Data Storage Technologies & Networks (CS C446, CS F446 & IS C446)

Lecture 29– Storage

Directories

- Are the files that give file system its hierarchical structure.
- They play important role in converting file name in to inode number.
- Directory is a file whose content is sequence of entries
- Each entry consist of inode number and the name of a file contained in the directory.
- UNIX system restrict component names to a maximum of 14 characters, and 2 byte inode number.
- Size of one directory entry is 16 bytes.

Directories

- Every directory contain
  - file name and inode number of all files in that directory
  - dot and the inode number
    - represents this directory
  - double dots and the parent inode number
    - represents this directory’s parent directory

- Access Permissions
  - Read → allows a process to read the directory
  - Write → allows process to create, remove directory entries.
  - Execute → allows process to search the directory for a file name.

Conversion of a path name to an Inode

- Algorithm namei parses the path name one component at a time
- All path name searches start from the current directory of the process unless the path name start with ‘/’(start from the root).
- The current directory is stored in the process u area.
- Root inode is stored in global variable

- Example
  - fopen(“/etc/passwd”,”r”)
  - fopen(“DSTN/Test2/Que2.pdf”,”r”) where current directory is “/root/CSC446_ISC446”

```
Algorithm namei
Input: path name
Output: locked inode
{
    if (path name starts from root)
        working inode = root inode (algorithm iget);
    else
        working inode = current directory inode (algorithm
        iget);
    while (there is more path name)
    {
        read next path name component from input;
        verify that working inode is of directory, access
        permissions OK;
        if (working inode is of root and component is "..")
            continue;
        read directory (working inode) by repeated use of
        algorithm bmap, bread, and brelse;
        if (component matches an entry in directory (working
        inode))
        {
            get inode # for matched component;
            release working inode (algorithm iput);
            working inode = inode of matched component
            (algorithm iget);
        }
        else
            return (no inode);
    }
    return (working inode);
}
```

Algorithm for conversion of a path name to an inode

- Kernel make sure that working inode is a directory inode.
- Kernel does a linear search in directory entry to find the path name component.
- If found a match, record the inode number of the matched directory entry and release the block (brelse) and old working inode (iput).

Super block

- Consists of following fields
  - Size of the file system
  - Number of free blocks in the file system
  - List of free blocks available in the file system
  - Index of the next free block in the free block list
  - Size of the Inode list
  - Number of free Inodes in the file system
  - List of free Inodes in the file system
  - Index of the next free Inode in the free Inode list
  - Lock fields for the free block and free Inode lists
  - A flag indicating that the super block has modified.

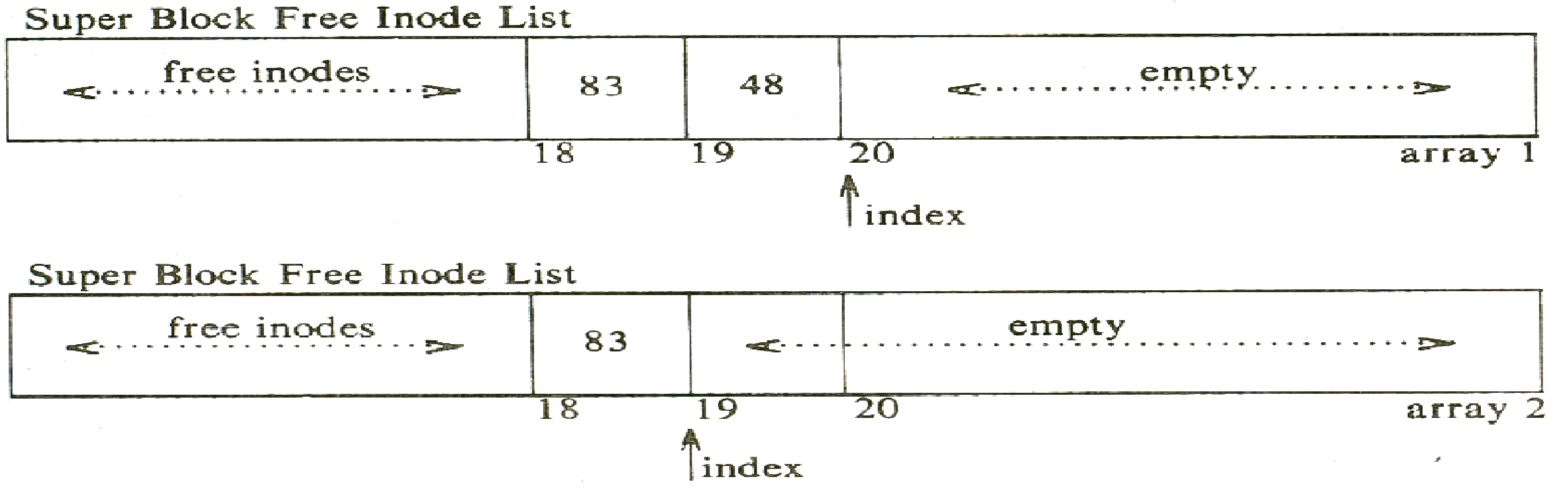
Kernel periodically writes super block into the disk for consistency.

Inode assignment to a new file

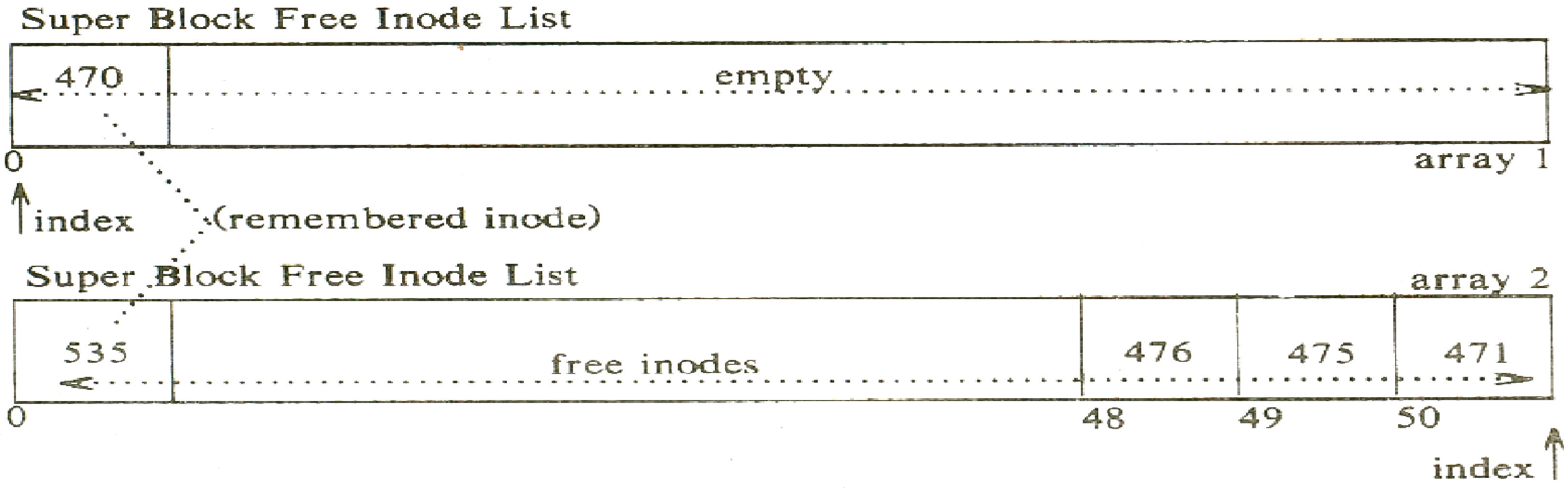
- iget algorithm allocate a known inode, whose inode number was previously determined.
- ialloc algorithm assigns a disk inode to a newly created file.
- File system contain a linear list of inodes
- An Inode is free if its type field is zero.
- To improve the performance (to avoid sequential search) the file system super block contains an array to cache the number of free inodes in the file system.
- Kernel make sure that no other process has locked access to the super block free Inode list.
- If the list of Inode numbers in the super blocks are not empty, kernel assigns the next Inode number, allocates a free in-core Inode for the newly assigned disk Inode (iget), copies the disk inode to the in-core copy, initialize the fields in the inode and returns the locked inode.
- It updates the disk inode to indicate that the Inode is now in use

```
Algorithm ialloc
Input: file system, Output: locked inode
{       while (not done)
        {
                if (super block locked)
                {
                        sleep (event supewr block becomes free);
                        continue;
                }
                if (inode list in super block is empty)
                {
                        lock super block;
                        get remembered inode for free inode search;
                        search disk for free inodes until super block full,
                        or no more free inodes (bread and brelse algo)
                        unlock super block;
                        wake up (event super block becomes free);
                        if (no free inodes found on disk)
                                return (no inode);
                        set remembered inode for next free inode search;
                }
                get inode number from super block inode list;
                get inode (algorithm iget);
                if (inode not free after all)
                {
                        write inode to disk;
                        release inode (algorithm iput);
                        continue;
                }
                initialize inode;
                write inode to disk;
                decrement file system free inode count;
                return (inode);
        }
}   // Algorithm for Assigning New Inodes
```

super block.



(a) Assigning Free Inode from Middle of List



(b) Assigning Free Inode - Super Block List Empty

Figure 4.13. Two Arrays of Free Inode Numbers

- If the super block list of free Inodes is empty, the kernel searches the disk and places as many free Inode numbers as possible into the super block.
- Kernel stores the highest inode number assigned (called “remembered”). The search to free inode number starts from remembered
- Whenever kernel assigns a disk Inode, it decrements the free Inode count recorded in the super block.

```
Algorithm iget
Input: file system inode number
Output: linked inode
{
    while (not done)
    {
        if (inode in inode cache)
        {
            if (inode locked)
            {
                sleep (event inode becomes unlocked);
                continue;
            }
            if (inode on inode free list)
                remove from free list;
            increment inode reference count;
            return inode;
        }
        if (no inode on free list)
            return error;
        remove new inode from free list;
        reset  inode number and file system;
        remove inode from old hash queue, place on new one;
        read inode from disk (bread algorithm);
        initialize inode (reference count to 1);
        return inode;
    }
}
```

Algorithm for allocation of In-core inodes

Accessing Inodes

- Kernel identifies a particular inode by their file system and inode number and allocates in-core inodes at the request of high level algorithms.
- Algorithm iget (similar to getblk) allocates an in core copy of an inode
- Kernel maps the device number and inode number into a hash queue and searches the queue for the inode.
- If it cannot find the inode, it allocates one from the free list and locks it.
- Kernel then prepares to read the disk copy in to the in-core copy.

- Calculating logical block number =

  ((inode number – 1)/ (number of inodes per block) + start block of inode list)
- Kernel reads the block using bread algorithm
- Calculating byte offset of the inode in the block =

  ((inode number – 1) modulo (number of inodes per block)) * size of disk inode

Releasing Inodes

- Decrements its in-core reference count
- If the count drops to 0, the kernel writes the inode to disk if the in-core copy differs (file data, owner or access permission has changed) from the disk copy.
- Kernel places the inode on the free list of inodes.
- Kernel may also release all the data blocks associated with the file and free the inode if the number of links to the file is 0.

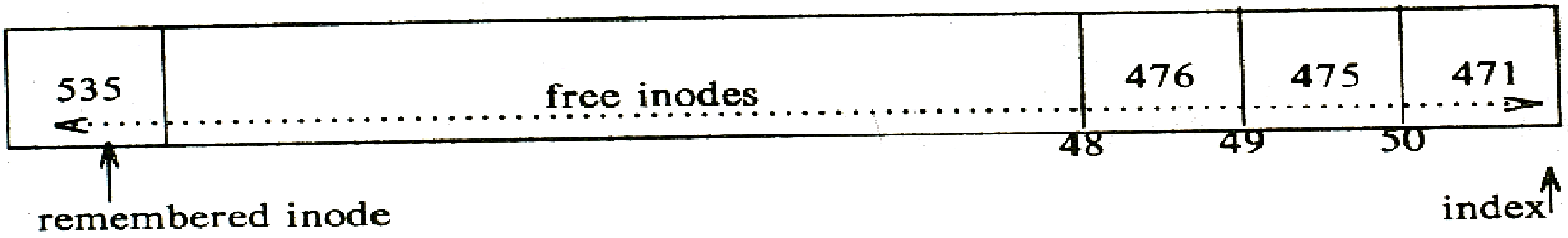
```
Algorithm iput
Input: pointer to in-core inode
Output: none
{
        lock inode if not already locked;
        decrement inode reference count;
        if (reference count = = 0)
        {
                if (inode link count = = 0)
                {
                        free disk blocks for file (algorithm free);
                        set file type to 0;
                        free inode (algorithm ifree);
                }
                if (file accessed or inode changed or file changed)
                        update disk inode;
                put inode on free list;
        }
        release inode lock;
}
```

Releasing an Inode

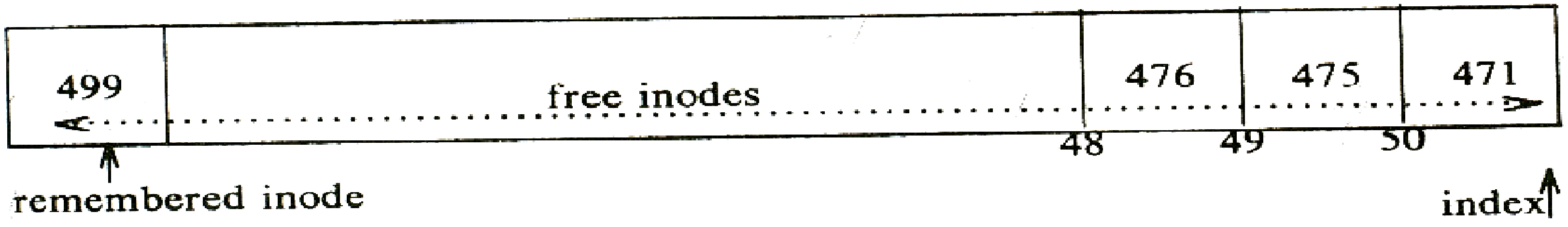
Freeing Inode

- Increment the total number of available Inodes in the file system.
- Kernel checks the lock on the super block
- If locked, avoids race condition by returning immediately (the Inode number is not put in to the super block, but can be found on disk and is available for reassignment).
- If not locked, kernel checks if it has room for more inode numbers (if it has, place it in the list and returns)
- If the list of free inodes is full, then kernel may not save the newly freed inode.

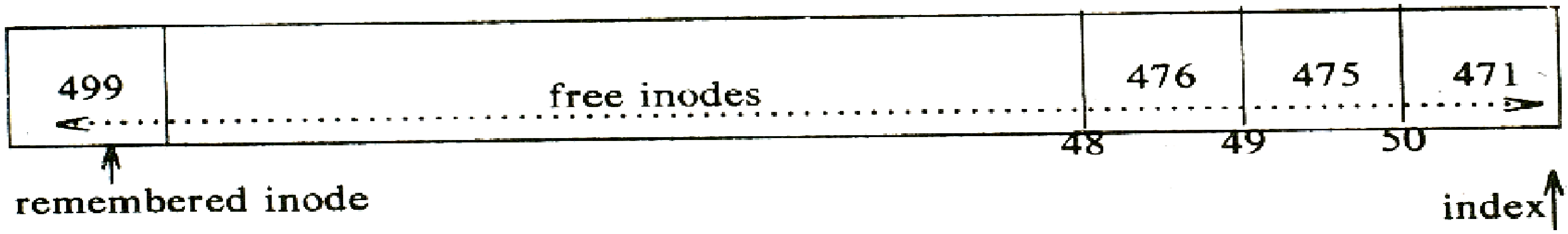
```
Algorithm ifree
Input: file system inode number
Output: none
{
        increment file system free inode count;
        if (super block locked)
                return;
        if (inode list full)
        {
                if (inode # less than remembered inode for search)
                        set remembered inode for search = input inode #;
        }
        else
                store inode # in inode list;
        return;
}
```



(a) Original Super Block List of Free Inodes



(b) Free Inode 499



(c) Free Inode 601

Figure 4.15. Placing Free Inode Numbers into the Super Block

Allocation of disk blocks

- File system super block contains an array which is used to cache the number of free disk blocks in the file system
- mkfs (make file system)utility program organizes data blocks of a file system in linked list.
- Each link of the list is a disk block that contain an array of free disk block numbers, and one array entry is the number of next block of the linked list.

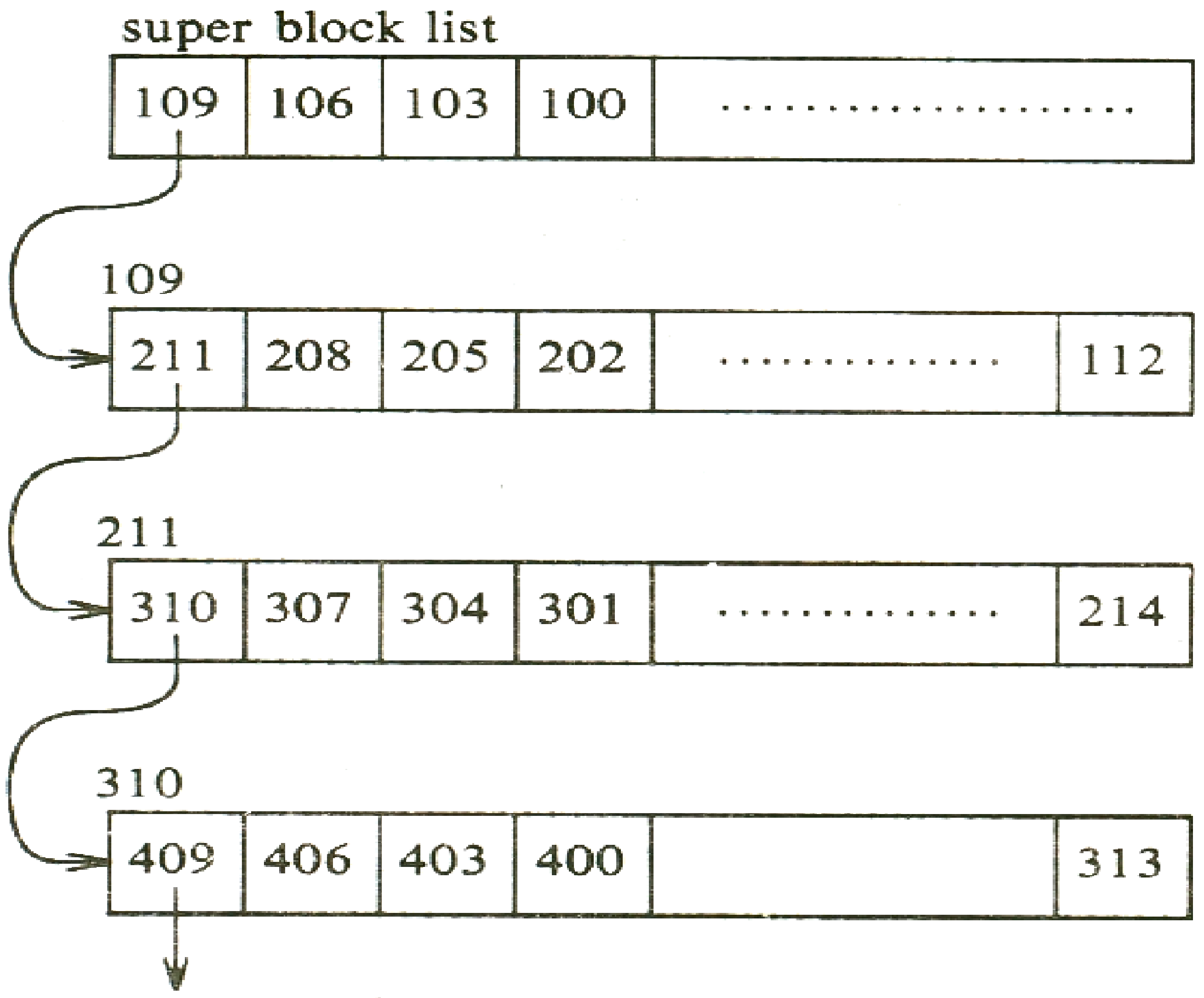


Figure 4.18. Linked List of Free Disk Block Numbers

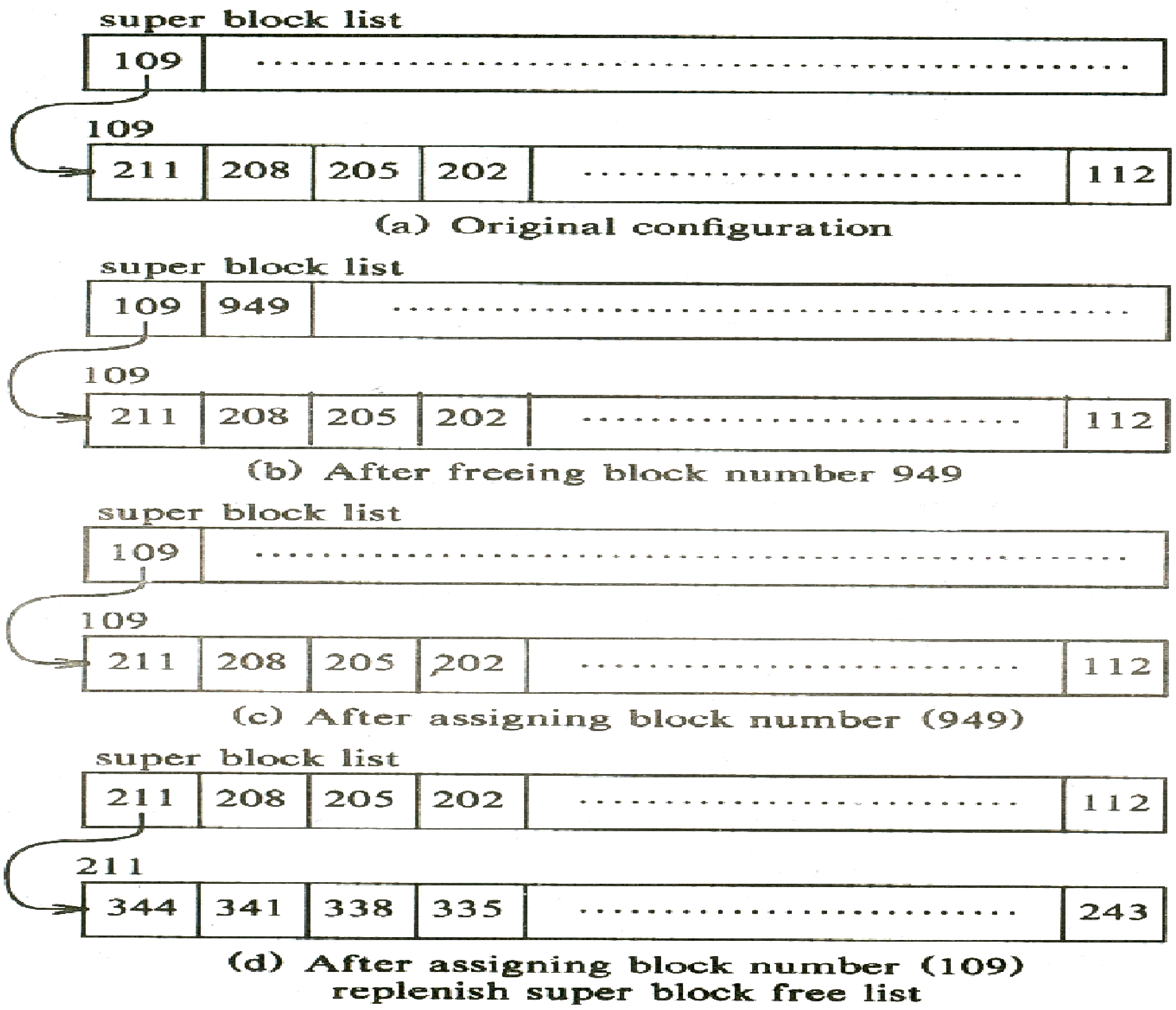
- When kernel wants to allocate a block from a file system, it allocates the next available block in the super block list.
- If allocated block is the last available block in the super block cache, kernel treat it as a pointer to a block that contains a list of free blocks.
- If the file system has no free blocks, then the calling process receives an error.

free algorithm

- If super block list is not full, the block number of the newly freed block is placed on the super block list.
- If super block is full, then the newly freed block becomes a link block.
  - Kernel writes the super block list into the block and writes block to disk.
  - It then places the block number of the newly freed block in the super block list.

```
Algorithm alloc
Input: file system number
Output: buffer for new block
{
        while (super block locked)
                sleep (event super block not locked);
        remove block from super block free list;
        if (removed last block from free list)
        {
                lock super block;
                read block just taken from free list (bread);
                copy block # in block into super block;
                release block buffer (brelse)
                unlock super block;
                wakeup processes (event super block not locked);
        }
        get buffer for block removed from super block list (getblk);
        zero buffer contents;
        decrement total count of free blocks;
        mark super block modified;
        return buffer;
}
```

Algorithm for Allocating Disk Block



(a) Original configuration
(b) After freeing block number 949
(c) After assigning block number (949)
(d) After assigning block number (109) replenish super block free list

Figure 4.20. Requesting and Freeing Disk Blocks