19 CS2106S: Lecture Notes Internal Representation of file systems

19CS2106S

Lecture Notes

## File System: Internal Representation of file systems

Session No:7

|  |  |  |
| --- | --- | --- |
| List of Topics:   * Internal Representation of File System * Lower level File System Algorithms: alloc,free,ialloc,ifree | Learning Outcomes:  Understand the Representation of the File System  Design and implement Lower level file system algorithms | Questions Answered from This session :  How internally file system is represented.  How allocation of in-core inode is done and accessed. |

# 

# References:

# 

# 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 inode list
* The number of free inodes in the file system
* A list of free inodes in the file system
* The 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 been modified.

The kernel periodically writes the super block to disk if it had been modified so that it is consistent with the data in the file system.

# INODE ASSIGNMENT TO A NEW FILE:

The kernel uses algorithm iget to allocate a known inode, one whose (file system and) inode number was previously determined.

In algorithm namei for instance, the kernel determines the inode number by matching a path name component to a name in a directory. Another algorithm, ialloc, assigns a disk inode to a newly created file.

The file system contains a linear list of inodes. An inode is free if its type field is zero. When a process needs a new inode, the kernel could theoretically search the inode list for a free inode.

However, such a search would be expensive, requiring at least one read operation (possibly from disk) for every inode.

To improve performance, the file system super block contains an array to cache the numbers of free inodes in the file system.

Figure 3.10 shows the algorithm ialloc for assigning new inodes. The kernel first verifies that no other processes have locked access to the super block free inode list.

### algorithm ialloc /\* allocate inode \*/ input: file system

***output: locked inode***

***{***

***while (not done)***

***{***

***if (super block locked)***

***{***

***sleep (event super block becomes free); continue; /\* while loop \*/***

***}***

***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 (algorithms bread and brelse);***

***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;***

***}***

***/\* there are inodes in super block inode list \*/ 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; /\* while loop \*/***

***}***

***/\* inode is free \*/ initialize inode; write inode to disk;***

***decrement file system free inode count; return (inode);***

***}***

***}* Figure 3.10: Algorithm for Assigning New Inodes**

If the list of inode numbers in the super block is not empty, the kernel assigns the next inode number, allocates a free in-core inode for the newly assigned disk inode using algorithm iget (reading the inode from disk if necessary), copies the disk inode to the in-core copy, initializes the fields in the inode, and returns the locked inode.

It updates the disk inode to indicate that the inode is now in use: A non-zero file type field indicates that the disk inode is assigned.

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.

The kernel reads the inode list on disk, block by block, and fills the super block list of inode numbers to capacity, remembering the highest-numbered inode that it finds. Call that inode the "remembered" inode; it is the last one saved in the super block.

The next time the kernel searches the disk for free inodes, it uses the remembered inode as its starting point, thereby assuring that it wastes no time reading disk blocks where no free inodes should exist.

After gathering a fresh set of free inode numbers, it starts the inode assignment algorithm from the beginning. Whenever the kernel assigns a disk inode, it decrements the free inode count recorded in the super block.

The algorithm for freeing an inode is much simpler. After incrementing the total number of available inodes in the file system, the kernel checks the lock on the super block.

***If locked***, it avoids race conditions by returning immediately: The inode number is not put into the super block, but it can be found on disk and is available for reassignment.

***If the list is not locked***, the kernel checks if it has room for more inode numbers and, if it does, places the inode number in the list and returns.

If the list is full, the kernel may not save the newly freed inode there: It compares the number of the freed inode with that of the remembered inode.

If the freed inode number is less than the remembered inode number, it "remembers" the newly freed inode number, discarding the old remembered inode number from the super block.

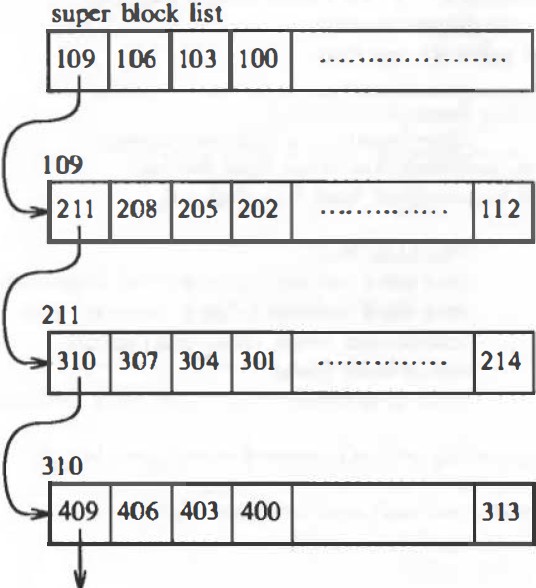
# ALLOCATION OF DISK BLOCKS

When a process writes data to a file, the kernel must allocate disk blocks from the file system for direct data blocks and, sometimes, for indirect blocks.

The file system 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.

Figure 3.11 shows an example of the linked list, where the first block is the super block free list and later blocks on the linked list contain freer block numbers.

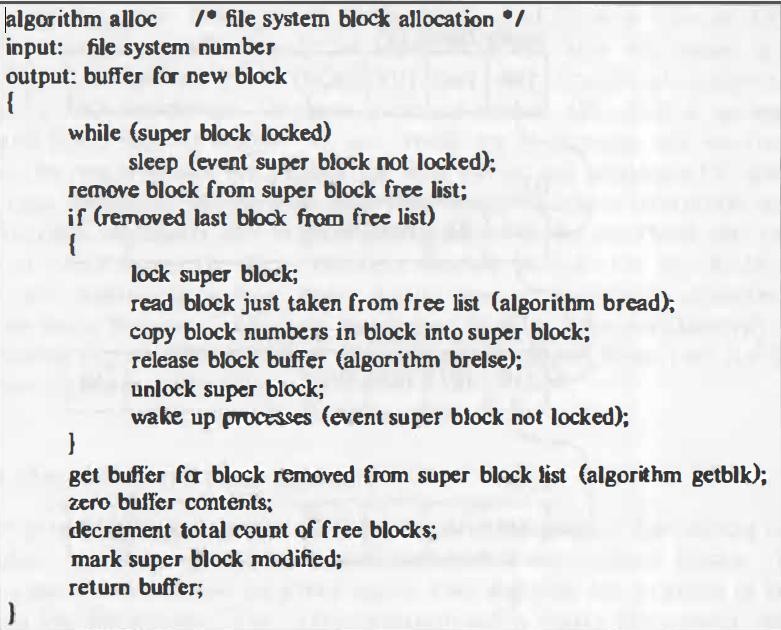


**Figure 3.11: Linked List of Free Disk Block Numbers**

When the kernel wants to allocate a block from a file system (algorithm alloc, Figure 3.12), it allocates the next available block in the super block list. Once allocated, the block cannot be reallocated until it becomes free.

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. The disk block has now been assigned, and the kernel has a buffer to work with. If the file system contains no free blocks, the calling process receives an error.

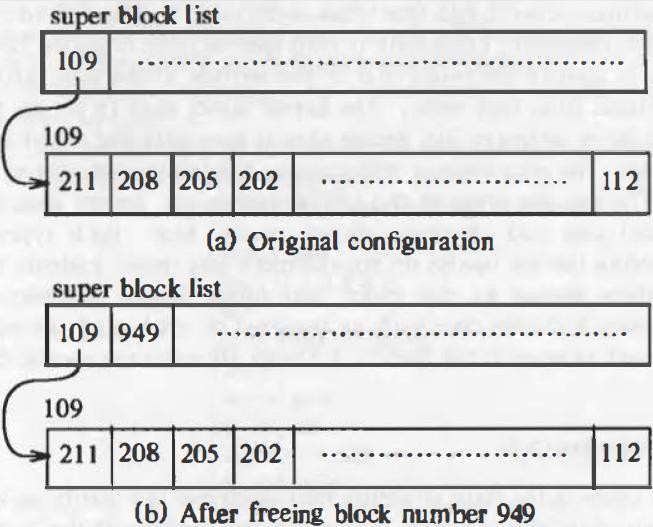


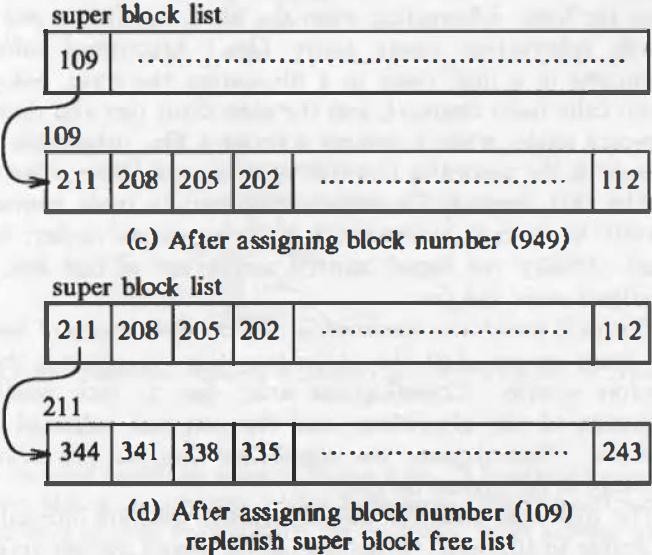
**Figure 3.12: Algorithm for Allocating Disk Block**

If a process writes a lot of data to a file, it repeatedly asks the system for blocks to store the data, 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 tile sequentially.

Figure 3.13 shows a sequence of alloc and free operations, starting with one entry on the super block free list. The kernel frees block 949 and places the block number on the free list.



**Figure: 3.13 Requesting and Freeing Disk Blocks**

It then allocates a block and removes block number 949 from the free list. Finally, it allocates a block and removes block number 109 from the free list.

Because the super block free list is now empty, the kernel replenishes the list by copying in the contents of block 109, the next link on the linked list.

Figure 3.13(d) shows the full super block list and the next link block, block 211.

# OTHER FILE TYPES:

The UNIX system supports two other file types: pipes and special files. A pipe, sometimes called a FIFO (for "first-in-first-out"), differs from a regular file in that its data is transient: Once data is read from a pipe, it cannot be read again.

Also, the data is read in the order that it was written to the pipe, and the system allows no deviation from that order.

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, not the indirect blocks.

The last file types in the UNIX system are special files, including block device special files and character device special files.

Both types specify devices, and therefore the file inodes do not reference any data.

Instead, the inode contains two numbers known as the major and minor device numbers.

The major number indicates a device type such as terminal or disk, and the minor number indicates the unit number of the device.