

INTERNAL REPRESENTATION OF FILES:

Every file on a UNIX system has a unique inode. The inode contains the information necessary for a process to access a file, such as file ownership, access rights, file size, and location of the file's data in the file system.

Processes access files by a well defined set of system calls and specify a file by a character string that is the path name. Each path name uniquely specifies a file, and the kernel converts the path name to the file's inode. File system algorithms are represented in the figure 3.1 given below:

Lower Level File System Algorithms						
namei			alloc	free	ialloc	ifree
iget	iput	bmap				
buffer allocation algorithms						
getblk	brelease	bread	breada	bwrite		

Figure 3.1 File System Algorithms

The algorithm **iget** returns a previously identified inode, possibly reading it from disk via the buffer cache, and the algorithm **iput** releases the inode. The algorithm **bmap** sets kernel parameters for accessing a file.

The algorithm **namei** converts a user-level path name to an inode, using the algorithms **iget**, **iput**, and **bmap**. Algorithms **alloc** and **free** allocate and free disk blocks for files, and algorithms **ialloc** and **ifree** assign and free inodes for files.

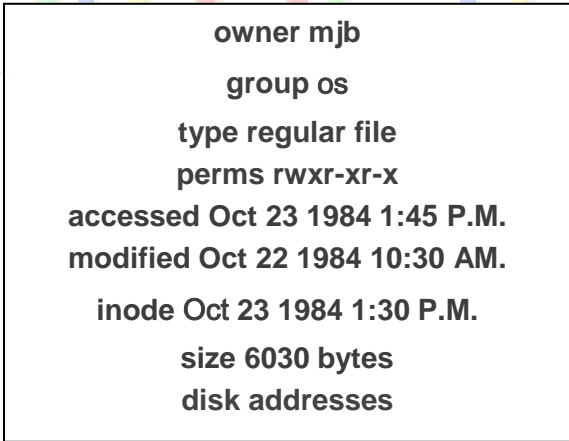
INODES:

Definition: Inodes exist in a static form on disk, and the kernel reads them into an in-core inode to manipulate them. Disk inodes consist of the following fields:

- 1) File owner identifier. Ownership is divided between an individual owner and a "group" owner and defines the set of users who have access rights to a file. The super user has access rights to all files in the system.
- 2) File type. Files may be of type regular, directory, character or block special, or FIFO (pipes).

- 3) File access permissions. The system protects files according to three classes: the owner and the group owner of the file, and other users; each class has access rights to read, write and execute the file, which can be set individually.
- 4) File access times, giving the time the file was last modified, when it was last accessed, and when the inode was last modified.
- 5) Number of links to the file, representing the number of names the file has in the directory hierarchy.
- 6) Table of contents for the disk addresses of data in a file. Although users treat the data in a file as a logical stream of bytes, the kernel saves the data in discontinuous disk blocks. The inode identifies the disk blocks that contain the file's data.
- 7) File size. Data in a file is addressable by the number of bytes from the beginning of the file, starting from byte offset 0, and the file size is 1 greater than the highest byte offset of data in the file.

For example, if a user creates a file and writes only 1 byte of data at byte offset 1000 in the file, the size of the file is 1001 bytes.



```
owner mjb
group os
type regular file
perms rwxr-xr-x
accessed Oct 23 1984 1:45 P.M.
modified Oct 22 1984 10:30 AM.
inode Oct 23 1984 1:30 P.M.
size 6030 bytes
disk addresses
```

Figure 3.2 Sample Disk Inode

Figure 3.2 shows the disk inode of a sample file. This inode is that of regular file owned by "mjb", which contains 6030 bytes. The system permits "mjb" to read, write, or execute the file; members of the group "os" and all other users can only read or execute the file, not write it.

The last time anyone read the file was on October 23, 1984, at 1:45 in the afternoon, and the last time anyone wrote the file was on October 22, 1984, at 10:30 in the morning. The inode was last changed on October 23, 1984, at 1:30 in the afternoon, although the data in the file was not written at that time. The kernel encodes the above information in the inode.

Note: The distinction between writing the contents of an inode to disk and writing the contents of a file to disk.

The contents of a file change only when writing it. The contents of an inode change when changing the contents of a file or when changing its owner, permission, or link settings. Changing the contents of a file automatically implies a change to the inode, but changing the inode does not imply that the contents of the file change.

The in-core copy of the inode contains the following fields in addition to the fields of the disk inode:

- 1) The status of the in-core inode, indicating whether
 - a. the inode is locked,
 - b. a process is waiting for the inode to become unlocked,
 - c. the in-core representation of the inode differs from the disk copy as a result of a change to the data in the inode,
 - d. the in-core representation of the file differs from the disk copy as a result of a change to the file data, the file is a mount point.
- 2) The logical device number of the file system that contains the file.
- 3) The inode number. Since inodes are stored in a linear array on disk, the kernel identifies the number of a disk inode by its position in the array. The disk inode does not need this field.
- 4) Pointers to other in-core inodes.
- 5) A reference count, indicating the number of instances of the file that are active (such as when opened).

Many fields in the in-core inode are analogous to fields in the buffer header, and the management of inodes is similar to the management of buffers. *The most striking difference between an in-core inode and a buffer header is the in-core reference count, which counts the number of active instances of the file.*

An inode is active when a process allocates it, such as when opening a file. An inode is on the free list only if its reference count is 0, meaning that the kernel can reallocate the in-core inode to another disk inode.

On the other hand, a buffer has no reference count; it is on the free list if and only if it is unlocked.

Accessing Inodes:

The kernel identifies particular inodes by their file system and inode number and allocates in-core inodes at the request of higher-level algorithms.

The algorithm *iget* allocates an in-core copy of an inode (Figure 3.3); it is almost identical to the algorithm *getblk* for finding a disk block in the buffer cache.

```

algorithm iget
input: file system inode number
output: locked inode
{
    while (not done)
    {
        if (inode in inode cache)
        {
            if (inode locked)
            {
                sleep (event inode becomes unlocked);
                continue; /* loop back to while */
            }
            /* 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 (algorithm bread);
        initialize inode (e.g. reference count to 1);
        return (inode);
    }
}

```

Figure 3.3: Algorithm for Allocation of In-Core Inodes

The 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.

The kernel then prepares to read the disk copy of the newly accessed inode into the in-core copy. It already knows the inode number and logical device and computes the logical disk block that contains the inode according to how many disk inodes fit into a disk block.

The computation follows the formula:

$$\text{block num} = ((\text{inode number} - 1) / \text{number of inodes per block}) + \text{start block of inode list}$$

When the kernel knows the device and disk block number, it reads the block using the algorithm `bread`, then uses the following formula to compute the byte offset of the inode in the block:

$$((\text{inode number} - 1) \text{ modulo } (\text{number of inodes per block})) * \text{size of disk inode}$$

Releasing Inodes:

When the kernel releases an inode (algorithm `iput`, Figure 3.4), it 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 from the disk copy.

```

algorithm iput          /* 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 (algorithm free, section 4.7);
            set file type to 0;
            free inode (algorithm ifree, section 4.6);
        }
        if (file accessed or inode changed or file changed)
            update disk inode;
        put inode on free list;
    }
    release inode lock;
}

```

Figure 3.4: Releasing an Inode

They differ if the file data has changed, if the file access time has changed, or if the file owner or access permissions have changed.

The kernel places the inode on the free list of inodes, effectively caching the inode in case it is needed again soon.

The kernel may also release all data blocks associated with the file and free the inode if the number of links to the file is 0.

STRUCTURE OF A REGULAR FILE:

The inode contains the table of contents to locate a file's data on disk. Since each block on a disk is addressable by number, the table of contents consists of a set of disk block numbers.

If the data in a file were stored in a contiguous section of the disk (that is, the file occupied a linear sequence of disk blocks), then storing the start block address and the file size in the inode would suffice to access all the data in the file.

However, such an allocation strategy would not allow for simple expansion and contraction of files in the file system without running the risk of fragmenting free storage area on the disk.

Furthermore, the kernel would have to allocate and reserve contiguous space in the file system before allowing operations that would increase the file size. For example, suppose a user creates three files, A, B and C, each consisting of 10 disk blocks of storage, and suppose the system allocated storage for the three files contiguously.

If the user then wishes to add 5 blocks of data to the middle file, B, the kernel would have to copy file B to a place in the file system that had room for 15 blocks of storage. Aside from the expense of such an operation, the disk blocks previously occupied by file B's data would be unusable except for files smaller than 10 blocks (Figure 3.5).

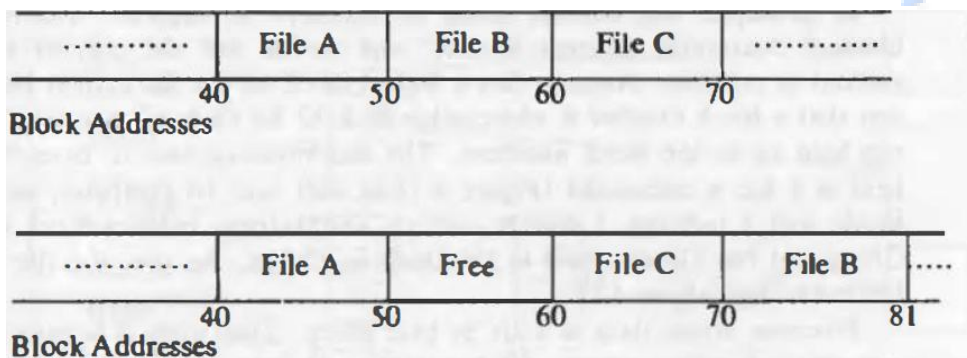


Figure 3.5: Allocation of Contiguous Files and Fragmentation of Free Space

The kernel could minimize fragmentation of storage space by periodically running garbage collection procedures to compact available storage, but that would place an added drain on processing power.

For greater flexibility, the kernel allocates file space one block at a time and allows the data in a file to be spread throughout the file system. But this allocation scheme complicates the task of locating the data.

To keep the inode structure small yet still allow large files, the table of contents of disk blocks conforms to that shown in Figure 3.6. The System V UNIX system runs with 13 entries in the inode table of contents, but the principles are independent of the number of entries. The blocks marked "direct" in the figure contain the numbers of disk blocks that contain real data.

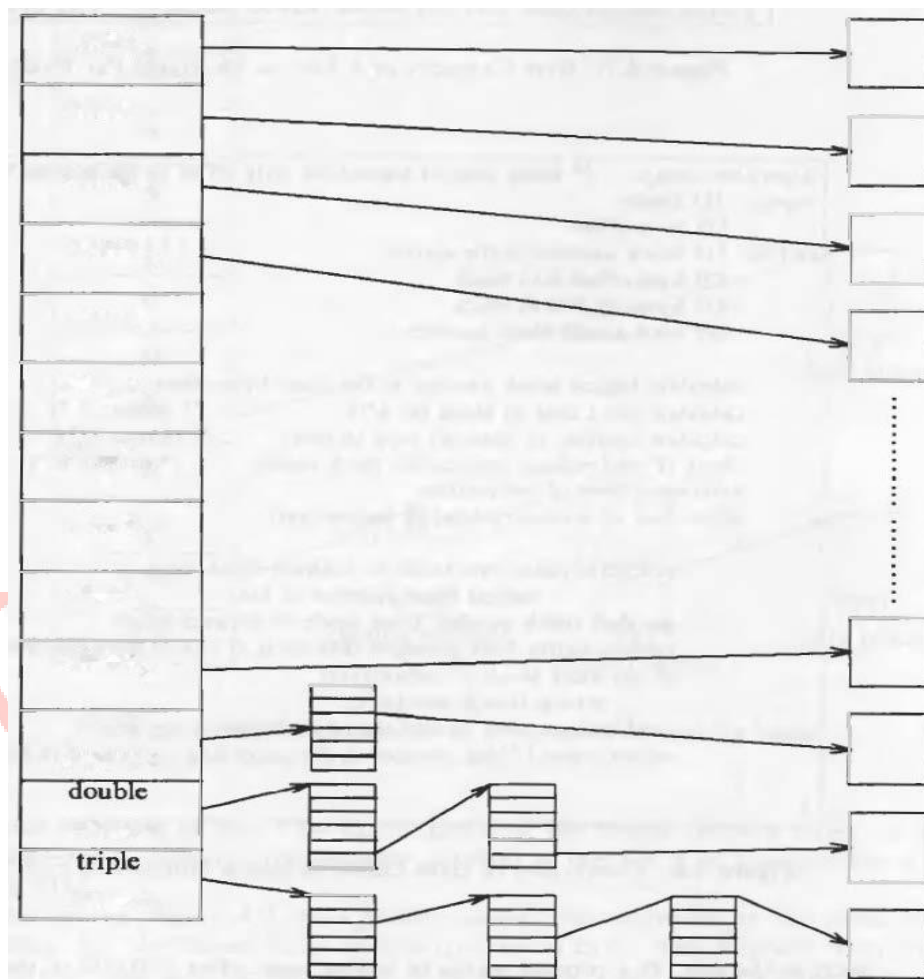


Figure 3.6: Direct and Indirect Blocks in Inode

The block marked "single indirect" refers to a block that contains a list of direct block numbers.

To access the data via the indirect block, the kernel must read the indirect block, find the appropriate direct block entry, and then read the direct block to find the data.

The block marked "double indirect" contains a list of indirect block numbers, and the block marked "triple indirect" contains a list of double indirect block numbers.

The maximum number of bytes that could be held in a file is calculated (Figure 3.7) at well over 16 gigabytes, using 10 direct blocks and 1 indirect, 1 double indirect, and 1 triple indirect block in the inode.

10 direct blocks with 1K bytes each =	10K bytes
1 indirect block with 256 direct blocks =	256K bytes
1 double indirect block with 256 indirect blocks =	64M bytes
1 triple indirect block with 256 double indirect blocks =	16G bytes

Figure 3.7: Byte Capacity of a File - 1 K Bytes per Block

Given that the file size field in the inode is 32 bits, the size of a file is effectively limited to 4 gigabytes.

DIRECTORIES:

Directories are the files that give the file system its hierarchical structure; they play an important role in conversion of a file name to an inode number.

A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file contained in the directory.

A path name is a null terminated character string divided into separate components by the slash ("/") character. Each component except the last must be the name of a directory, but the last component may be a non-directory file.

UNIX System V restricts component names to a maximum of 14 characters; with a 2 byte entry for the inode number, the size of a directory entry is 16 bytes. Figure 3.8 depicts the layout of the directory "etc".

Byte Offset in Directory	Inode Number (2 bytes)	File Names
0	83	.
16	2	..
32	1798	init
48	1276	fsck
64	85	clri
80	1268	motd
96	1799	mount
112	88	mknod
128	2114	passwd
144	1717	umount
160	1851	checklist
176	92	fsdb1b
192	84	config
208	1432	getty
224	0	crash
240	95	mkfs
256	188	inittab

Figure 3.8: Directory Layout for /etc

Every directory contains the file names dot and dot-dot (". " and ".. ") whose inode numbers are those of the directory and its parent directory, respectively.

The inode number of "." in "/etc" is located at offset 0 in the file, and its value is 83. The inode number of ".." is located at offset 16, and its value is 2. Directory entries may be empty, indicated by an inode number of 0.

The kernel stores data for a directory just as it stores data for an ordinary file, using the inode structure and levels of direct and indirect blocks.

Processes may read directories in the same way they read regular files, but the kernel reserves exclusive right to write a directory, thus insuring its correct structure.

The access permissions of a directory have the following meaning: *read permission* on a directory allows a process to read a directory; *write permission* allows a process to create new directory entries or remove old ones (via the creat, mknod, link, and unlink system calls), thereby altering the contents of the directory; *execute permission* allows a process to search the directory for a file name (it is meaningless to execute a directory).

CONVERSION OF A PATH NAME TO AN INODE:

The initial access to a file is by its path name, as in the open, chdir (change directory), or link system calls. Because the kernel works internally with inodes rather than with path names, it converts the path names to inodes to access files.

The algorithm namei parses the path name one component at a time, converting each component into an inode based on its name and the directory being searched, and eventually returns the inode of the input path name (figure 3.9).

```
algorithm namei /* convert path name to inode */
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 in ode is of root and component is " . . ")
```

```

{
    continue; /* loop back to while */
    read directory (working inode) 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 (algorithm iput);
        working inode = inode of matched component (algorithm iget);
    }
    else /* component not in directory */
        return (no inode);
}
    return (working inode);
}

```

Figure 3.9: Algorithm for Conversion of a Path Name to an inode

namei uses intermediate inodes as it parses a path name; call them working inodes. The inode where the search starts is the first working inode.

During the iteration of the *namei* loop, the kernel makes sure that the working inode is indeed that of a directory. Otherwise, the system would violate the assertion that non-directory files can only be leaf nodes of the file system tree.

For example, suppose a process wants to open the file *"/etc/passwd"*. When the kernel starts parsing the file name, it encounters *"/"* and gets the system root inode. Making root its current working inode, the kernel gathers in the string *"etc"*.

After checking that the current inode is that of a directory (*"/"*) and that the process has the necessary permissions to search it, the kernel searches root for a file whose name is *"etc"*: It accesses the data in the root directory block by block and searches each block one entry at a time until it locates an entry for *"etc"*.

On finding the entry, the kernel releases the inode for root (algorithm *iput*) and allocates the inode for *"etc"* (algorithm *iget*) according to the inode number of the entry just found.

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 free block numbers.

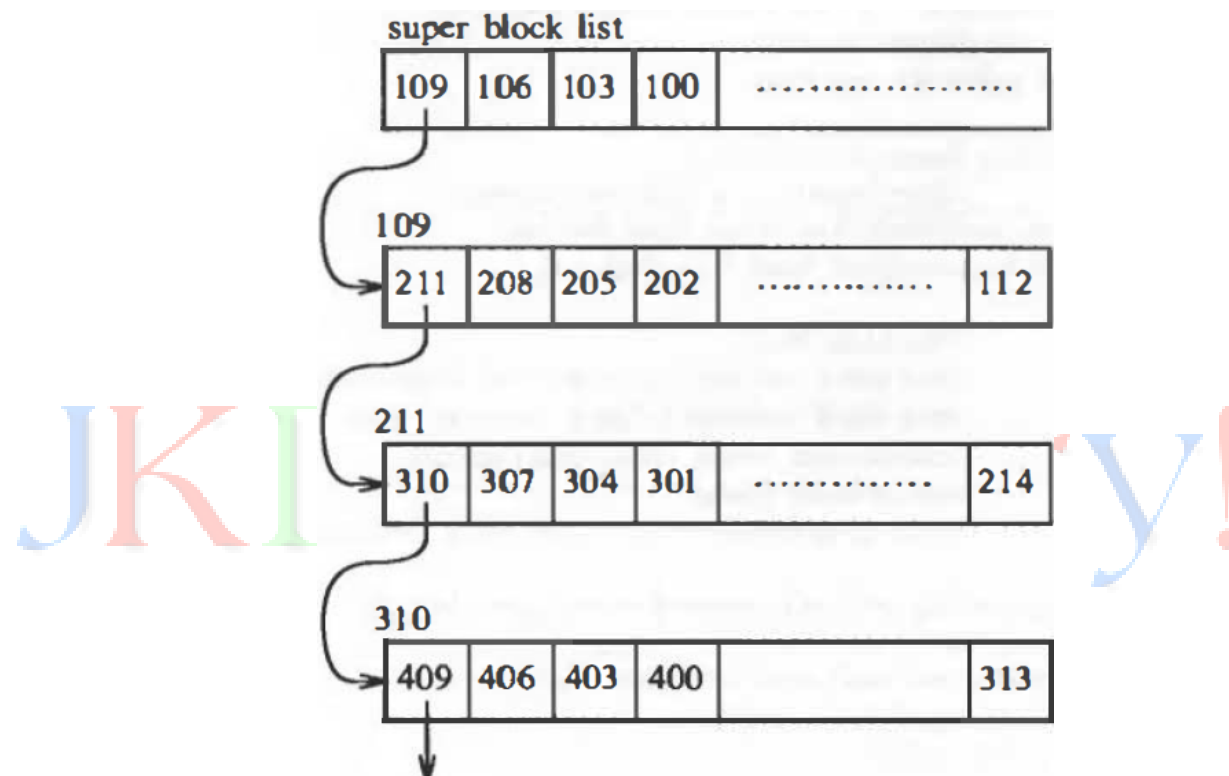


Figure 3.11: Linked List of Free Disk Block Numbers

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.

```

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 (algorithm bread);
        copy block numbers in block into super block;
        release block buffer (algorithm brelse);
        unlock super block;
        wake up processes (event super block not locked);
    }
    get buffer for block removed from super block list (algorithm getblk);
    zero buffer contents;
    decrement total count of free blocks;
    mark super block modified;
    return buffer;
}

```

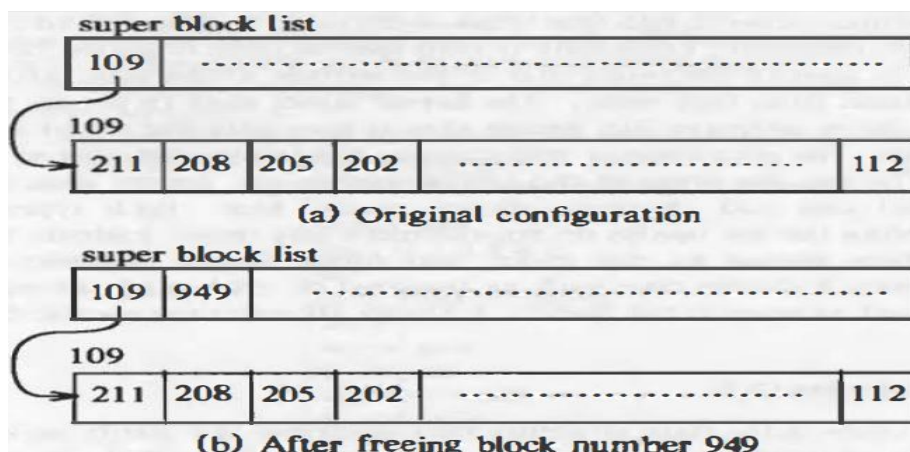
Figure 3.12: Algorithm for Allocating Disk Block

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 file 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.



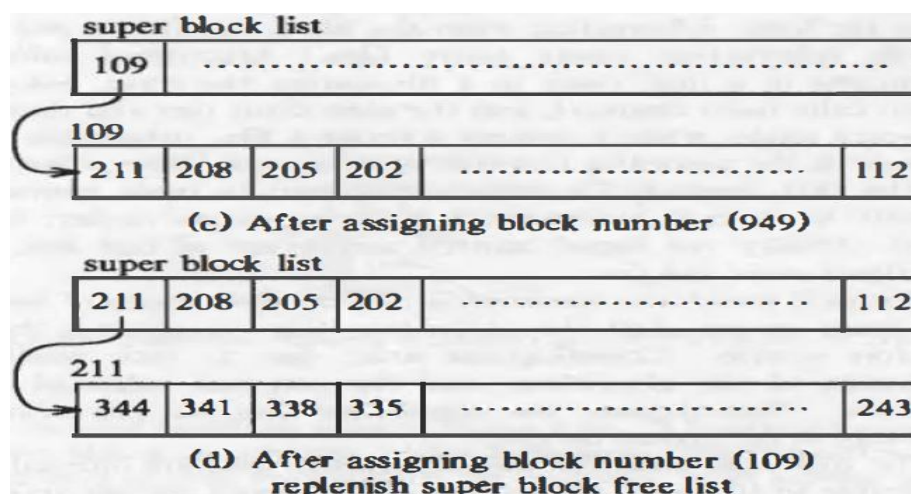


Figure: 3.13 Requesting and Freeing Disk Blocks

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.