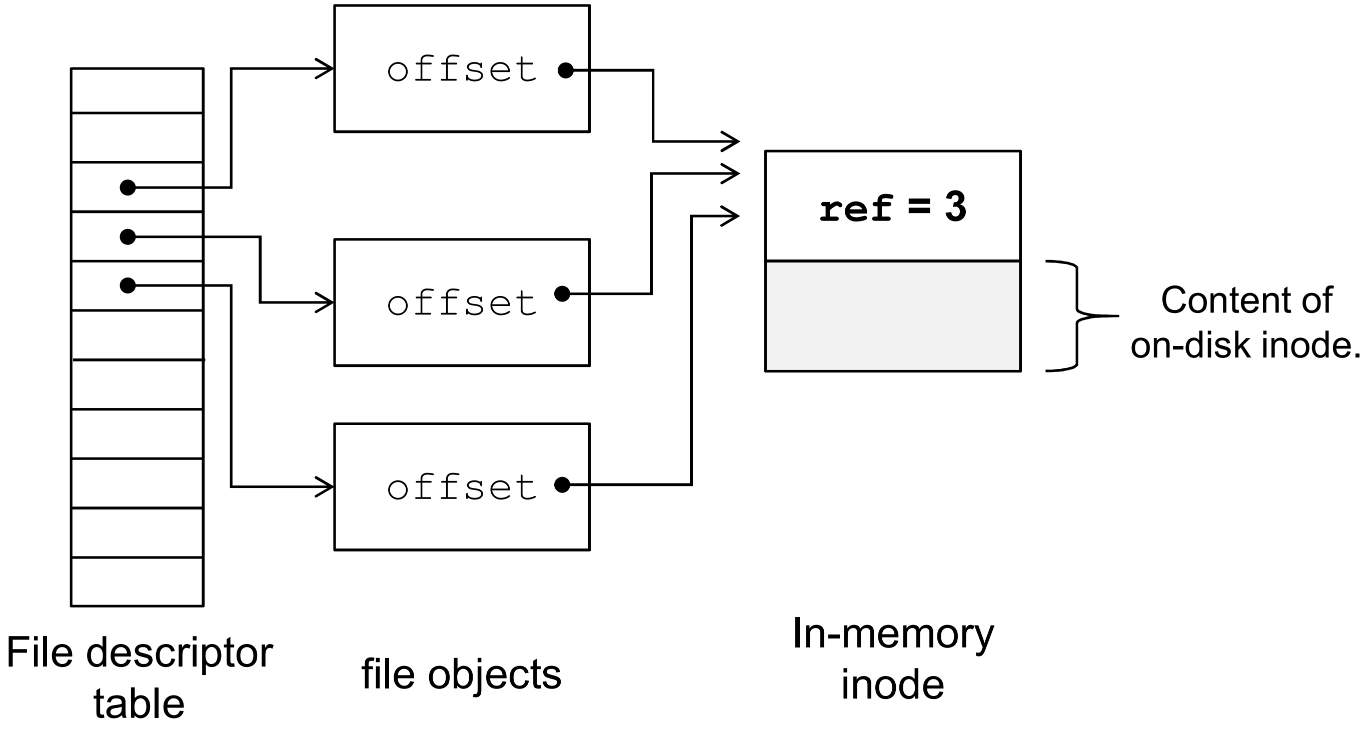
```
struct inode {
  uint dev;
  uint inum;
  int ref;
  struct sleeplock lock;
  int valid;

  // copy of disk inode
                    …
};
```

ref

- The number of C-pointers that refer to this inode.
- There is only one copy of a single on-disk inode in the memory.

ex)

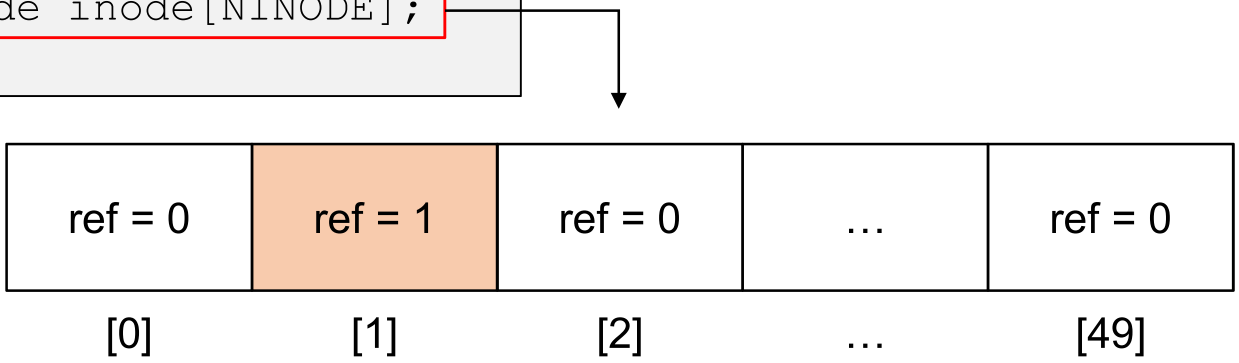
```
fd1 = open(a.c);
fd2 = open(a.c);
fd3 = open(a.c);
```



Inode cache: `struct icache`

- xv6 maintains an array of in-memory inodes, which is called inode cache.
- Inode cache contains the `NINODE` (= 50) entries.
- It is protected by spin lock from the race conditions.
- `ref` attribute in inode structure represents the cache entry is free or not.
  - If the value of `ref` is larger than 0, cache entry is not free.
  - If the value of `ref` is 0, cache entry is free.

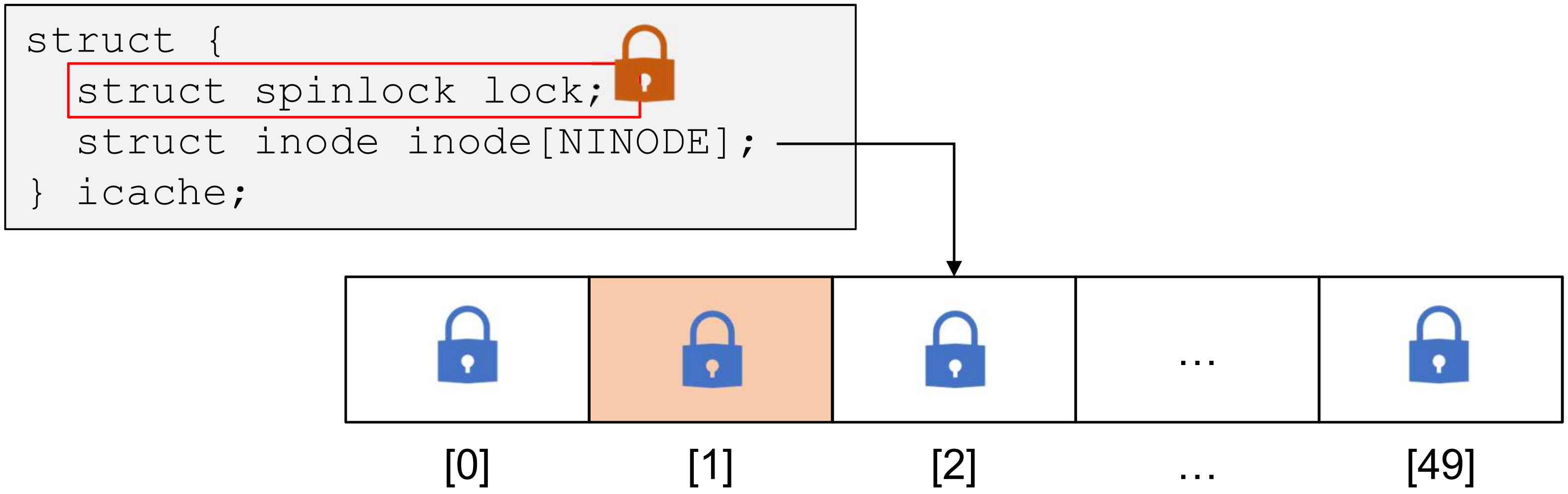
```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;
```



ref = 0	ref = 1	ref = 0	…	ref = 0
[0]	[1]	[2]	…	[49]

Inode cache lock vs. per-inode locks

- To prevent race condition for inode cache and its entries, xv6 uses two types of lock.
  - Inode cache lock (spin lock)
    - It protects the variable for managing the inode cache.
    - e.g) `dev`, `inum`, and `ref` for all the in-memory inodes.
  - Per-inode locks (sleep lock)
    - It controls the concurrent accesses to inode and serializes them.
    - Each lock protects the file data, `valid`, and the content of on-disk inode of the corresponding inode.



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

`iinit(int dev)`: Initializing the inode cache.

- It is called right before executing the very first user process.
- It initializes the two types of lock; inode cache lock and per-inode locks.

```
void forkret(void){
                    …
  if (first) {
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
  }
                    …
}
```

```
void
iinit(int dev)
{
  int i = 0;

  initlock(&icache.lock, "icache");
  for(i = 0; i < NINODE; i++) {
    initsleeplock(&icache.inode[i].lock, "inode");
  }
                                …
}
```

Functions for in-memory inode

- `iget(uint dev, uint inum)` and `iput(struct inode *ip)`
  - Reserve or release the in-memory inode in the inode cache.
- `ilock(struct inode *ip)` and `iunlock(struct inode *ip)`
  - Acquire or release the per-inode lock for a given inode.
  - `ilock()` function loads the on-disk inode if it is invalid.

struct inode *iget(uint dev, uint inum)

- Return the pointer of in-memory inode for the given `dev` and inode number (`inum`).
- If target inode is already in cache,
  - Increase the reference count by 1.
  - Then, return the pointer to target in-memory inode with `dev` and `inum`.
- If not,
  - Allocate the free entry in inode cache.
  - Set the reference count of allocated cache entry to 1.
  - Then, return the pointer to allocated cache entry.

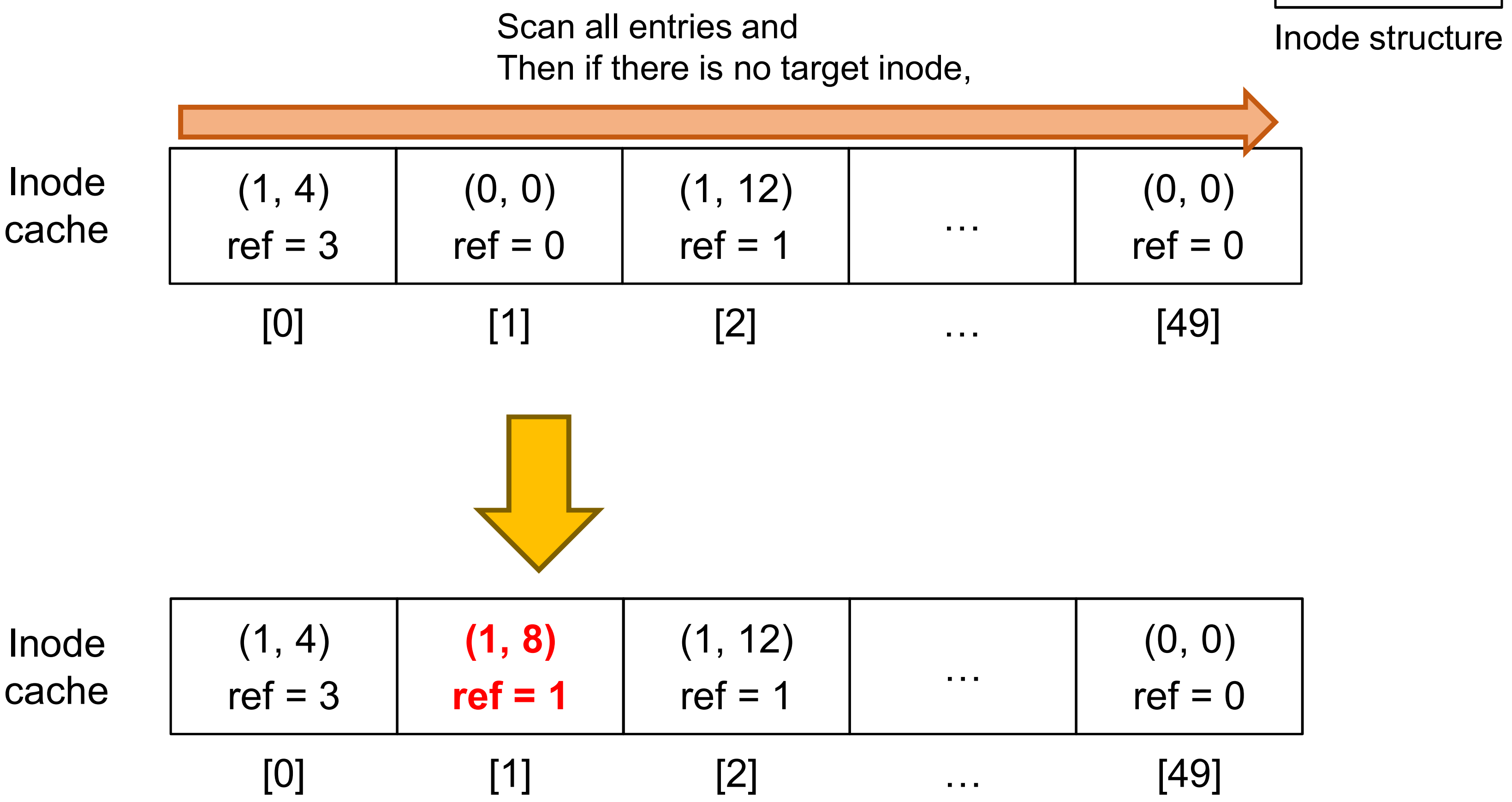
- By setting the reference count, it guarantee that the inode will stay in the inode cache and will not be deleted.

`iget()`: Target inode is not in cache.

- ex) `iget(dev=1, inum=8)`

(dev, inum) ref = 1

Inode structure



Scan all entries and
Then if there is no target inode,

Inode cache

[0]	[1]	[2]	…	[49]
(1, 4) ref = 3	(0, 0) ref = 0	(1, 12) ref = 1	…	(0, 0) ref = 0

Inode cache

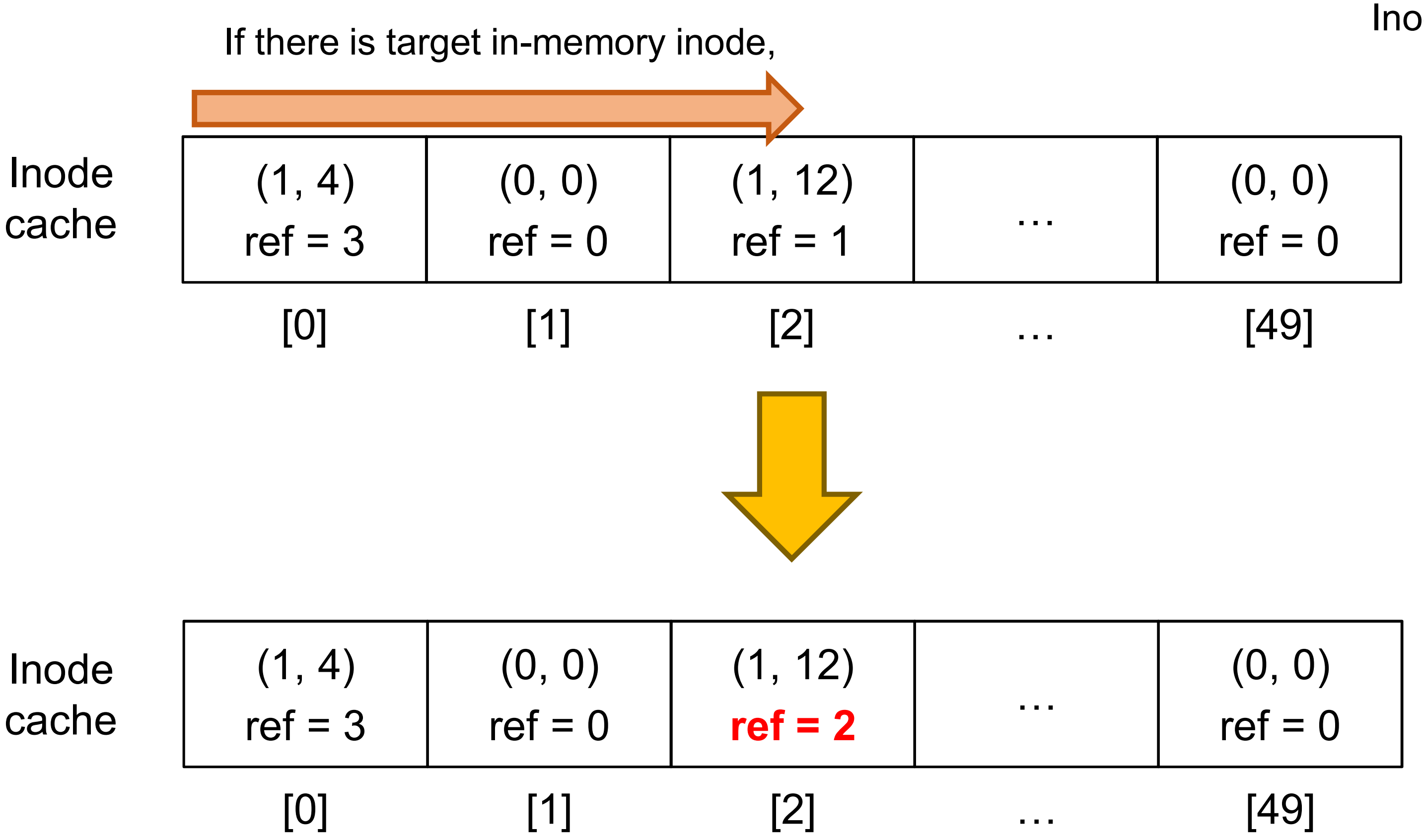
[0]	[1]	[2]	…	[49]
(1, 4) ref = 3	**(1, 8)** **ref = 1**	(1, 12) ref = 1	…	(0, 0) ref = 0

`iget()`: Target inode is already in cache.

- ex) `iget(dev=1, inum=12)`

(dev, inum)
ref = 1

Inode structure



If there is target in-memory inode,

Inode cache

[0]	[1]	[2]	…	[49]
(1, 4) ref = 3	(0, 0) ref = 0	(1, 12) ref = 1	…	(0, 0) ref = 0

Inode cache

[0]	[1]	[2]	…	[49]
(1, 4) ref = 3	(0, 0) ref = 0	(1, 12) **ref = 2**	…	(0, 0) ref = 0

`iget()`: Acquire the inode cache lock.

- Acquire the inode cache lock to prohibit other processes from modifying `dev`, `inum`, and `ref`.

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

  acquire(&icache.lock);

  // Is the inode already cached?
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
  … // Removed for saving space.
}
```

iget(): Scan all entries.

- Scan all the entries in the inode cache.

  ① Check whether the inode is with number inum on device dev.

  ② Check if the entry is free or not.

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

  acquire(&icache.lock);

  // Is the inode already cached?
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
  …  // Removed for saving space.
}
```

iget(): Target inode is already in cache.

- If the target in-memory inode with dev and inum is already in cache, increases reference count by 1.
- Then, release the icache lock and return the pointer of target inode.

```
static struct inode*
iget(uint dev, uint inum)
{
                                         …
  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;
  }
                                         …
}
```

iget(): Find the first free entry.

- If ref is 0, this entry is free.
- While scanning all the entries of inode cache, stores the first free entry in inode cache at the variable “empty”.

```
static struct inode*
iget(uint dev, uint inum)
{
  … // Removed for saving space.
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      … // Removed for saving space.
    }
    if(empty == 0 && ip->ref == 0)    // Remember empty slot.
      empty = ip;
  }
                                 …
}
```

`iget()`: Target inode is not in cache.

- Set the proper value to first free entry.
- Then, return the start address of it.

```
static struct inode*
iget(uint dev, uint inum)
{
  … // Removed for saving space.
  ip = empty;
  ip->dev = dev;
  ip->inum = inum;
  ip->ref = 1;
  ip->valid = 0;
  release(&icache.lock);

  return ip;
}
```

Same with the given `dev` and `inum`.

Reference count for this process.

`iget()`: invalid content

- It does not read the inode from the disk.
- There would be the invalid content in in-memory inode.
- xv6 separates the process of reserving a slot in inode cache from the process of reading the associated on-disk inode from the disk.

```
static struct inode*
iget(uint dev, uint inum)
{
  … // Removed for saving space.
  ip = empty;
  ip->dev = dev;
  ip->inum = inum;
  ip->ref = 1;
  ip->valid = 0;
  release(&icache.lock);

  return ip;
}
```

xv6 reads the on-disk inode when process try to acquire the per-inode lock.

void iput(struct inode *ip)

- Decreases the reference count of an in-memory inode.
- If the reference counter is 0, it frees the in-memory inode.
    - The entry in the inode cache can be recycled.
- If reference counter is 0 and `nlink` is 0 (no link), it frees the in-memory inode as well as on-disk inode.
    - To free the on-disk inode, free all the file blocks and set the type to 0.

`iput()`: Order of lock acquisition.

- First, acquire the per-inode lock to protect `vaild` and `nlink`.
- Next, acquire the inode cache lock to protect `ref`.

```
void iput(struct inode *ip){
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
    … // Removed for saving space.
    }
  }
  releasesleep(&ip->lock);
                                   …
}
```

`iput()`: The case of no link

- If `nlink` is 0 and this process is the last reference of this inode, xv6 removes this inode.

```
void iput(struct inode *ip){
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
    … // Removed for saving space.
    }
  }
  releasesleep(&ip->lock);
                                                    …
}
```

`iput()`: Delete the inode.

- `itrunc()`: Free all the file block.
- Set the type of inode to 0 to free the on-disk inode.
- `iupdate()`: Synchronize the modified in-memory inode to the on-disk inode in the disk.

```
void iput(struct inode *ip){
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    … // Removed for saving space.
    if(r == 1){
      itrunc(ip);
      ip->type = 0;
      iupdate(ip);
      ip->valid = 0;
    }
  }
  releasesleep(&ip->lock);
                                        …
}
```

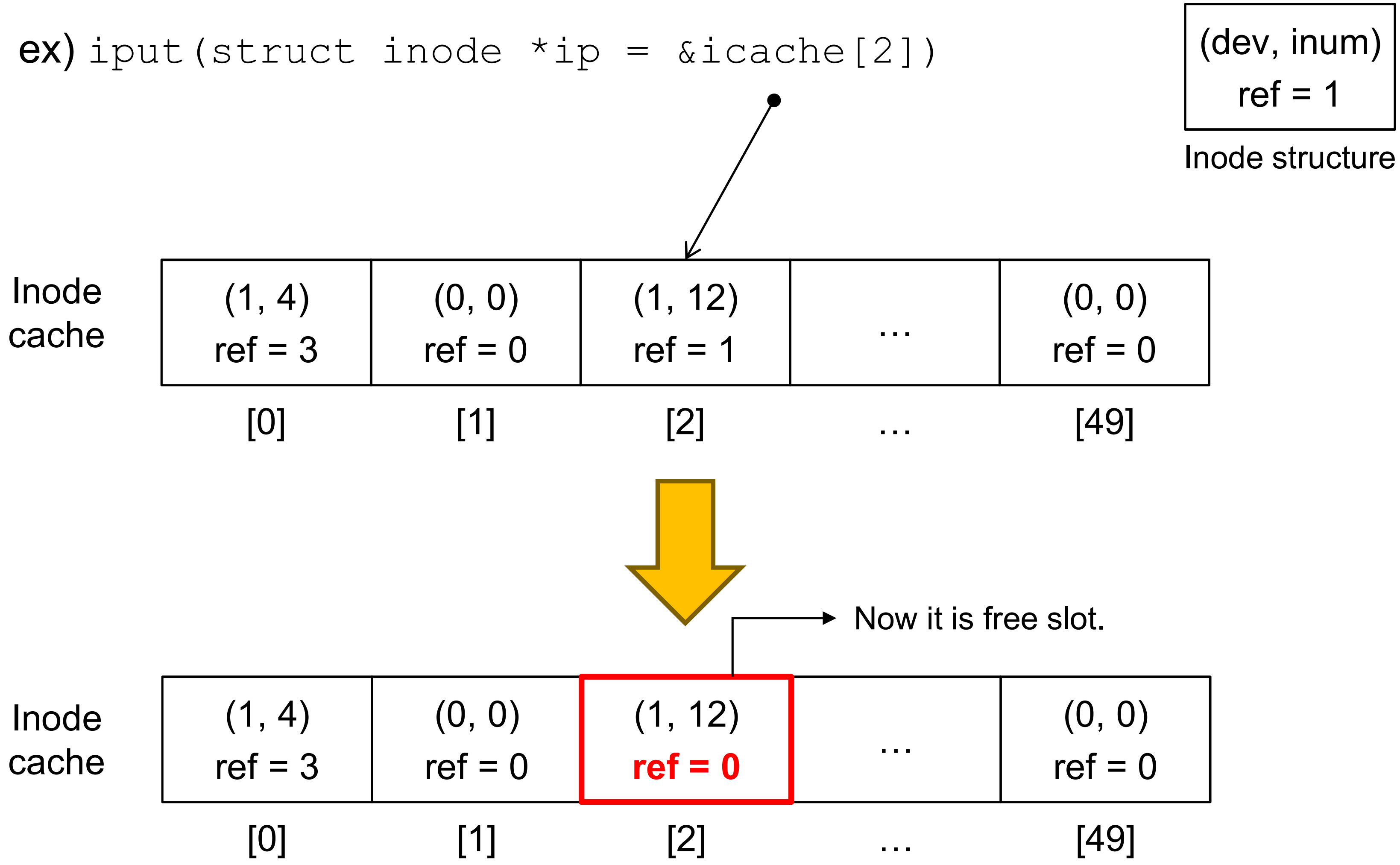
`iput()`: Drop the reference.

- Decreases the reference count of an in-memory inode.
- If `ref` becomes 0, the in-memory is free entry.
- To update `ref`, xv6 holds the inode cache lock.

```
void iput(struct inode *ip){
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    … // Removed for saving space.
  }
  releasesleep(&ip->lock);

  acquire(&icache.lock);
  ip->ref--;
  release(&icache.lock);
}
```

`iput()`: `ref` becomes 0.

- ex) `iput(struct inode *ip = &icache[2])`



(dev, inum) ref = 1

Inode structure

Inode cache

(1, 4) ref = 3	(0, 0) ref = 0	(1, 12) ref = 1	…	(0, 0) ref = 0
[0]	[1]	[2]	…	[49]

Now it is free slot.

Inode cache

(1, 4) ref = 3	(0, 0) ref = 0	(1, 12) **ref = 0**	…	(0, 0) ref = 0
[0]	[1]	[2]	…	[49]

Eviction policy

- If `ref` becomes 0, the in-memory inode is evicted immediately.
  - The inode cache never keeps the no referred on-disk inode at all even if the valid content is on the inode cache.
- The function `iget()` checks if it is target or not only when `ref` is larger than 0.

```c
static struct inode*
iget(uint dev, uint inum)
{
                                        …
  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;
    }
                                        …
}
```

Role of inode cache.

- The main job of inode cache is really synchronizing access by multiple processes, not caching.
- Multiple processes share the same in-memory inode in the inode cache.
- The shared inode structure is protected by the per-inode lock.

- So, xv6 never caches the on-disk inodes? **Yes, it does!**
  - If an inode is used frequently, the buffer cache will probably keep it in memory.

ilock(inode *ip) and iunlock(inode *ip)

- In xv6, multiple processes can share a single in-memory inode, returned by iget() function.
- To prevent race condition, xv6 uses the per-inode lock to allow only one process can access the file data and metadata at a time.
- These functions are interfaces that manipulate the per-inode lock.

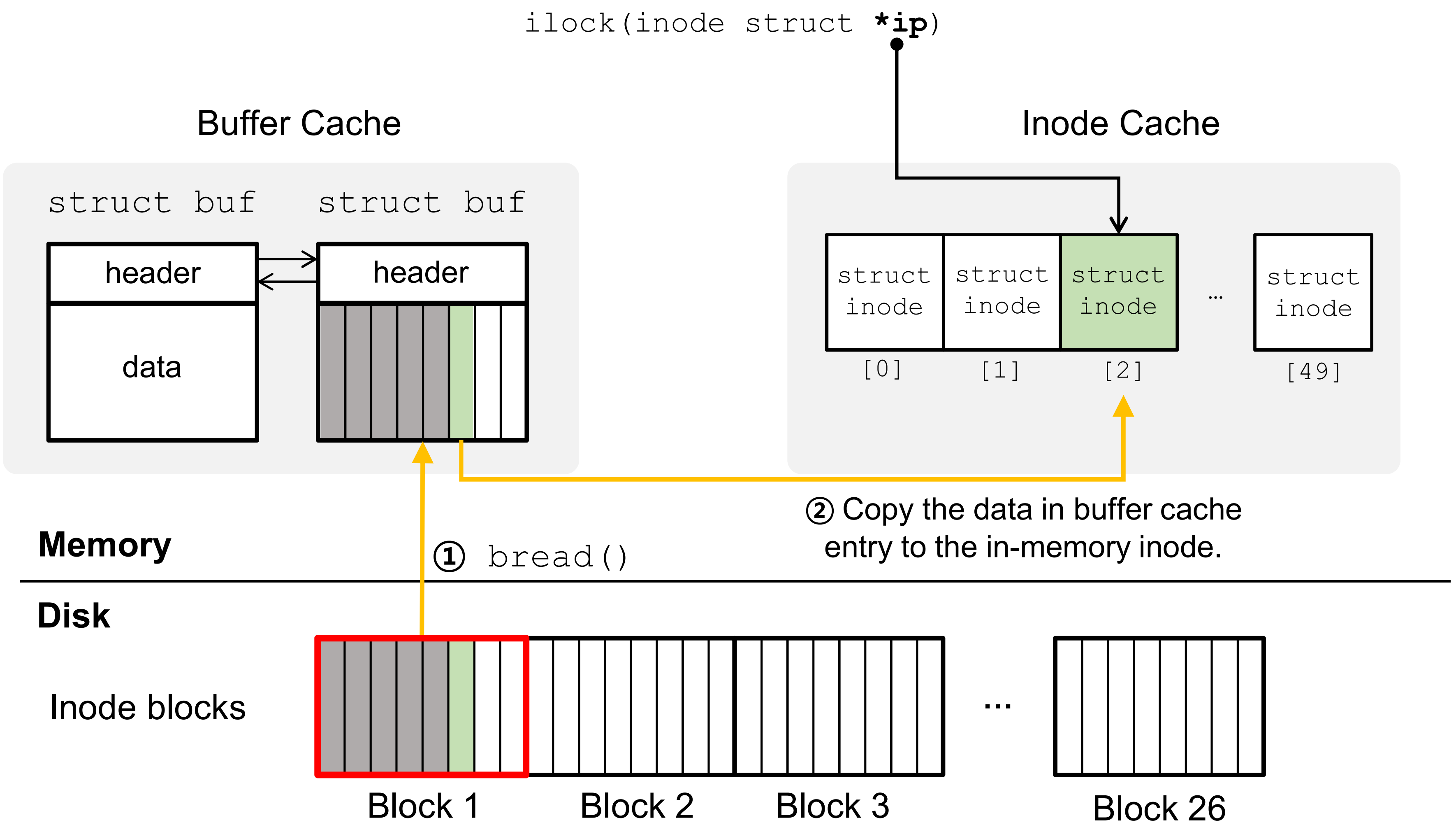
`ilock()`: Acquire the per-inode lock.

- Acquire the per-inode lock (sleep lock) for a given inode.
- Return without releasing the lock.
- The red box is executed only when the given entry in the inode cache is invalid.
  - In this box, xv6 loads the on-disk inode.
  - Per-inode lock acquisition prevents the race condition.

```
void
ilock(struct inode *ip)
{
                                        …
  acquiresleep(&ip->lock);

  if(ip->valid == 0){
                                        …
  }
}
```

Loading the on-disk inode

`ilock(struct inode *ip)`: Load the on-disk inode.



ilock(struct inode *ip): Load the on-disk inode.

- If the value of valid is 0, load the on-disk inode.
- #define IBLOCK(i, sb) ((i) / IPB + sb.inodestart)
  - Return the block number that contains the inode i.
- Read a single inode block and find the data of corresponding on-disk inode.

```
void ilock(struct inode *ip) {
                                                   …
  if(ip->valid == 0){
    bp = bread(ip->dev, IBLOCK(ip->inum, sb));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
                                                   …
  }
}
```

`ilock(struct inode *ip)`: Load the on-disk inode.

```
void ilock(struct inode *ip) {
                              …
  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 = 1;
    if(ip->type == 0)
      panic("ilock: no type");
  }
}
```

Copy the data to the in-memory inode.

iunlock()

- If the process does not hold the per-inode lock for a given inode or there is no process that refers this inode, panic occurs.
- If not, release the per-inode lock.

```
void
iunlock(struct inode *ip)
{
  if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
    panic("iunlock");

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

Inode APIs

```
struct inode*   ialloc(uint dev, short type);
void            iupdate(struct inode*);
void            itrunc(struct inode*);
```

`struct inode *ialloc(uint dev, short type)`

- Allocate the new inode at the disk and load it to icache.
- Then, return the start address of cached in-memory inode.

① Scan the inode blocks on the disk to find the free on-disk inode slot.

➔ The slot is free if the type is 0.

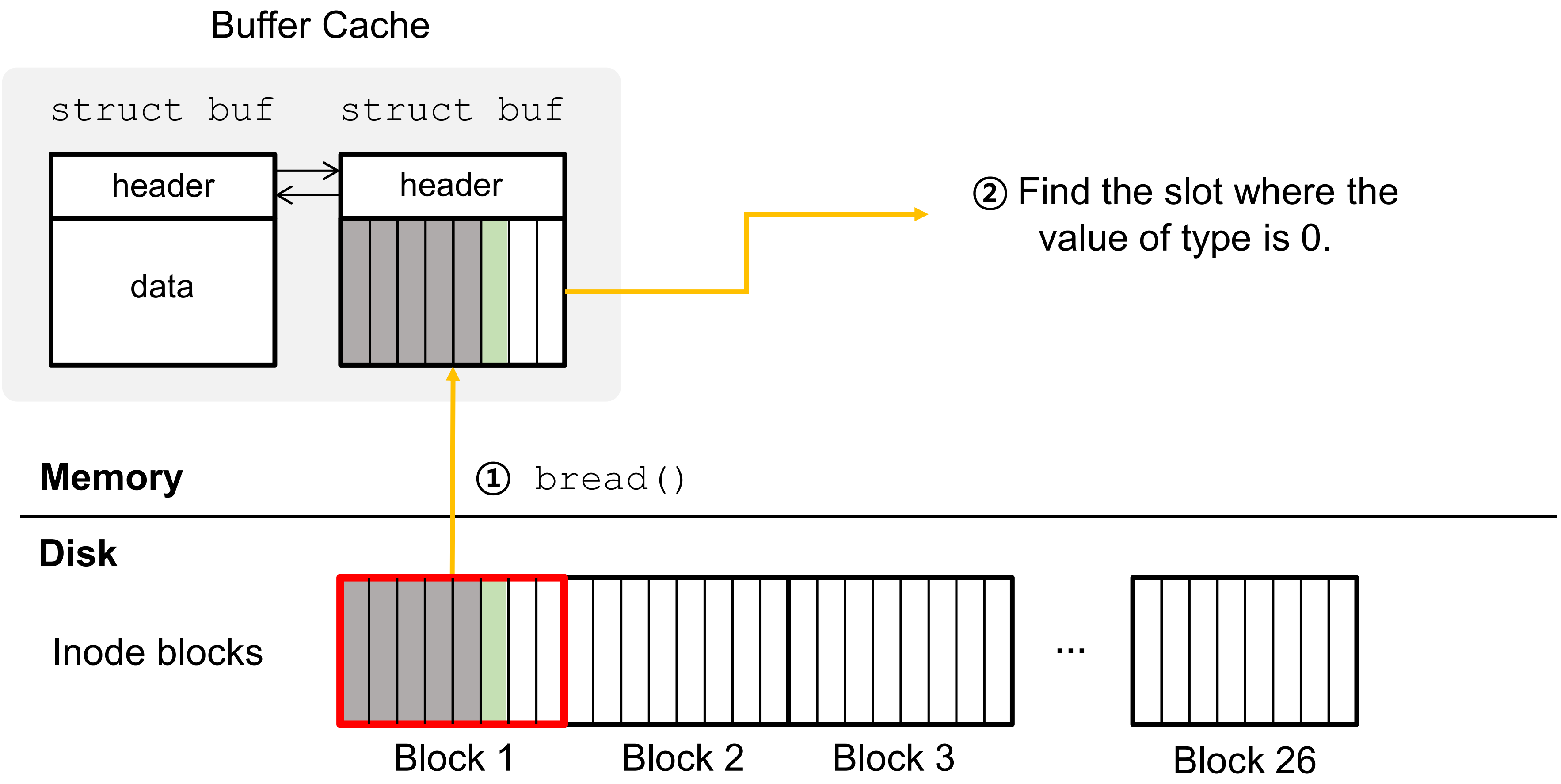
② Zero the on-disk inode and set the new type.

③ Register the buffer cache entry of the modified inode block at the in-memory log structure.

④ Call `iget()` and return the return value of `iget()`.

ialloc(uint dev,short type): Find free inode slot.

① Scan the inode blocks on the disk to find the free on-disk inode slot.



`ialloc()` : Find free inode slot. (Cont.)

- Loop check all the inodes on the disk from index 1 to index “ninodes - 1” if the type of each inode is 0 or not.
  - Index 0 is always occupied by the root directory so skip it.
- Check the inode at index `inum%8` in the inode block if the type is 0 or not.

```
struct inode*
ialloc(uint dev, short type)
{
                        …
  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){
                        …
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

Read an inode block.

`ialloc()`: Find free inode slot. (Cont.)

- xv6 calls `bread()` and `brelse()` for each on-disk inode to check if it is free or not.
- How can we optimize it?

```
struct inode*
ialloc(uint dev, short type)
{
                            …
  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){
                            …
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

`ialloc()` : Update the inode block.

- Zero the on-disk inode and set the new type.
- Register the buffer cache entry of the modified inode block at the in-memory log structure.
- Then, call `iget()`. It returns the in-memory inode for newly allocated inode.

```
struct inode*
ialloc(uint dev, short type)
{
                              …
  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      log_write(bp);
      brelse(bp);
      return iget(dev, inum);
    }
```

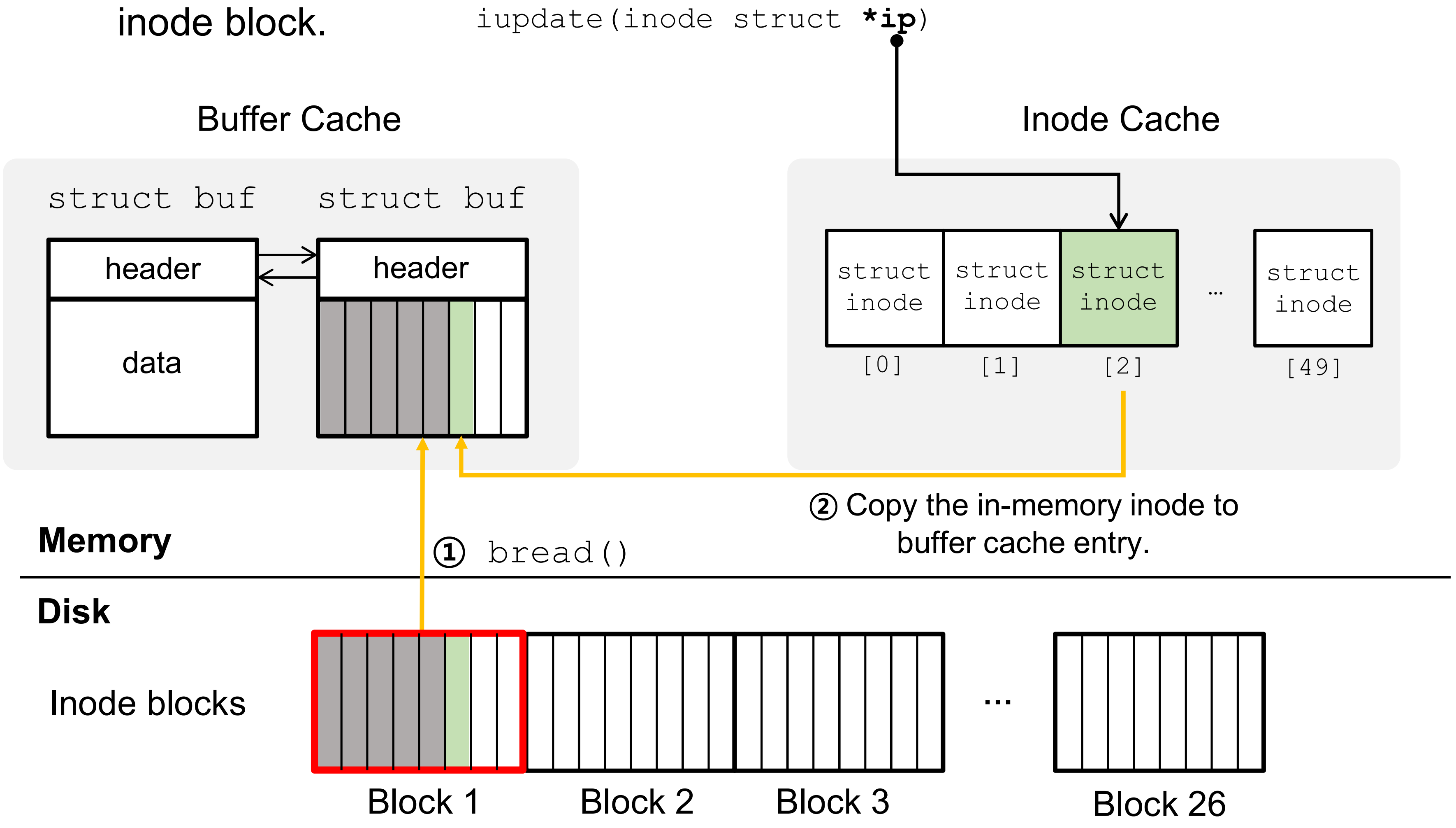
void iupdate (struct inode *ip)

① Copy the modified in-memory inode to the buffer cache entry of associated inode block.

② Register the buffer cache entry of the inode block at the in-memory log structure.

`iupdate(inode *ip)`: Copy the modified data to `buf`.

- Copy the modified in-memory inode to the buffer cache entry of associated inode block.



iupdate(): Copy the modified data to buf. (Cont.)

- `#define IBLOCK(i, sb) ((i) / IPB + sb.inodestart)`
  - Return the block number containing inumber i.

```
void
iupdate(struct inode *ip)
{
                                ...
  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);
}
```

Read an inode block.

iupdate(): Copy the modified data to buf. (Cont.)

- Copy the updated content of on-disk inode in the in-memory inode to the buffer cache entry.

```
void
iupdate(struct inode *ip)
{
                        …
  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);
}
```

iupdate(): Log the updated inode block.

- To synchronize the updated buffer cache entry with the disk, register the buffer cache entry of the inode block at the in-memory log structure.

```
void
iupdate(struct inode *ip)
{
                              …
  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);
}
```

void itrunc (struct inode * ip)

- It truncates the file.
- It frees all the file blocks for a given inode.

① Free all the valid direct blocks.

② Free all the file blocks that pointed by the indirect block.

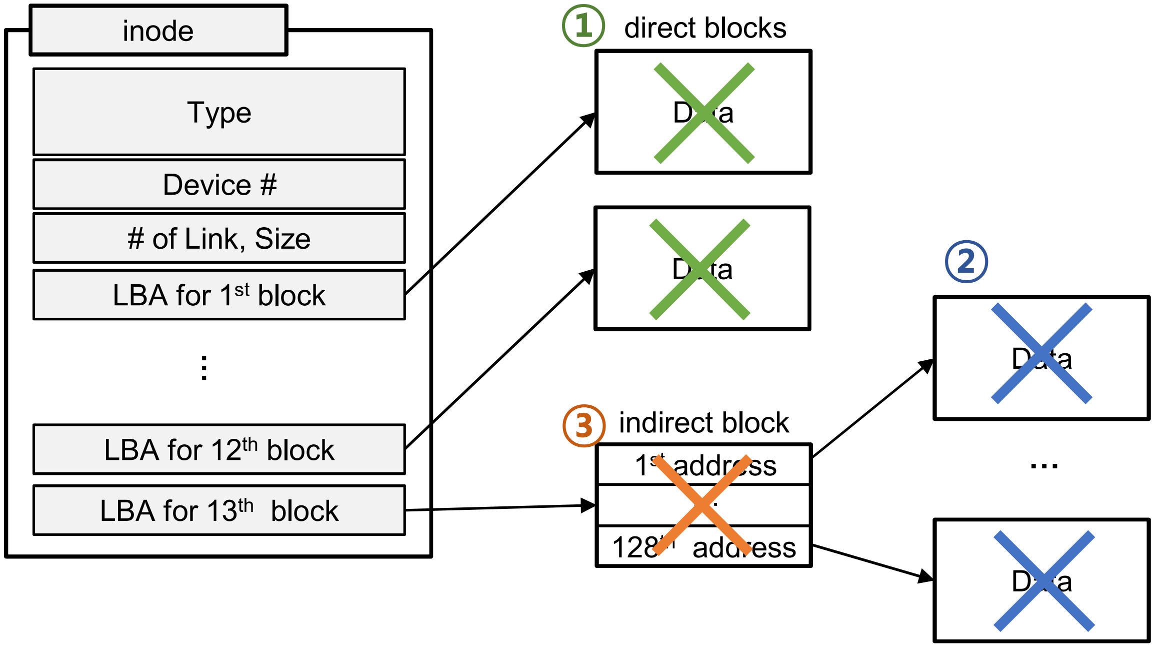
③ Free the indirect block.

④ Set the file size to 0.

⑤ Store the updated inode by calling `iupdate()`.

void itrunc (struct inode * ip)

- It frees all the file blocks for a given inode.
  - ① Free all the valid direct blocks.
  - ② Free all the file blocks that pointed by the indirect block.
  - ③ Free the indirect block.



`itrunc()`: Free all direct blocks.

- Scan the 12 entries for direct blocks. (Index 0 ~ 11).
- If the LBA is not 0, free the block and set the LBA to 0.

```
static void
itrunc(struct inode *ip)
{
                              …
  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }
                              …
}
```

itrunc(): Free all file blocks pointed by indirect block.

- Read the indirect block and scan all the LBAs in the data of indirect block.
  - If the LBA is not 0, free the file block.

```
static void
itrunc(struct inode *ip)
{
  … // Removed for saving space.
  if(ip->addrs[NDIRECT]){
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
                      …
  }
                      …
}
```

itrunc(): Free the indirect block.

- Free the indirect block.
- Set the LBA for indirect block to 0.

```
static void
itrunc(struct inode *ip)
{
  … // Removed for saving space.
  if(ip->addrs[NDIRECT]){
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT]);
    ip->addrs[NDIRECT] = 0;
  }
                         …
}
```

`itrunc()`: Free all direct blocks.

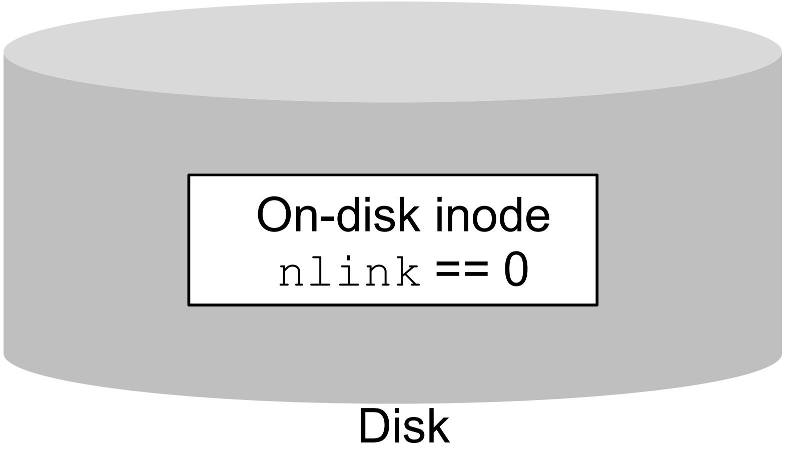
- Set the file size to 0.
- Synchronize the updated inode to the disk by calling iupdate().

```
static void
itrunc(struct inode *ip)
{
      … // Removed for saving space.
  ip->size = 0;
  iupdate(ip);
}
```

`iput(struct inode *ip)` and crash

- `iput(struct inode *ip)`
  - If **`nlink` is 0** and this process is **the last reference of this inode**, xv6 removes this inode.

- Although `nlink` is 0, xv6 waits till the `ref` becomes 0 to remove the inode.
- **What happened if the crash occurs** before `ref` becomes 0?
  - The on-disk inode without the references to it still stored on the inode block.



fsck and orphan list

There are two approaches to solve this problem.

① fsck style

- After crash, scan all the inodes.
- Remove all inodes with no link (`nlink` == 0).

② Orphan list

- Maintain the list of inodes with no link.
- Remove the inodes in this orphan list.

readi() and writei()

- readi(inode *ip, char*dst, uint off, uint u): read the data from the inode.
- writei(inode *ip, char *dst, uint off, uint n): write the data to the inode.
- It uses interfaces of block cache layer, logging layer, and inode layer.
  - bread() and brelse()
  - begin_op(), end_op(), and log_write()
  - iupdate()

readi(inode *ip, char*dst, uint off, uint n)

- Read n byte to dst from off position of ip.
  - First, load the data from the disk to a buffer cache entry.
  - Then, copy the data in buffer cache entry to the user buffer.
  - Repeat it until read n bytes from the disk.
- It reads the file data so caller must hold the per-inode lock.

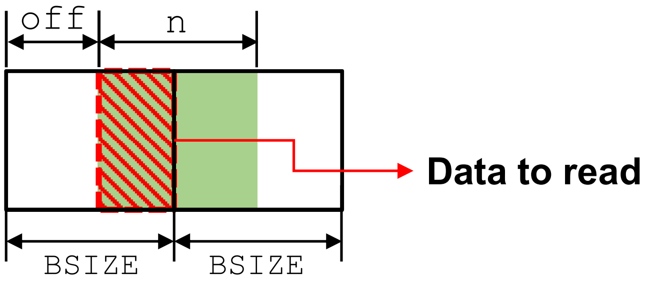
`readi()`: Load the data to buffer cache entry.

- `bmap()`: Return the disk block address of nth file block in inode.
- `bread()`: allocate the buffer cache entry and load the data from the disk to this entry.

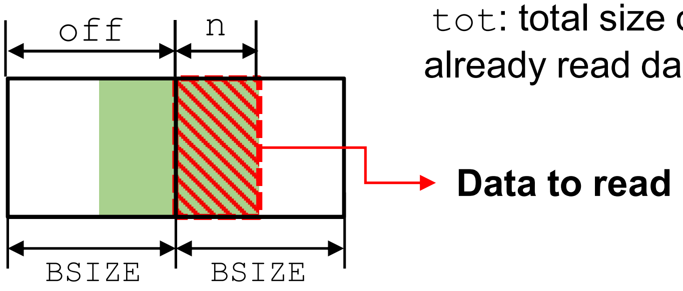
```
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
                                            …
  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

`readi()`: Calculate the data size to read.

Copy "`BSIZE - off % BSIZE`" byte.



Copy "`n - tot`" byte.



`tot`: total size of already read data

```
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
                                 …
  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

readi(): Copy the data to user buffer.

- Copy the data in buffer cache entry to the user buffer.
- Repeat it utill read and copy n byte from the disk to the user buffer.

```
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
                              …
  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

writei(inode *ip, char *dst, uint off, uint n)

- This function writes `n` byte of `dst` to `off` position of `ip`.
  - First, load the data from the disk to a buffer cache entry.
  - Next, copy the data in user buffer to the buffer cache entry.
  - Then, register the buffer cache entry to in-memory log structure.
  - Repeat it until copy `n` bytes to the buffer cache entry.
- It updates the file data and metadata so caller must hold the per-inode lock.

`writei()`: Load the data to buffer cache entry.

- `bmap()`: Return the disk block address of nth file block in inode.
- `bread()`: allocate the buffer cache entry and load the data from the disk to this entry.

```
int
writei(struct inode *ip, char *src, uint off, uint n)
{
                                 …
  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    brelse(bp);
  }
                                 …
  return n;
}
```

`writei()`: Copy the data to buffer cache entry.

- Copy the data in user buffer to the buffer cache entry.
- Register the buffer cache entry to in-memory log structure.

```
int
writei(struct inode *ip, char *src, uint off, uint n)
{
                                …
  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    brelse(bp);
  }
                                …
  return n;
}
```

writei(): log_write() and brelse()

- What happened if you switch the line 502 and line 503?
  - Call `log_write(bp)` after calling `brelse(bp).`

```
481 int
482 writei(struct inode *ip, char *src, uint off, uint n)
483 {
                                   …
498   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
499     bp = bread(ip->dev, bmap(ip, off/BSIZE));
500     m = min(n - tot, BSIZE - off%BSIZE);
501     memmove(bp->data + off%BSIZE, src, m);
502     log_write(bp);
503     brelse(bp);
504   }
                                   …
510   return n;
511 }
```

The buffer cached entry can be evicted.
If the other data is loaded in evicted cache entry before commit,
unexpected data can be written to the log area.

`writei()`: Repeat writing until copying n bytes.

- Repeat it until copy `n` bytes to the buffer cache entry.

```
int
writei(struct inode *ip, char *src, uint off, uint n)
{
                                …
  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    brelse(bp);
  }
                                …
  return n;
}
```

writei(): Enlarge the file size.

- If the `offset` is larger than the file size after writing the data, update the file size.
- It calls `iupdate()` after modifying the in-memory inode to synchronize the modified in-memory inode to the disk.

```
int
writei(struct inode *ip, char *src, uint off, uint n)
{
                              …
  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    … // Removed for saving space.
  }

  if(n > 0 && off > ip->size){
    ip->size = off;
    iupdate(ip);
  }
  return n;
}
```

filewrite(struct file *f, char *addr, int n)

- Write the n byte data from addr to the file that pointed by f.
- It calls the writei().
  - Use ilock() and iunlock() to protect the inode.
  - Use begin_op() and end_op() to synchronize atomically the several updated blocks with the disk.

Putting everything together: `filewrite()`

- It calls `writei()` to write the data to the file.

```
int
filewrite(struct file *f, char *addr, int n)
{
                                        …
    while(i < n){
                                        …

      begin_op();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      end_op();
                                        …
    }
                                        …
}
```

Putting everything together: `filewrite()`

- The caller must hold the per-inode lock of the inode when writing the data.
- After writing the data, lock must be released.

```
int
filewrite(struct file *f, char *addr, int n)
{
                                        …
    while(i < n){

      begin_op();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      end_op();
                                        …
    }
                                        …
}
```

Putting everything together: `filewrite()`

- `struct file`: the data structure to represent a file
- `end_op()` writes all buffer cache entry registered in in-memory log structure by `log_write()` to the log area on the disk.

```
// file.h
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

Putting everything together: `filewrite()`

- To guarantee the file system consistency even if crash occurs at the middle of function `writei()`, it calls `begin_op()` and `end_op()`.
- `end_op()` writes all buffer cache entry registered in in-memory log structure by the function `log_write()` to the log area on the disk.

```
int
filewrite(struct file *f, char *addr, int n){

      begin_op();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      end_op();
                             …
    }
                             …
}
```

Summary

- Inode structure: On-disk and in-memory inode.
- Code:
  - `iget(),iput(), ilock(),` and `iunlock()`
  - `ialloc(), iupdate(),` and `itrunc()`
- Protection
  - Inode cache lock: spin lock, protect the changes in the in-memory field of the inode
  - Per-inode lock: sleep lock, synchronize the accesses of multiple processes.
- Real examples that read or write the file data through inode.
  - `readi(),writei(),` and `filewrite()`