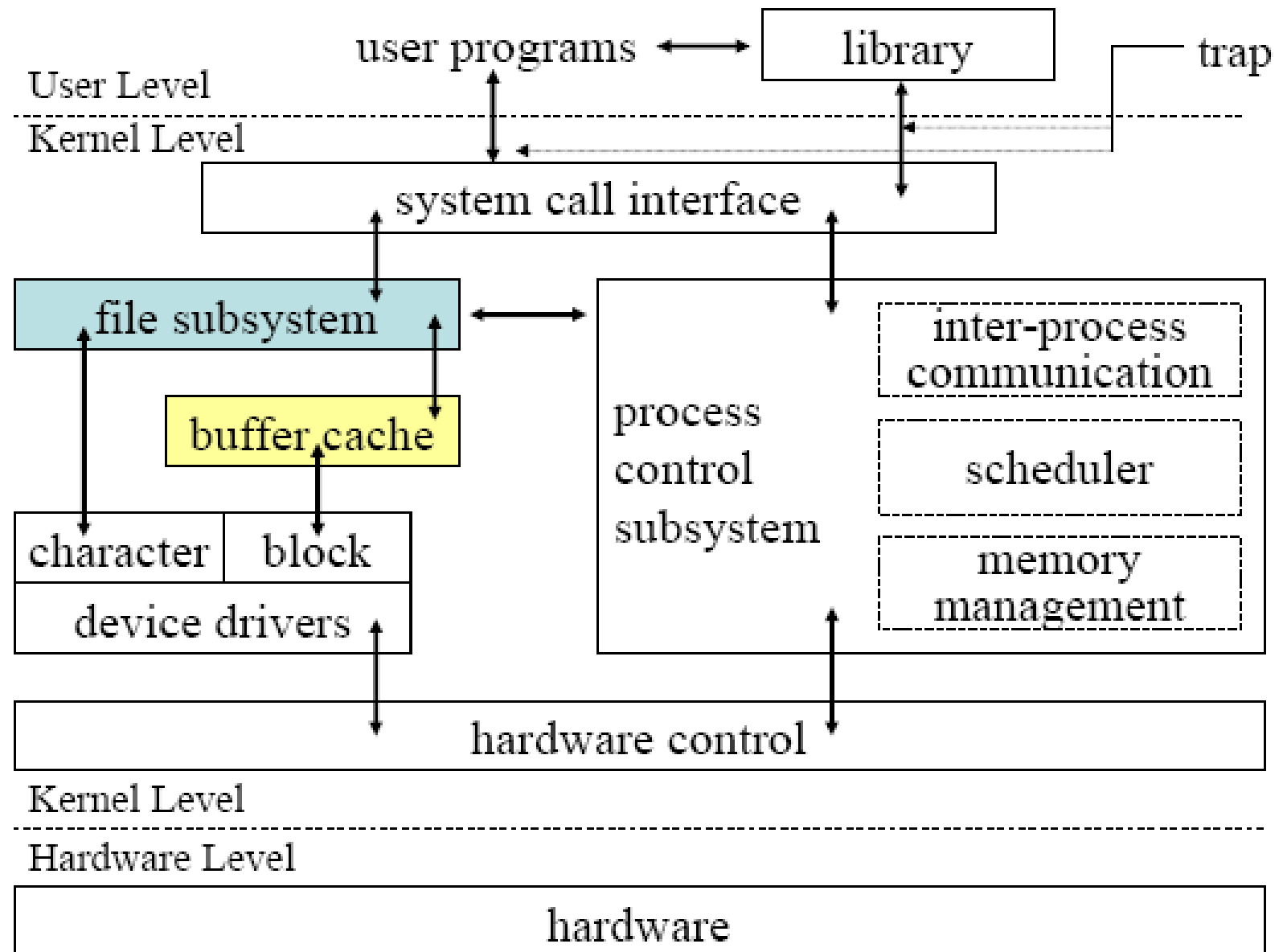
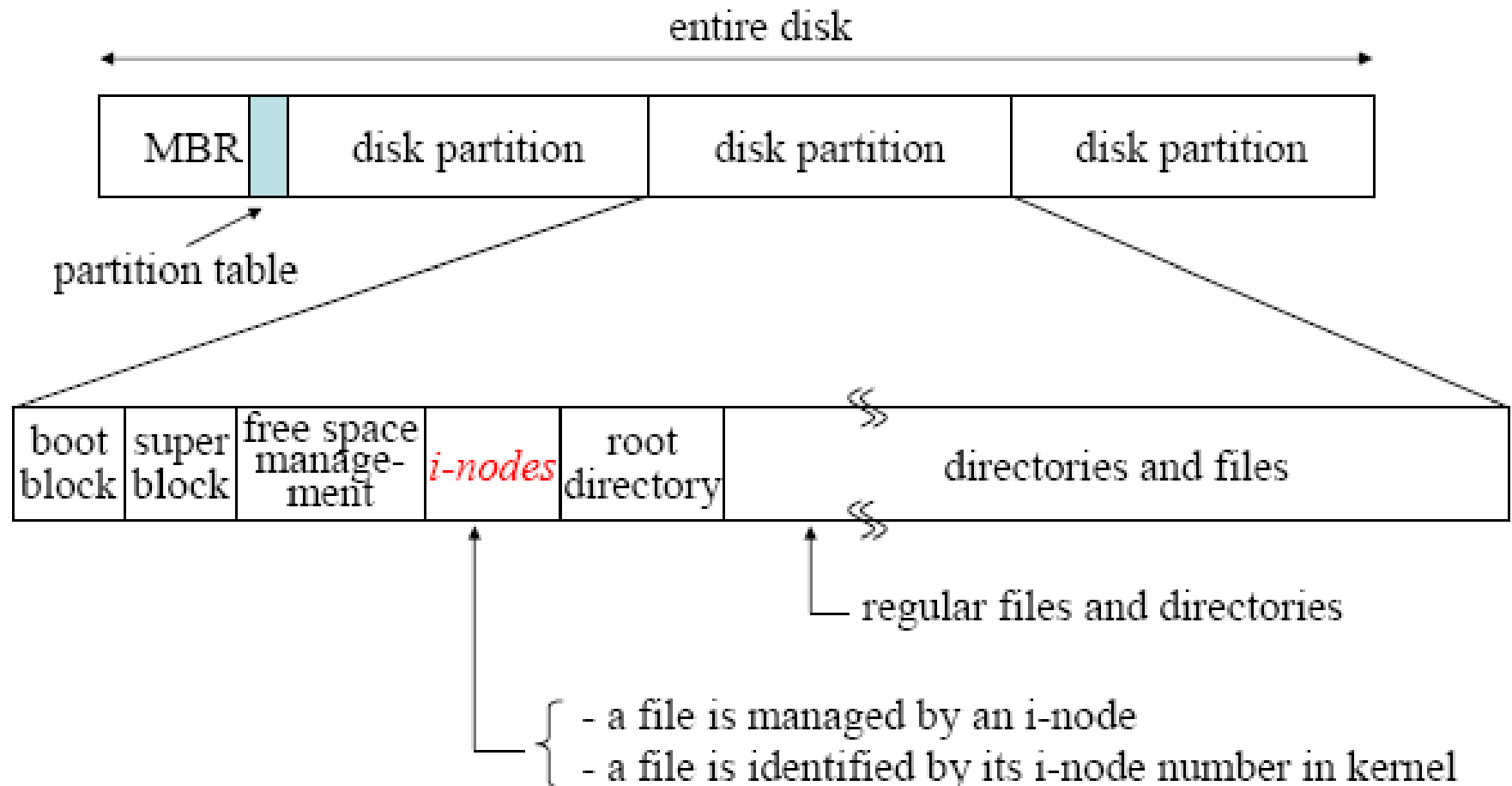


# Internal Representation of File

# Block Diagram of the System Kernel

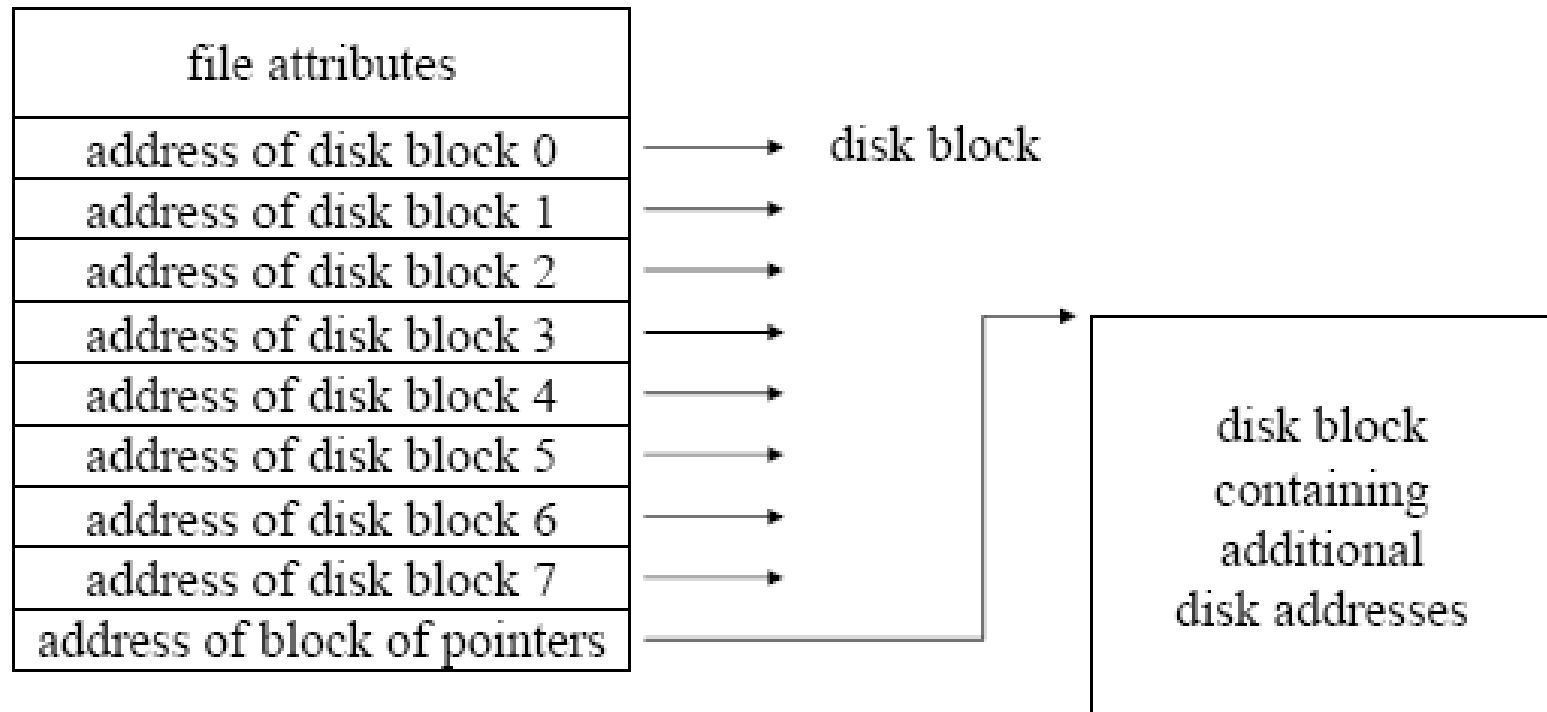


# Example of File System Layout



# i-node

- An **i-node** (index-node) lists the attributes and disk addresses of the file's blocks.
  - simple example:



# An Example on UNIX

- Access to /usr/tera/mbox
  - i-node table resides at the fixed portion on the disk.
  - finding root directory, and then finding /usr, ...

root  
directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

i-node 6  
(/usr)

attribute
132

block 132  
(/usr)

6	.
1	..
19	alice
26	tera
51	bob

i-node 26  
(/usr/tera)

attribute
406

block 406  
(/usr/tera)

26	.
6	..
64	.cshrc
92	books
60	mbox



/usr/tera/mbox is i-node 60

# File System Algorithms

- *namei* converts a user-level path name to an *i-node*.
- *iget* returns a previously identified i-node, possibly reading it from disk via buffer cache.
- *iput* releases the i-node.
- *bmap* sets kernel parameters for accessing a file.
- *alloc* and *free* allocate and free disk blocks.
- *ialloc* and *ifree* assign and free i-nodes for files.

namei			alloc   free		ialloc   ifree	
iget	iput	bmap				
buffer allocation algorithms						
getblk		brelse	bread	breada	bwrite	

# Disk I-nodes

- *I-nodes* exist in a static form on disk, and the kernel reads them into an *in-core i-node*.
- Disk i-nodes consist of the following fields:
  - file owner identifier: an individual owner and a group owner
  - file type: regular, directory, character/block special, etc.
  - file access permissions: owner/group/others read/write/exec
  - file access times: last modified/accessed, i-node last modified
  - number of links to the file: number of names
  - table of contents for the disk address
  - file size

# In-core I-nodes (1)

- The in-core copy of the i-node contains the following fields in addition to the disk i-node:
  - **status** of the in-core i-node, indicating:
    - the i-node is locked
    - a process is waiting for the i-node to become unlocked
    - in-core i-node differs from the disk copy as a result of a change to the data in the i-node or change to the file data
  - **logical device number** of the file system
  - **i-node number**
  - **pointers to other in-core i-nodes**: hash queue and free list
  - **reference count**: number of instances of the file that are active (such as when *opened*)

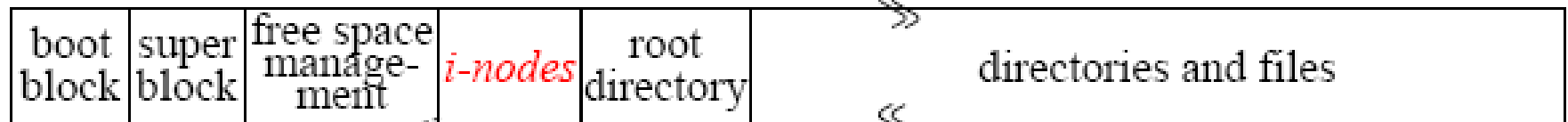


## In-core I-nodes (2)

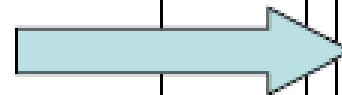
- Many fields in the in-core i-node are analogous to fields in the buffer header.
- I-node lock prevents other processes from accessing the i-node.
  - Other processes set a flag when attempting to access it.
- The most important difference between in-core i-node and buffer header is the **reference count**.
  - An i-node is active when a process allocate it, such as when *opening* a file.
  - An i-node is on the free list only if its reference count is 0.
  - The free list of i-nodes serves as a cache of inactive i-nodes.
  - Buffer header has no reference count; it is on the free list if and only if it is unlocked

# Disk I-nodes and In-core I-nodes

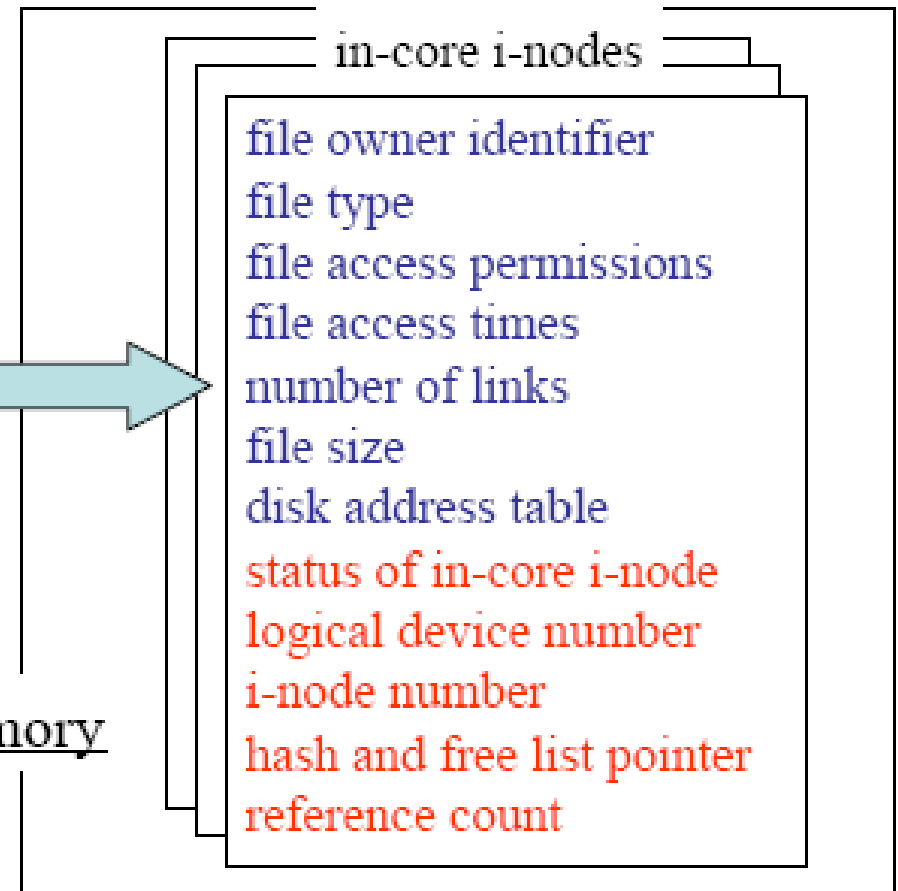
disk



disk i-nodes



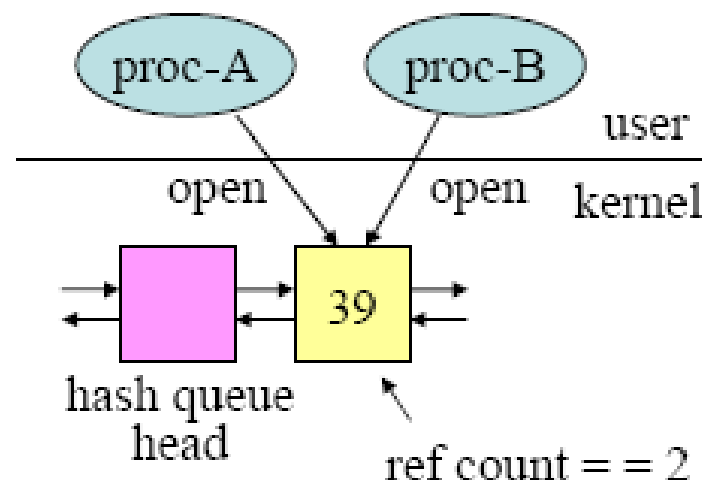
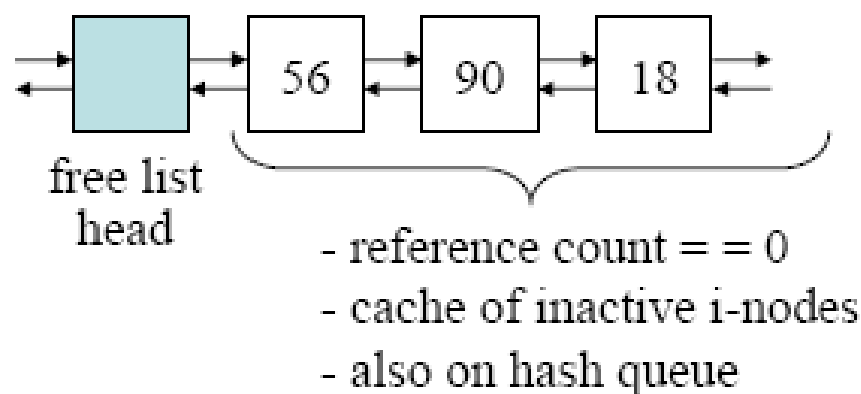
memory



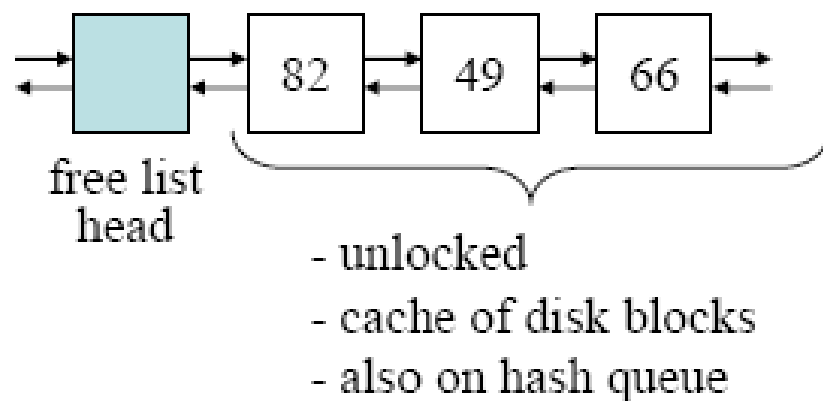
in-core i-nodes

# In-core I-nodes and Buffer Header

[in-core i-node]



[buffer header]



algorithm **iget**

input: inode number

output: locked inode

```
{  
    while (not done) {  
        if (inode in inode cache) {  
            if (inode locked) {  
                sleep(event inode becomes  
                           unlocked);  
                continue;  
            }  
            /* special processing for mount  
               points */  
            if (inode on inode free list)  
                remove from free list;  
            increment inode reference count;  
            return (inode);  
        }  
    }
```

```
/* inode not in inode cache */
```

```
if (no inodes 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)
```

```
    initialize inode;
```

```
    return (inode);
```

```
    }  
}
```

algorithm for allocation of  
in-core i-nodes

# Accessing I-nodes (1)

- *iget* is almost identical to *getblk*.
- The kernel maps the device number and i-node number into a hash queue and searches the queue for the i-node.
- If it cannot find the i-node, it allocates one from the free list and locks it.
- The kernel then prepares to read the disk copy into the in-core copy.
  - $\text{block num} = ((\text{i-node number} - 1) / \text{number of i-nodes per block}) + \text{start block of i-node list}$
  - $\text{offset} = ((\text{i-node number} - 1) \bmod (\text{number of i-nodes per block})) * \text{size of disk i-node}$

## Accessing I-nodes (2)

- The kernel manipulates the i-node **lock** and **reference count** independently.
  - The lock is set during execution of a system call to prevent other processes from accessing the i-node
  - An i-node is never locked across system calls.
  - The kernel increments the reference count for every active reference to a file.
  - The reference count remains set across system calls.
  - The lock is free between system calls to allow processes to share simultaneous access; the reference count remains set between system calls to prevent the kernel from reallocating an active in-core i-node.

## Accessing I-nodes (3)

- If the kernel attempts to take an i-node from the free list but finds the free list empty, it reports **error**.
  - This is different from the philosophy the kernel follows for disk buffers, where a process sleeps until a buffer becomes free.
  - Process have control over the allocation of i-nodes at user level via execution of *open* and *close*; **the kernel cannot guarantee when an i-node will become available**.
  - Therefore, a process that goes to sleep waiting for a free i-node to become available may never wake up.
  - Processes do not have such control over buffers; a process cannot keep a buffer locked across system calls; **the kernel guarantee that a buffer will become free soon**.

```

algorithm input /* release (put) access to in-core inode */
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 (free);
            set file type to 0;
            free inode (ifree);
        }
        if (file accessed or inode changed or file changed)
            update disk inode;
        put inode on free list;
    }
    release inode lock;
}

```

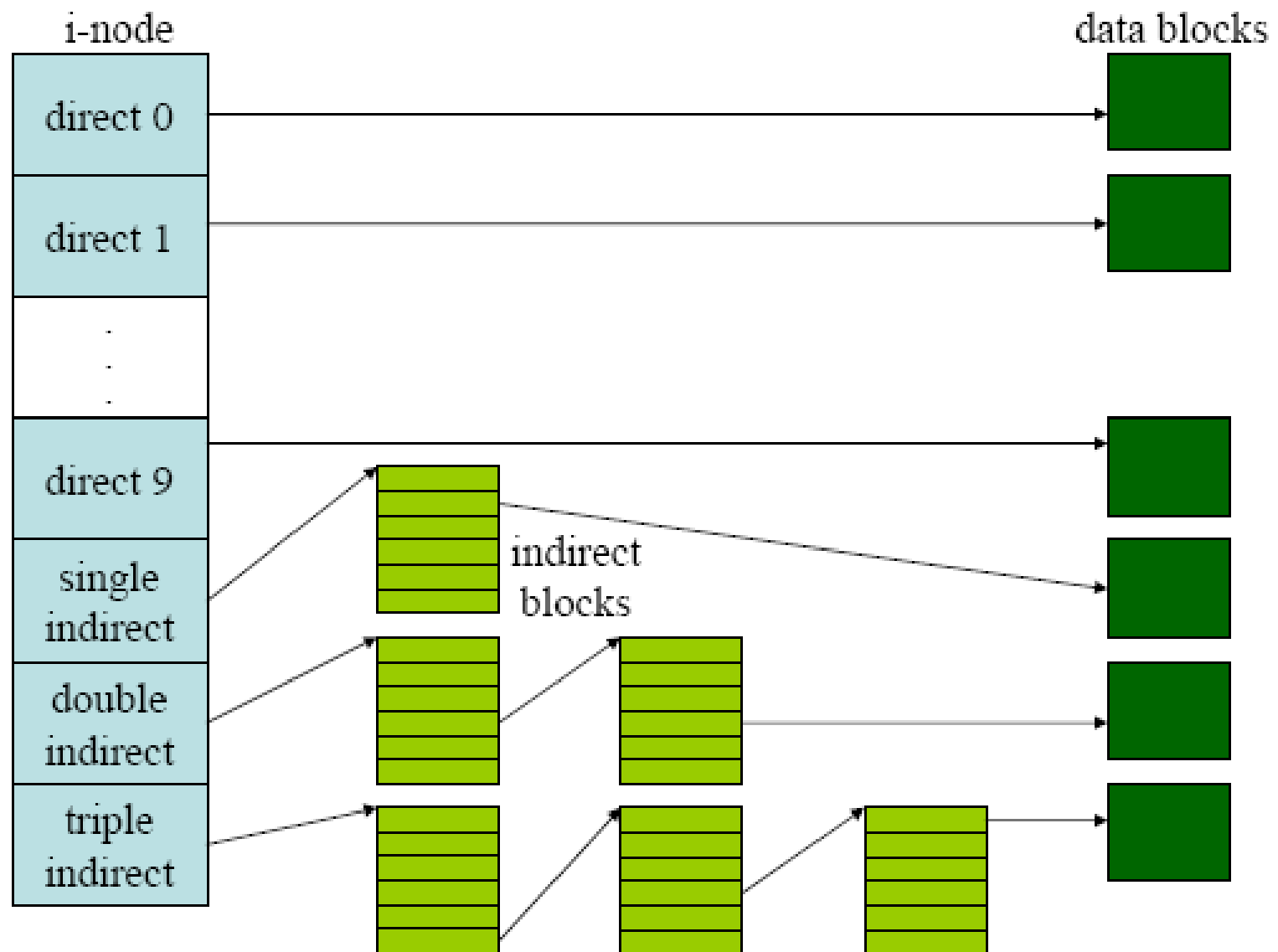
Releasing an I-node



# Structure of a Regular File

- UNIX has 13 entries in the i-node table of contents:
  - 10 “direct” entries
  - 1 “single indirect” entry
  - 1 “double indirect” entry
  - 1 “triple indirect” entry
- Assume that logical block holds 1K bytes and that a block number is addressable by a 32 bit integer
  - 10 direct blocks with 1K bytes each: 10KB
  - 1 indirect block with 256 direct blocks: 256KB
  - 1 double indirect block with 256 indirect blocks: 64MB
  - 1 triple indirect block with 256 double indirect blocks: 16GB

# Direct and Indirect Blocks in I-node



# Conversion from Logical File Block into Disk Block

- Processes access data in a file by byte offset; they work in terms of byte counts and view a file as stream of bytes starting at byte address 0 and going up to the size of the file.
- The kernel converts the user view of bytes into a view of blocks; the file starts at logical block 0 and continues to a logical block number corresponding to the file size.
- The kernel accesses the i-node and converts the logical file block into the appropriate disk block by algorithm *bmap*.

algorithm **bmap**

input: (1) inode

(2) byte offset

output: (1) block number in file system

(2) byte offset into block

(3) bytes of I/O in block

(4) read ahead block number

{

calculate logical block number

in file from byte offset;

calculate start byte in block for I/O;

/\* output 2 \*/

calculate number of bytes to copy to user;

/\* output 3 \*/

check if read ahead applicable, mark inode;

/\* output 4 \*/

determine level of indirection;

Conversion of Byte Offset to  
Block Number in File System

while (not at necessary level  
of indirection) {

calculate index into inode  
or indirect block from  
logical block number  
in file;

get disk block number from  
inode or indirect block;

release buffer from previous  
disk read, if any;

(**brelse**)

if (no more levels of  
indirection)

return (block number);

read indirect disk block;

(**bread**)

adjust logical block number  
in file according to level  
of indirection;

}

}

4096
228
45423
0
0
11111
0
101
367
0
428
9156
824

disk block size = 1024 bytes

1) access to byte offset 9000 => direct block 8

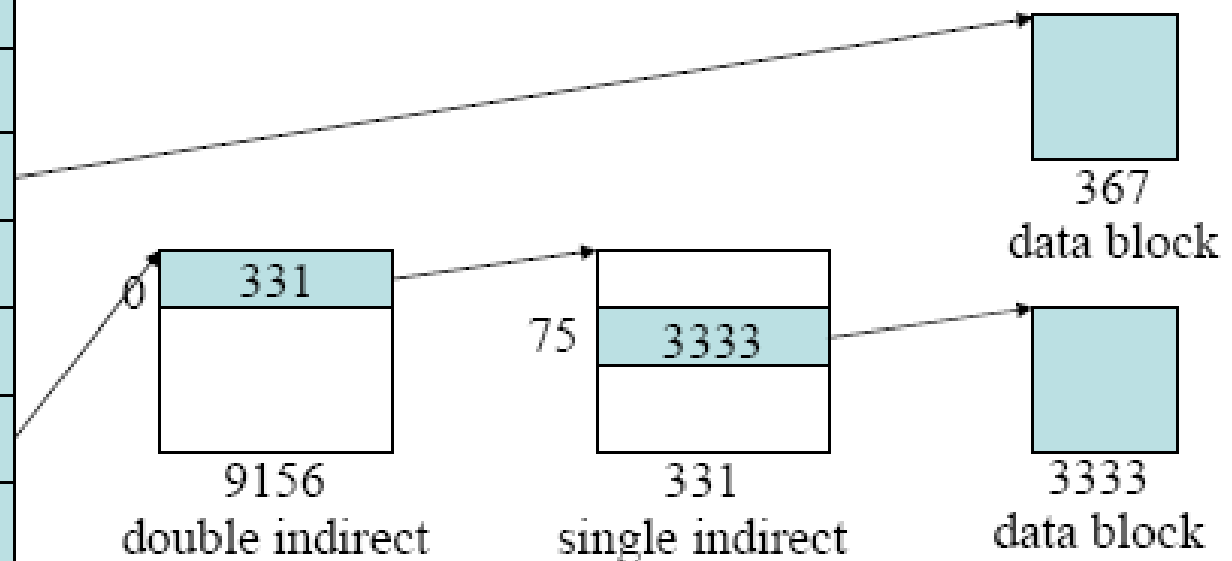
=> block number 367 => 808<sup>th</sup> byte in that block

2) access to byte offset 350,000 => the double indirect block

=> the first byte via the double indirect block = 272,384  
(256K + 10K)

=> byte number 77,616 of the double indirect block is  
byte number 350,000 => 75<sup>th</sup> block

=> block number 3333 => byte number 816 in block 3333




# Examining I-node More Closely

- Several block entries in the i-node are 0, meaning that the logical block entries contain no data.
  - This happens if no process ever wrote data to those blocks.
  - No disk space is wasted for such blocks.
- Conversion of a large byte offset is an arduous procedure that could require the kernel to access multiple disk blocks in addition to the i-node and data block.
  - Even if the kernel finds the blocks in the buffer cache, the operation is still expensive, because the kernel must make multiple requests of the buffer cache and may have to sleep awaiting locked buffer.
- BSD file system has several extensions.

# Directories (1/2)

- Directories play an important role in conversion of a file name to an i-node number.
- A directory is a file whose data is a sequence of entries, each consisting of an **i-node number** and the **name of a file**.

example layout of /etc

byte offset	i-node number	file name	- each entry is 16 byte long - 2 bytes for i-node number - 14 bytes for file name	
0	83	.		
16	2	..		
32	1798	init		
48	1276	fsck		
64	85	clri		
...				
224	0	crash	 empty entry (i-node number is 0)	
240	95	mkfs		
256	188	inittab		

## Directories (2/2)

- The kernel stores data for a directory just as it stores data for an ordinary file; processes may read directories in the same way they read regular files.
- The access permissions of a directory have the following meaning:
  - **read permission** allows a process to read a directory.
  - **write permission** allows a process to create new directory entries or remove old ones (via *creat* (*open*), *mknod*, *link*, and *unlink* system calls).
  - **execute permission** allows a process to search the directory for a file name.



# Conversion of a Path Name to an I-node

- The initial access to a file is by its path name; the kernel converts the path names to i-nodes.
- The algorithm *namei* parses the path name one component at a time, converting each component into an i-node based on its name and the directory being searched, and eventually returns the i-node of the input path name.
- Every process is associated with a current directory; the *u area* contains a pointer to the current directory i-node.
  - The current directory of each process starts out as the current directory of its parent process at the time it was created.

algorithm **namei**

input: path name

output: locked inode

```
{
  if (path name starts from root)
    working inode = root inode; (iget)
  else
    working inode = current dir inode;
                                (iget)
  while (there is more path name) {
    read next path name component;
    verify that working inode is of dir,
      access permissions OK;
    if (working inode is of root and
        component is "..")
      continue;
    read directory by repeated use of
      algorithms bmap, bread, and brelse;
```

```
    if (component matches an
        entry in directory
        (working inode)) {
      get inode number for matched
        component;
      release working inode;
                                (iput);
      working inode = inode of
        matched component;
                                (iget)
    } else /* component not in dir */
      return (no inode);
  } /* while */
  return (working inode);
}
```

Conversion of a Path Name  
to an I-node

# Algorithm *namei*

- The kernel does a linear search of the directory file associated with the working i-node.
  - Starting at byte offset 0, the kernel converts the byte offset in the directory to the appropriate disk block according to *bmap* and reads the block using *bread*.
  - If the kernel finds the path name component in the directory block, it records the i-node number of the matched directory entry, release the block (*brelse*) and the old working i-node (*iput*), and allocates the i-node of the matched component (*iget*).
  - If the kernel does not match the path name in the block, it converts the new offset to a disk block number (*bmap*), and reads the next block.

# Example (*namei*)

- Suppose a process wants to open “/etc/passwd”
  - Making root its current working i-node, the kernel gathers in the string “etc”.
  - After checking that the process has the necessary permissions to search “/”, the kernel searches root for “etc”.
  - On finding the entry, the kernel releases the i-node for root (*iput*) and allocates the i-node for “etc” (*iget*).
  - After checking the permissions, the kernel searches “etc” for “passwd”.
  - On finding it, the kernel releases the i-node for “etc”, allocates the i-node for “passwd”, and – since the path name is exhausted – returns that i-node.

# Super Block

- To understand how to bind an i-node to a file and how to allocate disk blocks to files, let us examine the structure of the super block.
- The super block consists of the following fields:
  - the size of the file system
  - the number of free blocks in the file system
  - a list of free blocks available on the file system
  - the index of the next free block in the free block list
  - the size of the i-node list
  - the number of free i-nodes in the file system
  - a list of free i-nodes in the file system
  - the index of the next free i-node in the free i-node list
  - lock fields for the free block and free i-node lists
  - a flag indicating that the super block has been modified

# I-node Assignment to a New File

- *ialloc* assigns a disk i-node to a newly created file.
  - cf. *iget* allocates a known i-node, one whose i-node number was previously determined; *namei* determines the i-node number by matching a path name component to a name in a directory.
- When a process needs a new i-node, the kernel could theoretically search the i-node list for a free i-node; however, such a search would be expensive, requiring at least one read operation for every i-node.
- To improve performance, the file system super block contains an array to cache the numbers of free i-nodes in the file system.

```

algorithm ialloc /* allocate inode */
input: file system
output: locked inode
{
    while (not done) {
        if (super block locked) {
            sleep(event super block becomes
                free);
            continue;
        }
        if (super block inode list is empty) {
            lock super block;
            get remembered inode for free
                inode search;
            search disk for free inode until
                super block full; (bread, brelse)
            unlock super block;
            wakeup(event super block becomes
                free);
            if (no free inodes found on disk)
                return (no inode);
        }
    }
}

```

```

        set remembered inode for
            next free inode search;
    } /* if (list is empty) */
    /* there are inodes in super
        block inode list */
    get inode number from super
        block inode list;
    get inode; (iget)
    if (inode not free after all) {
        write inode to disk;
        release inode; (iput)
        continue;
    }
    initialize inode;
    write inode to disk;
    decrement file system free inode
        count;
    return (inode);
} /* while */
}

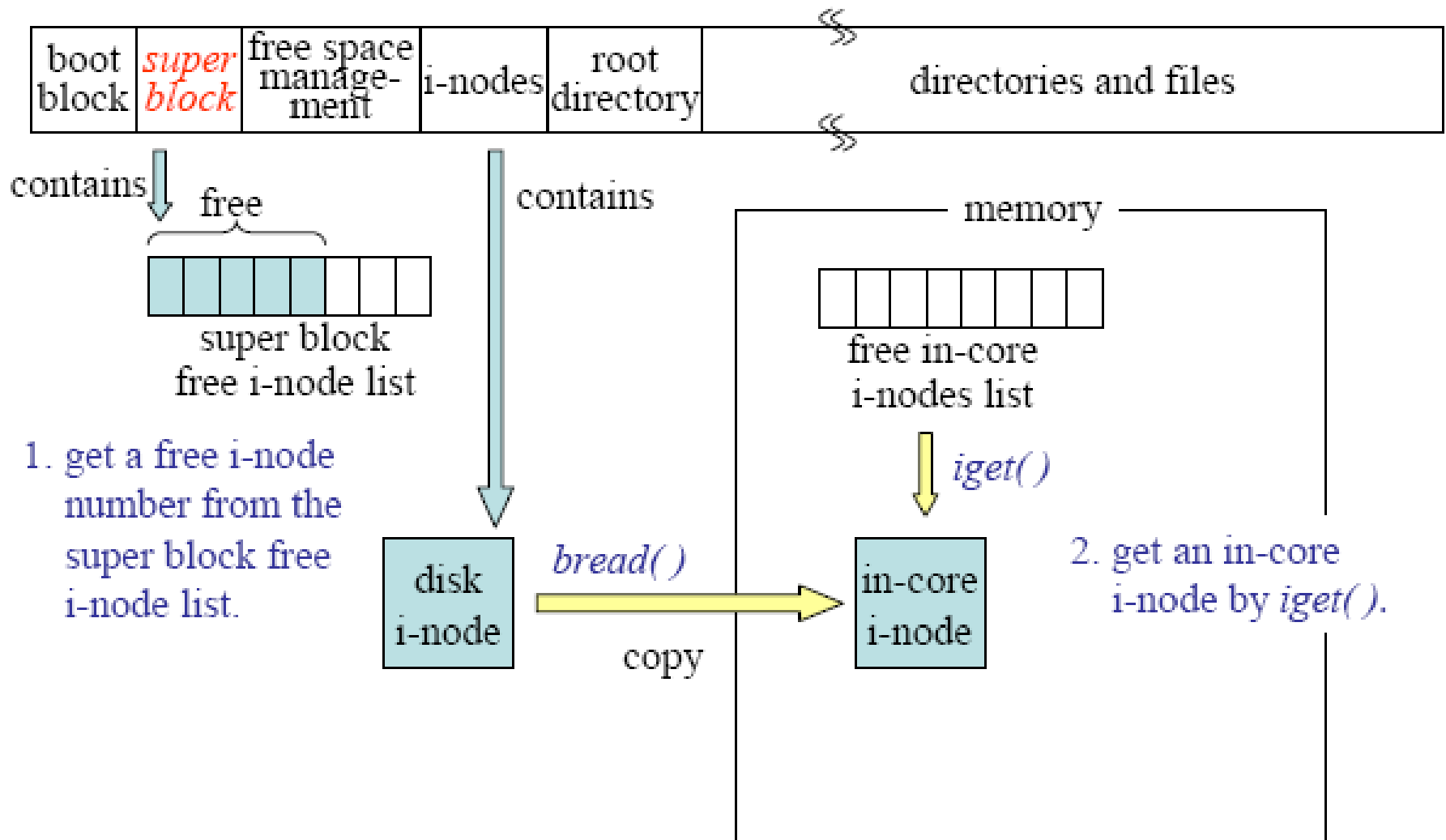
```

# I-node Assignment to a New File

- The kernel first verifies that no other processes have locked the super block free i-node list.
- If the super block free i-node list is not empty, the kernel assigns the next i-node number, allocates a free in-core i-node for the newly assigned disk i-node by *iget*, copies the disk i-node to the in-core copy, initializes the fields, and returns the locked i-node.
- It updates the disk i-node to indicate that the disk i-node is now in use.
- A race condition arises when several processes alter common data structures such that the resulting computations depend on the order in which the processes executed, even though all processes obeyed the locking protocol.



# I-node Assignment to a New File

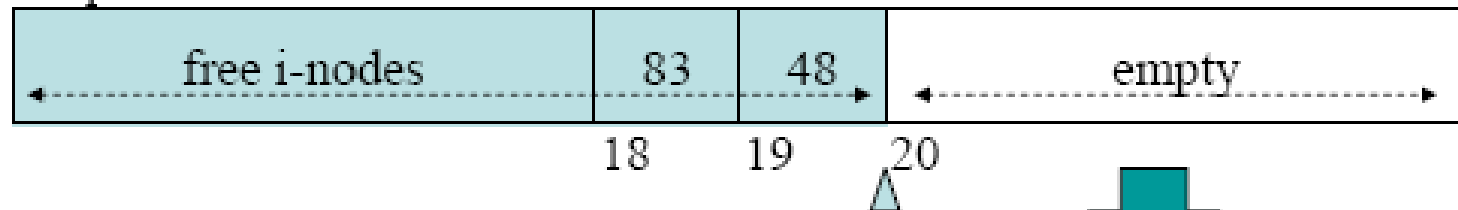


# I-node Assignment to a New File

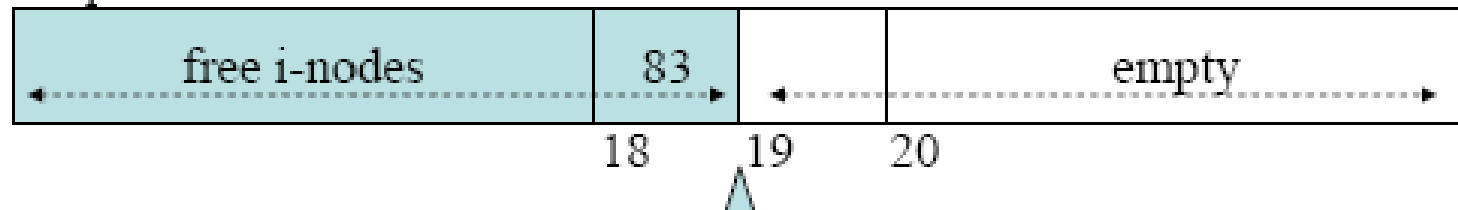
- If the super block free i-node list is empty, the kernel searches the disk and places as many free i-node numbers as possible into the super block.
- The kernel reads the i-node list on disk, block by block, and fills the super block list of i-node numbers to capacity, remembering the highest-numbered i-node that it finds. (“remembered” i-node)
- The next time the kernel searches the disk for free i-nodes, it uses the remembered i-node as its starting point.
- After gathering a fresh set of free i-node numbers, it starts the i-node assignment algorithm from the beginning.

# I-node Assignment to a New File

Super block free i-node list



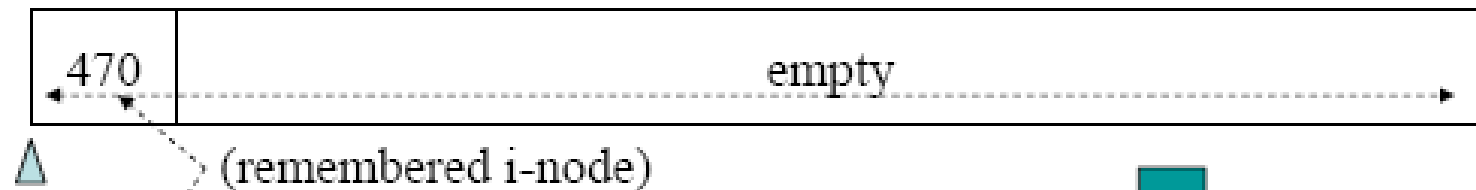
Super block free i-node list



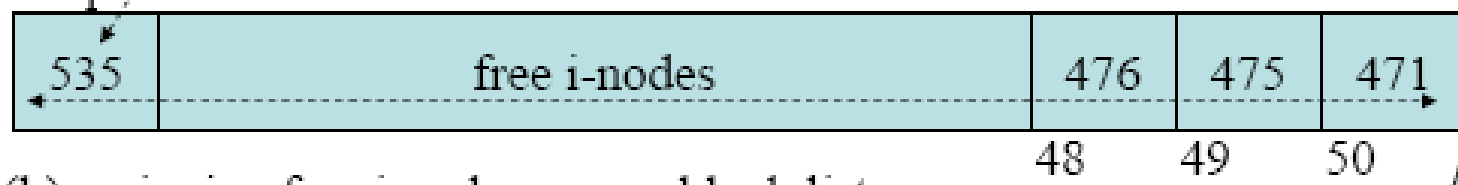
(a) assigning free i-node from middle of list

△: index

Super block free i-node list



Super block free i-node list



(b) assigning free i-node – super block list empty

# I-node Assignment to a New File

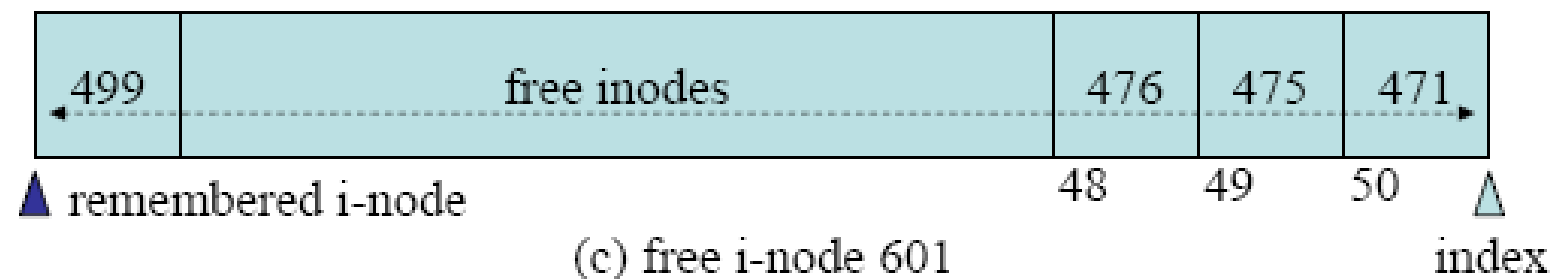
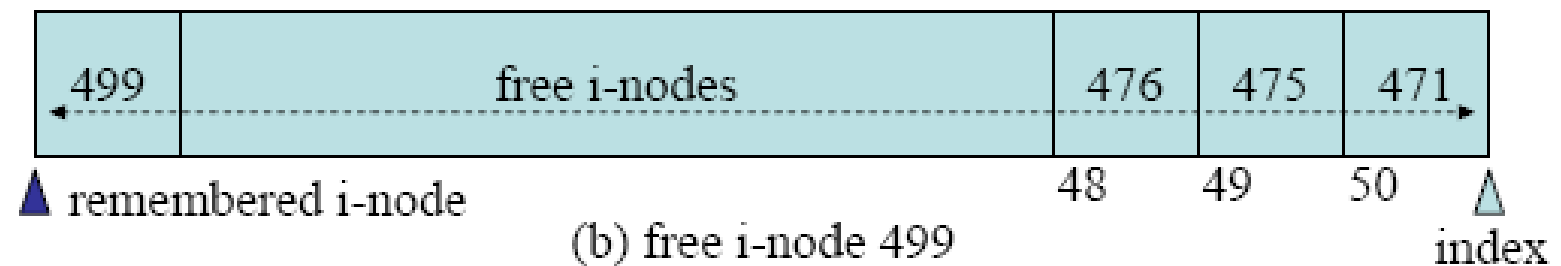
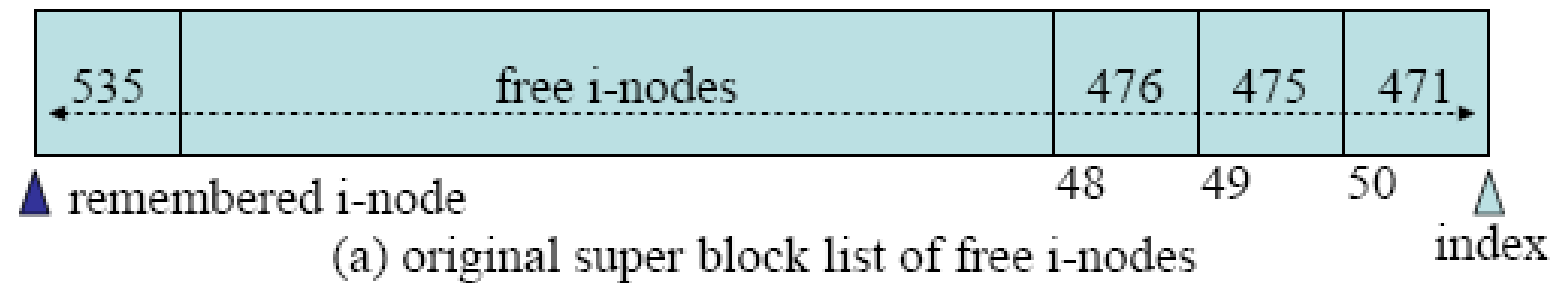
- When the kernel assigns an i-node, it decrements the index for the next valid i-node number to 18 and takes i-node number 48.
- If the list of free i-nodes in the super block has only one entry, it will notice that the array is empty and search the disk for free i-nodes.
  - starting from 470 = **the remembered i-node**
- When the kernel fills the super block free list to capacity, it remembers the last i-node as the start point for next search of the disk.

# Freeing I-node

- After incrementing the total number of available i-nodes in the file system, the kernel checks the lock on the super block.
  - If locked, it avoids race conditions by returning immediately: the i-node number is not put into the super block, but it can be found on disk.
- If the list is not locked, the kernel checks if it has room for more i-node numbers.
  - If the list is full, the kernel may not save the newly freed i-node there: it compares the number of the freed i-node with that of the remembered i-node.
  - If the freed i-node number is less than the remembered i-node number, it “remembers” the newly freed i-node number.

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

# Placing Free I-node Numbers into the Super Block



## Placing Free I-node Numbers into the Super Block

- If the super block list of free i-nodes has room for more free i-node numbers, the kernel places the i-node number on the list, increments the index.
- If the list is full, the kernel compares the i-node number it has freed to the remembered i-node number.
  - i.e., “535” and “499”
- The kernel makes 499 the remembered i-node and evicts number 535 from the free list.
- If the kernel then frees i-node number 601, it does not change the contents of the free list.
  - the kernel will search the disk for free i-nodes starting from 499, and find i-nodes 535 and 601 again.



# Race Condition in Assigning I-nodes

- Consider three processes, A, B, and C, and suppose that the kernel, acting on behalf of process A, assigns i-node *I* but goes to sleep before it copies the disk i-node into the in-core copy.
  - *iget* (in *ialloc*) and *bread* (in *iget*) give opportunity to go to sleep.
- While process A is asleep, suppose process B attempts to assign a new i-node but discovers that the super block list of free i-nodes is empty.
- It is possible for process B to find i-node *I* free on the disk since process A is still asleep, and the kernel does not know that the i-node is about to be assigned.

## Race Condition in Assigning I-nodes (2)

- When process A wakes up, it completes the assignment of i-node *I*.
- Now suppose process C later requests an i-node and happens to pick i-node *I* from the super block free list. When it gets the in-core copy of the i-node, it will find its file type set, implying that the i-node was already assigned.
- The kernel checks for this condition and, finding that the i-node has been assigned, tries to assign a new one.
- Writing the updated i-node to disk immediately after its assignment in *ialloc* makes the chance of the race smaller, because the file type fields will mark the i-node in use.

Process A

Process B

process C

assign i-node *I*  
from super block

sleeps while  
reading i-node (a)

i-node *I* in-core  
does usual activity

tries to assign i-node  
from super block

super block empty (b)

search for free i-node  
on disk, put i-node *I*  
in super block (c)

completes search,  
assigns another i-node (d)

assign i-node *I*  
from super block

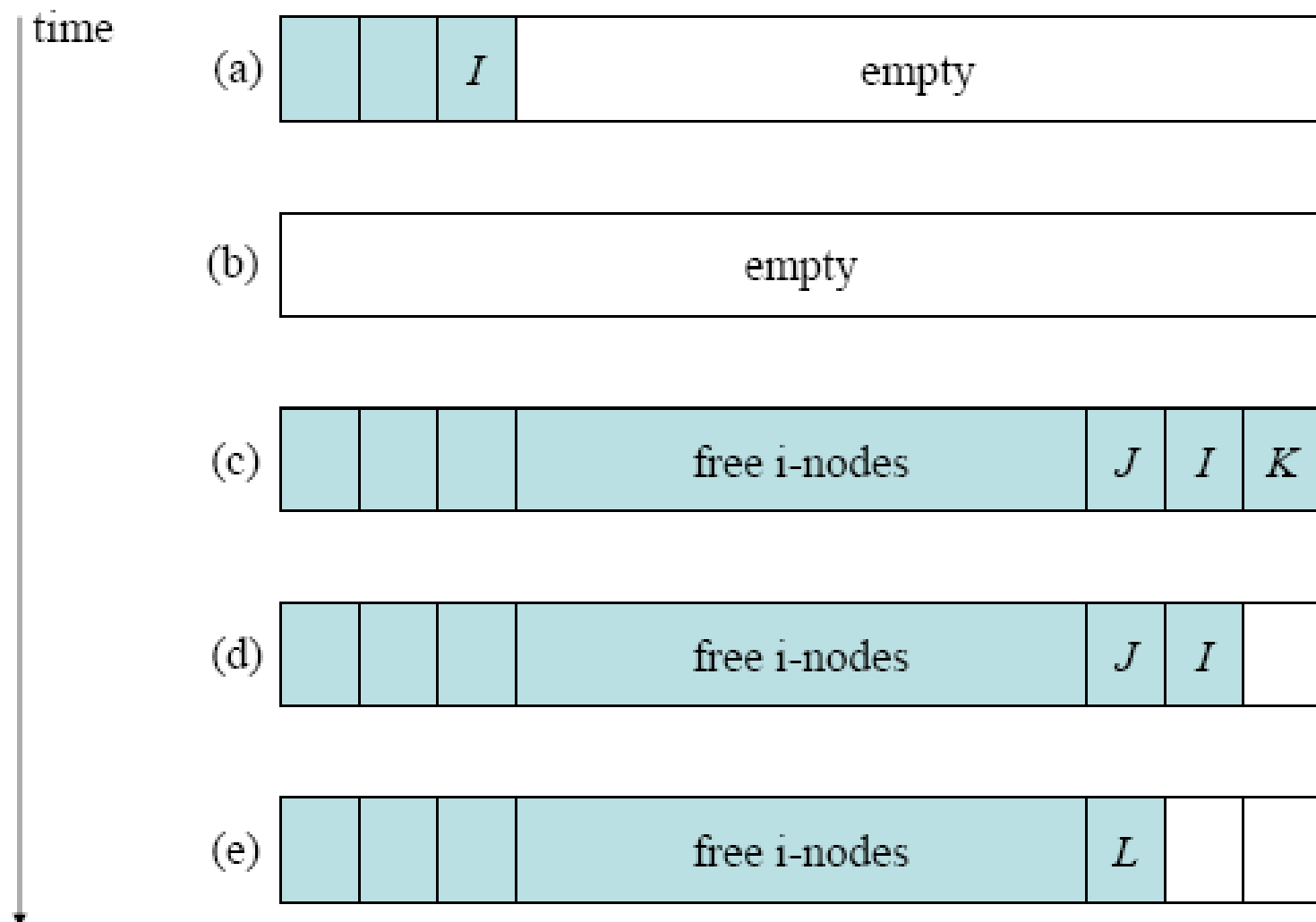
*I* is in use!

assign another i-node (e)

time



# Race Condition in Assigning I-nodes (3)



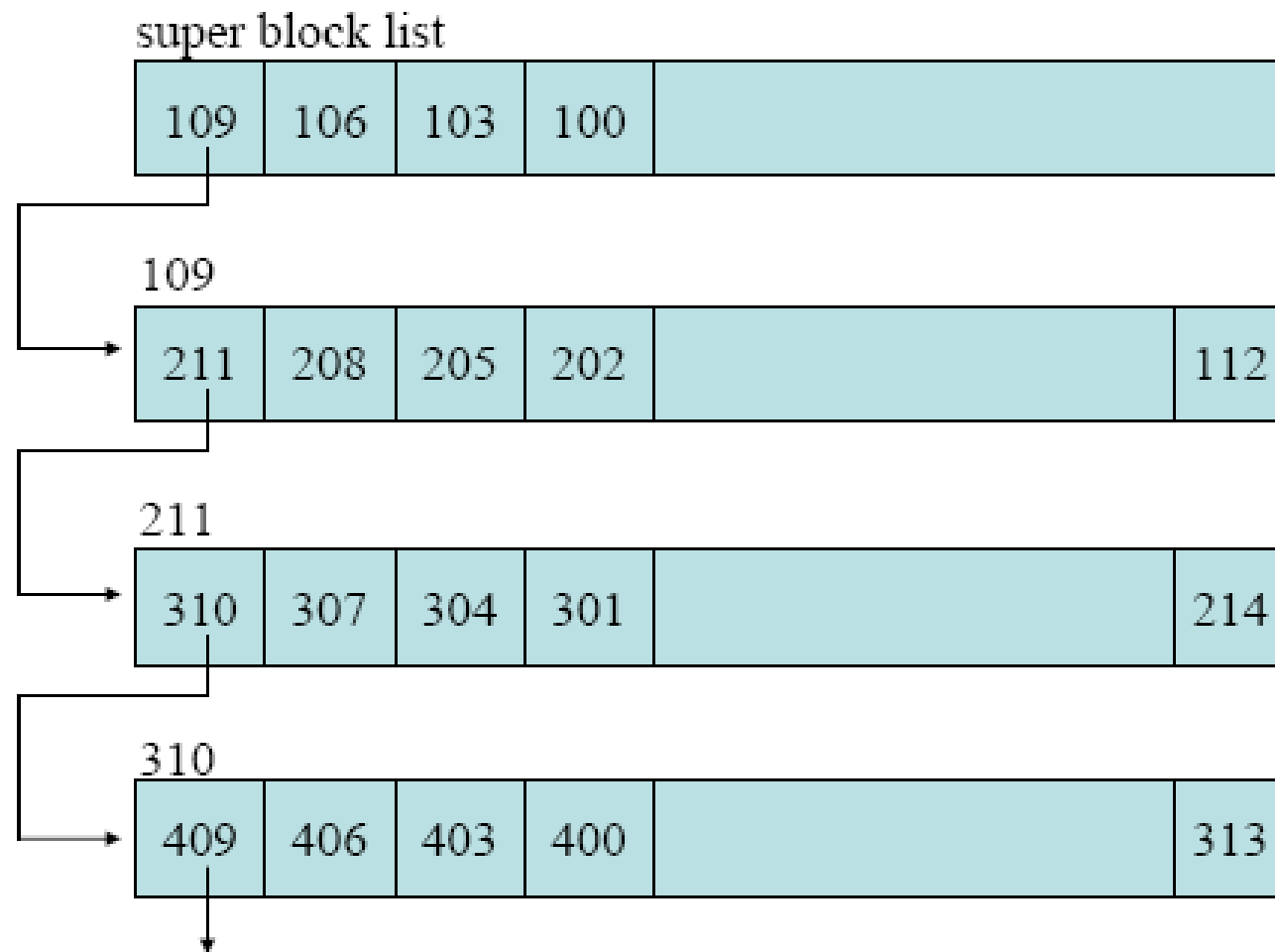
# Race Condition in Assigning I-nodes

- Locking the super block while reading in a new set from disk prevents other race conditions.
- If the super block list were not locked, a process could find it empty and try to populate it from disk, occasionally sleeping while waiting for I/O completion.
- Suppose a second process also tried to assign a new i-node and found the list empty: it, too, would try to populate the list from disk.
- At worst, race condition shown above would be more frequent.
- Similarly, if a process freeing an i-node did not check that the list is locked, it could overwrite i-node numbers already in the free list while another process was populating it from disk.

# Allocation of Disk Blocks

- When a process writes data to a file, the kernel must allocate disk blocks from the file system.
- The super block contains an array that is used to cache the numbers of free disk blocks in the file system.
- The utility program *mkfs* (make file system) organizes the data blocks of a file system in a linked list, such that each link of the list is a disk block that contains an array of free disk block numbers, and one array entry is the number of the next block of the linked list.

# Linked List of Free Disk Block Numbers



# Allocation of Disk Blocks

- When the kernel wants to allocate a block from a file system by *alloc*, it allocates the next available block in the super block list.
- If the allocated block is the last available block in the super block cache, the kernel treats it as a pointer to a block that contains a list of free blocks.
  - it reads the block, populates the super block array with the new list of block numbers, and then proceeds to use the original block number.
  - it allocates a buffer for the block and clears the buffer's data.
- If the file system contains no free blocks, the calling process receives an error.



# Original Linked List

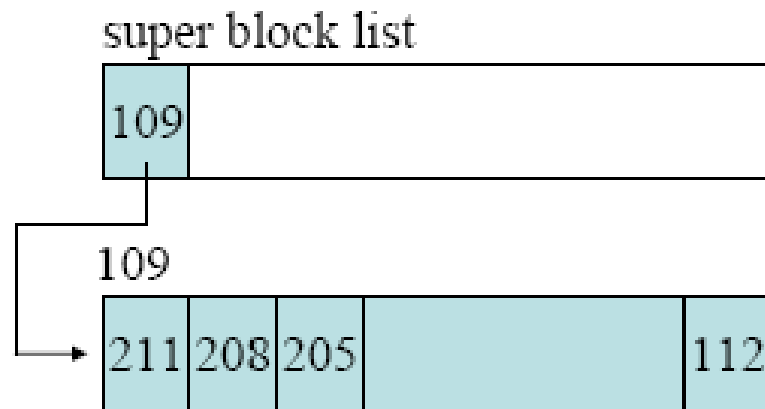
- If a process *writes* a lot of data to a file, it repeatedly asks for the blocks, but the kernel assigns only one block at a time.
- The program *mkfs* tries to organize the original linked list of free block numbers so that block numbers dispensed to a file are near each other.
  - this helps performance, because it reduces disk seek time and latency when a process reads a file sequentially.
- The order of block numbers on the free block linked lists breaks down with heavy use as processes *write* files and remove them.

```
algorithm alloc /* file system block allocation */
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 numbers 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;
}
```

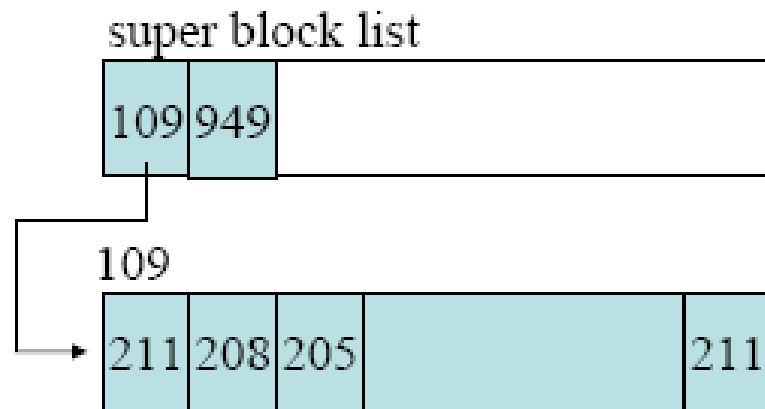
# Freeing a Block

- The algorithm *free* for freeing a block is the reverse of the one for allocating a block.
- If the super block list is not full, the block number of the newly freed block is placed on the super block list.
- If the super block list is full, the newly freed block becomes a link block.
  - the kernel writes the super block list into the block and write the block to disk.
  - it then places the block number of the newly freed block in the super block list: that block number is the only member of the list.

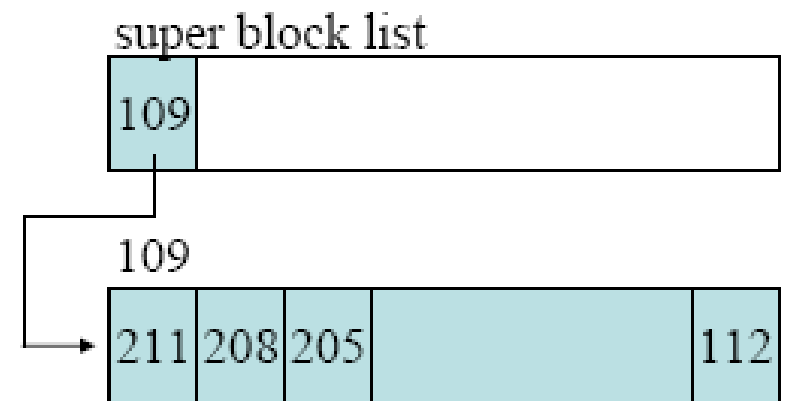
# Requesting and Freeing Disk Blocks



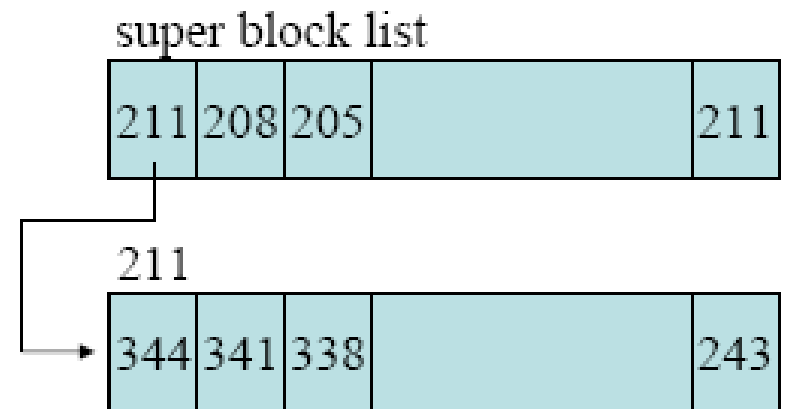
(a) original configuration



(b) after freeing block number 949



(c) after assigning block number (949)



(d) after assigning block number (109)  
replenish super block free list

# Other File Types

- Pipes
  - a pipe differs from a regular file in that its data is transient: once data is read from a pipe, it cannot be read again.
  - the data is read in the order that it was written to the pipe.
  - the kernel stores data in a pipe the same way it stores data in an ordinary file, except that it uses only the direct blocks.
- Special files (block/character device special files)
  - both types specify devices, and therefore the file i-nodes do not reference any data.
  - instead, the i-node contains two numbers known as the major and minor device numbers.
  - the major number indicates a device type such as disk and terminal, and the minor number indicates the unit number.