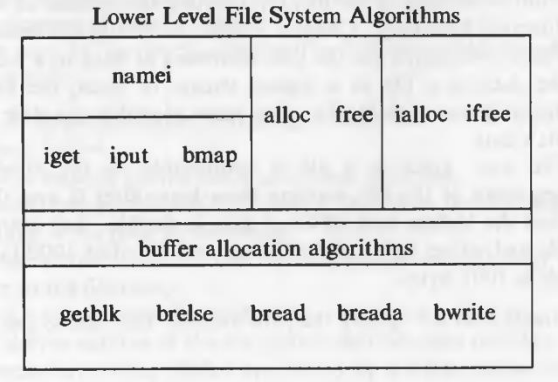
Write a short note on File system algorithms.

- File system algorithms are crucial for managing and maintaining the efficiency of file systems**.**
- These algorithms are designed to optimize various aspects of file system operations, such as file access, file migration, and file system synchronization.
- The algorithms described below are above the layer of buffer cache. Diagrammatically, it can be shown like this:



namei (Name Lookup):

- Namei, short for "name lookup," is an algorithm used to resolve pathnames provided by users or applications into corresponding inodes.
- This algorithm is used for file system traversal. It starts from the current directory and follows the path to the requested directory.
- If the path is invalid or the user does not have the necessary permissions, the algorithm returns an error.

get/iput:

- Explanation: Igget and iput are operations used for inode management within the file system.
- **iget**: This algorithm is used for file system synchronization. It ensures that changes made to a file system are propagated to all users and processes that are using the file system.
- **iput:** This algorithm is also used for file system synchronization. It releases the lock on a file or directory, allowing other processes to access it.

alloc/free

- Alloc and free are operations involved in the allocation and deallocation of disk blocks for storing file data.
- **alloc:** This algorithm is used for file system locking. It allocates a lock on a file or directory, preventing other processes from accessing it until the lock is released.
- **free:** This algorithm is used for file system locking. It frees a lock on a file or directory, allowing other processes to access it.

ialloc/ifree

- Ialloc and ifree are operations responsible for managing inodes, which store metadata about files and directories.
- **ialloc:** This algorithm is used for file system locking. It allocates a lock on a file or directory, preventing other processes from accessing it until the lock is released.

- **ifree:** This algorithm is used for file system locking. It frees a lock on a file or directory, allowing other processes to access it.

bmap (Block Mapping):

- Bmap, short for "block mapping," is an algorithm used to map logical block numbers (block numbers used by the file system) to physical block numbers on the disk.
- It takes the inode and byte offset as inputs and outputs the block number in the file system

Getblk :

- the getblk algorithm is used to retrieve a locked buffer that can be used for a block in the file system.
- The algorithm takes the file system number and block number as inputs and returns a locked buffer that can be used for the block.

Brelse :

- The brelse algorithm is used to release a locked buffer. It takes a locked buffer as input and does not have any output.
- the algorithm first wakes up all processes that are waiting for any buffer to become free and all processes that are waiting for this buffer to become free.

Bread :

- The bread algorithm is used to retrieve a buffer containing data from a block in the file system.
- It takes the file system number and block number as inputs and returns a buffer containing the data.
- It also ensures that the buffer is properly obtained from the file system and that the disk read is completed before returning the buffer.

Breada :

- The breada algorithm is used to retrieve a buffer containing data for an immediate read and an asynchronous read from a file system.
- It takes the file system number and block number for the immediate read and the file system number and block number for the asynchronous read as inputs.
- The algorithm returns a buffer containing the data for the immediate read.

bwrite :

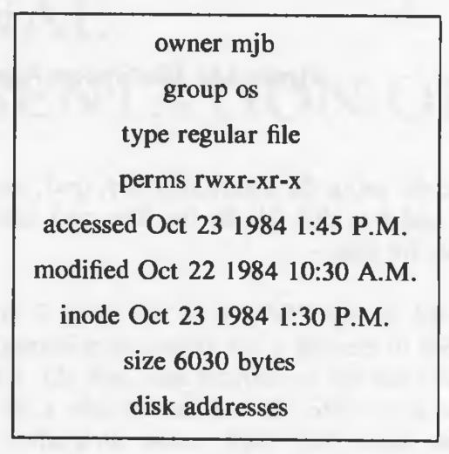
- The bwrite algorithm is used to write a buffer to disk. It takes a buffer as input and does not have any output.
- The algorithm initiates a disk write and checks if the I/O is synchronous.
- The bwrite algorithm is used to ensure that the buffer is properly written to disk and that the I/O is completed before releasing the buffer

What is INODES ? Explain Sample Disk Inode.

- Inodes are data structures that store information about files and directories in a Unix-like file system.
- They contain metadata about the file, such as ownership, permissions, access times, and file size.
- Inodes are used to locate the physical data blocks on the disk where the file's content is stored. The disk inode is a copy of the in-core inode that is stored on the disk.
- The disk node is a data structure that stores information about a file on a disk. It contains metadata about the file, such as ownership, permissions, access times, and file size.
- The table of contents in the disk node points to the physical data blocks on the disk where the file's content is stored.

Components of a disk inode:

- Owner Information: Inodes store ownership information, indicating both the user and group associated with the file or directory. The root user typically has access to all files.
- File Type: Specifies the type of the file, whether it's a regular file, directory, block or character special file, or a device file.
- File Access Permissions: Defines the access permissions for the file, including permissions for the owner, group, and others. Permissions may include read, write, and execute permissions. Execute permission on a directory allows searching inside the directory.
- Access Times: Tracks the times at which the file was last accessed, modified, and when the inode itself was last modified.
- Number of Links: Indicates the number of directory entries (hard links) that refer to the inode. When creating a new link to a file, the number of links associated with the inode increases.
- Array of Disk Blocks: Stores pointers to disk blocks where the file's data is stored. Files may be fragmented across multiple disk blocks, and this array keeps track of their locations.
- File Size: Specifies the size of the file in bytes. The size is calculated based on the maximum offset of the file data plus one. For example, if a file has data written at offset 999, its size would be 1000 bytes.



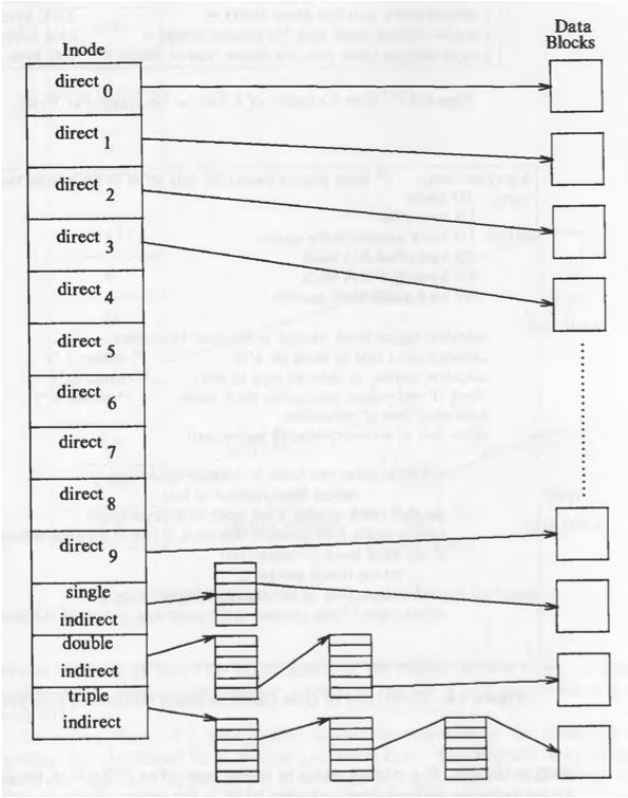
The in-core inodes, which reside in memory, contain additional fields such as:

- Status of the Inode: Indicates whether the inode is locked or if any processes are waiting for it to be unlocked.
- Change Status Flags: Reflects changes in inode data or file data, indicating if the inode or file has been modified.
- Device Number: Specifies the device number of the logical device where the file system resides.

- Inode Number: Represents the index of the inode in the inode array. This number is used to uniquely identify the inode within the file system.
- Points to Inodes: In-core inodes serve as a cache for disk inodes, with additional information and hash queues for efficient lookup and management.
- Reference Count: Indicates the number of active references to the inode. The count increases when a process allocates the inode, such as when opening a file.

Explain Structure of a regular file.

- In computing, a regular file is a type of file that contains user-created data in a specific format.
- In UNIX, a regular file is a type of file that stores data in a file system. The data in files is not stored sequentially on disk, but rather in a scattered manner.
- This is because sequential storage would require the inode to store only the starting address and size of the file, which would not be flexible and would result in large fragmentation.
- Instead, the inode stores the disk block numbers on which the data is present.
- But for such strategy, if a file had data across 1000 blocks, the inode would need to store the numbers of 1000 blocks and the size of the inode would differ according to the size of the file.
- To be able to have constant size and yet allow large files, indirect addressing is used.

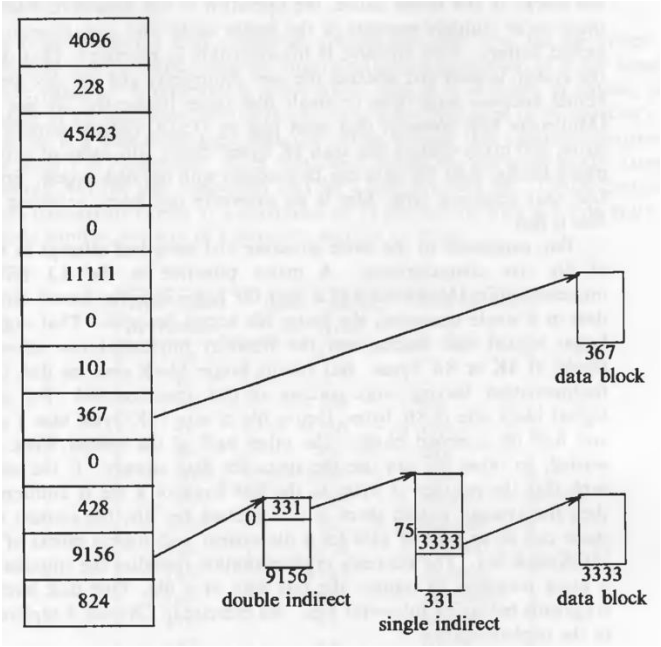


- The inode has an array of size 13 for storing block numbers, although the number of elements in the array is independent of the storage strategy.
- The first 10 members of the array are "direct addresses," which store the block numbers of actual data.
- The 11th member is "single indirect," which stores the block number of the block that has "direct addresses." The 12th member is "double indirect," which stores the block number of a "single indirect" block.
- the 13th member is "triple indirect," which stores the block number of a "double indirect" block. This strategy can be extended to "quadruple" or "quintuple" indirect addressing.
- If a logical block on the file system holds 1K bytes and a block number is addressable by a 32-bit integer, then a block can hold up to 256 block numbers.

The maximum file size with a 13-member data array is:

- 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

- However, the file size field in the inode is only 32 bits, which limits the size of a file to 4 gigabytes.
- The inode contains a table of contents that locates the file's data on disk .Each block on disk is addressable by a number.
- Example –



- To access byte offset 9000: The first 10 blocks contain 10K bytes. So 9000 should be in the first 10 block. 9000 / 1024 = 8 so it is in the 8th block (starting from 0). And 9000 % 1024 = 808 so the byte offset into the 8th block is 808 bytes (starting from 0). (Block 367 in the figure.)

Write a note on Directories.

- A directory is a container that is used to contain folders and files. It organizes files and folders in a hierarchical manner.
- They are used to organize and store files and subdirectories in a hierarchical structure.
- Directories contain entries that map file names to their corresponding inode numbers .Each directory entry takes 16 bytes, with 14 bytes allocated for the file name and 2 bytes for the inode number.

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

- The directory itself has entries for the current directory (.) and the parent directory (..).
- The inode number for these entries refers to the inode number of the directory file itself and the inode number of the parent directory, respectively.
- Entries with an inode number of 0 indicate that the corresponding file is empty or deleted.
- The kernel has exclusive write access to directories. Access permissions for directories have different meanings.
- The read permission allows users to read the contents of the directory, such as the file names and their corresponding inode numbers.
- The write permission grants users the ability to create new files and directories within the directory. The execute permission enables users to search for files and subdirectories within the directory.

Explain with algorithm for Conversion of a Path Name to an Inode.

- The conversion of a path name to an inode is essential for accessing files and directories in a file system.
- When a user requests access to a file or directory, the operating system needs to determine the inode number associated with that file or directory.
- The inode number is used to locate the file's data on disk and to perform various operations such as reading, writing, or modifying the file.
- The algorithm namei is used for converting a path to an inode number. The kernel parses the path by accessing each inode in the path and finally returns the inode of the required file.
- Every process has a current directory. The current directory of process 0 is the root directory.
- For every other process, it is the current directory of its parent process. Later the process can change the current directory with the system call chdir.

Namei algorithm –

```
/*  Algorithm: namei
 *  Input: pathname
 *  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 a directory and access permissions are OK;
                if (working inode is of root and component is "..")
                        continue;
                read directory (working inode) by repeated use of algorithms: bmap, bread, brelse;
                if (component matches an entry in the directory (working inode)
                {
                        get inode number for matched component;
                        release working inode (algorithm: iput);
                        working inode = inode of matched component (algorithm: iget);
                }
                else
                        return (no inode)  // component not in the directory
        }

        return (working inode);     }
```

- The algorithm namei takes a pathname as input and returns a locked inode.
- If the path name starts from the root, the working inode is set to the root inode using the iget algorithm.
- Otherwise, the working inode is set to the current directory inode using the iget algorithm.
- If the kernel finds a match, it records the inode number of the matched directory entry, releases the block and the old working inode, and allocates the inode of the match component.
- If the kernel does not match the path name in the block, it releases the block, adjusts the byte offset by the number of bytes in a block, converts the new offset to a disk block number and reads the new block.

What is Superblock?

- The superblock is essentially file system metadata and defines the file system type, size, status, and information about other metadata structures (metadata of metadata).
- The superblock is very critical to the file system and therefore is stored in multiple redundant copies for each file system.

Redundancy and Importance:

- Due to its critical nature, the superblock is stored in multiple redundant copies within the file system.
- If the superblock of a partition becomes corrupt, the corresponding file system cannot be mounted by the operating system, making the superblock vital for file system integrity and accessibility.
- In such cases, file system repair utilities like fsck (file system consistency check) can attempt to recover the file system using alternate backup copies of the superblock.

Location and Backup Copies:

- Backup copies of the superblock are stored in block groups spread throughout the file system.
- The first backup copy is typically located at a specific offset from the start of the partition.
- Information about backup superblock locations can be retrieved using utilities like dumpe2fs.

Contents of the Superblock:

- The superblock contains various pieces of information essential for file system management, including:
- Size of the file system.
- Number of free blocks and a list of free blocks in the file system.
- Pointer to the next free block in the free blocks list.
- Size of the inode list.
- Number of free inodes and a list of free inodes in the file system.
- Pointer to the next free inode in the free inodes list.

- Lock fields for the free blocks and free inodes lists.
- Field indicating whether the superblock has changed.

Kernel Operations and Consistency:

- The kernel periodically writes the superblock to the disk if it has been modified, ensuring that it remains consistent with the data on the disk.
- This ensures that the file system metadata remains up-to-date and accurately reflects the current state of the file system.

Explain race condition in Assigning Inodes.

- Race condition is a problem that can occur when assigning inodes in a file system. It happens when multiple processes try to assign the same inode number to different files, and one of the processes succeeds in assigning it before the other process can check if the inode is already in use.
- This can lead to incorrect file type information in the in-core copy of the inode, as the other process may find the inode already assigned and with its file type set.
- To prevent this race condition, the algorithm ialloc checks if the inode number is already in use before assigning it to a new file.

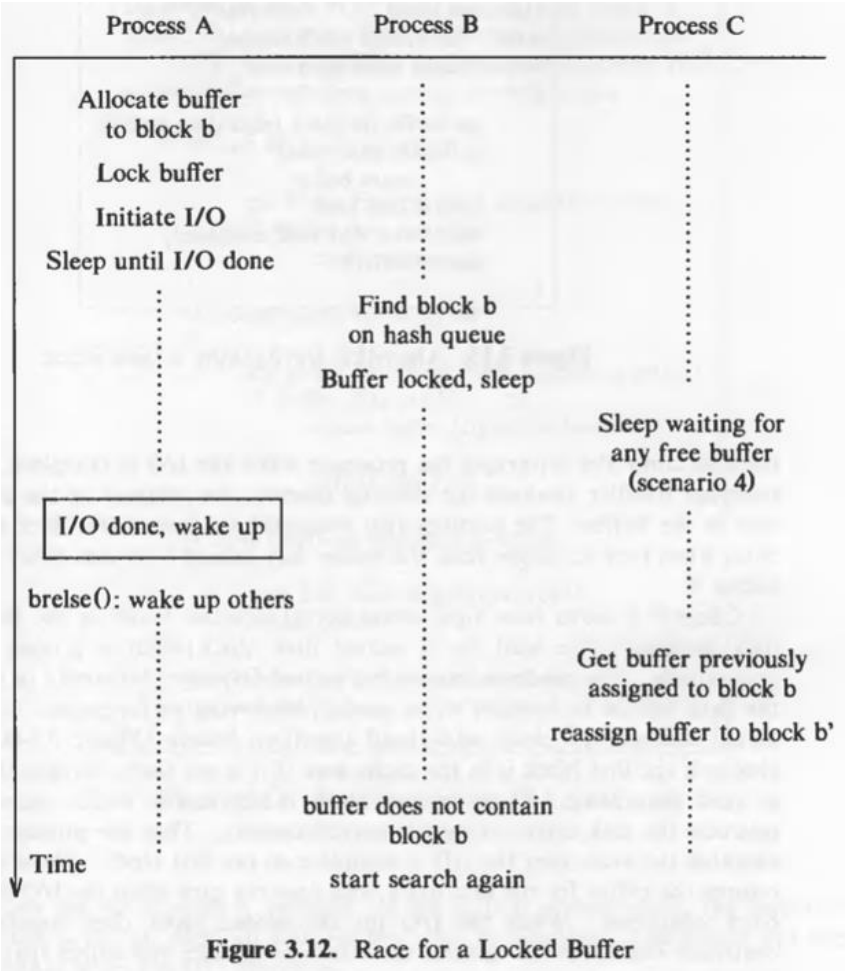


Figure 3.12. Race for a Locked Buffer

Following is the working of ialloc algorithm –

1. The algorithm starts by checking if the super block is locked.
2. If the super block is locked, the algorithm sleeps until it becomes free.
3. If the inode list in the super block is empty, the algorithm locks the super block and searches the disk for free inodes.
4. The algorithm unlocks the super block and wakes up the process.
5. If no free inodes were found on the disk, the algorithm returns no inode.
6. The algorithm sets the remembered inode for the next free inode search.

7. If there are inodes in the super block inode list, the algorithm gets the inode number from the list and gets the inode using the iget algorithm.
8. The algorithm checks if the inode is free after all.
9. If the inode is not free, the algorithm writes the inode to disk and releases it using the iput algorithm.
10. If the inode is free, the algorithm initializes the inode, writes it to disk, decrements the file system free inode count, and returns the inode.
11. The algorithm continues until it has found a free inode or until it has searched the entire disk.

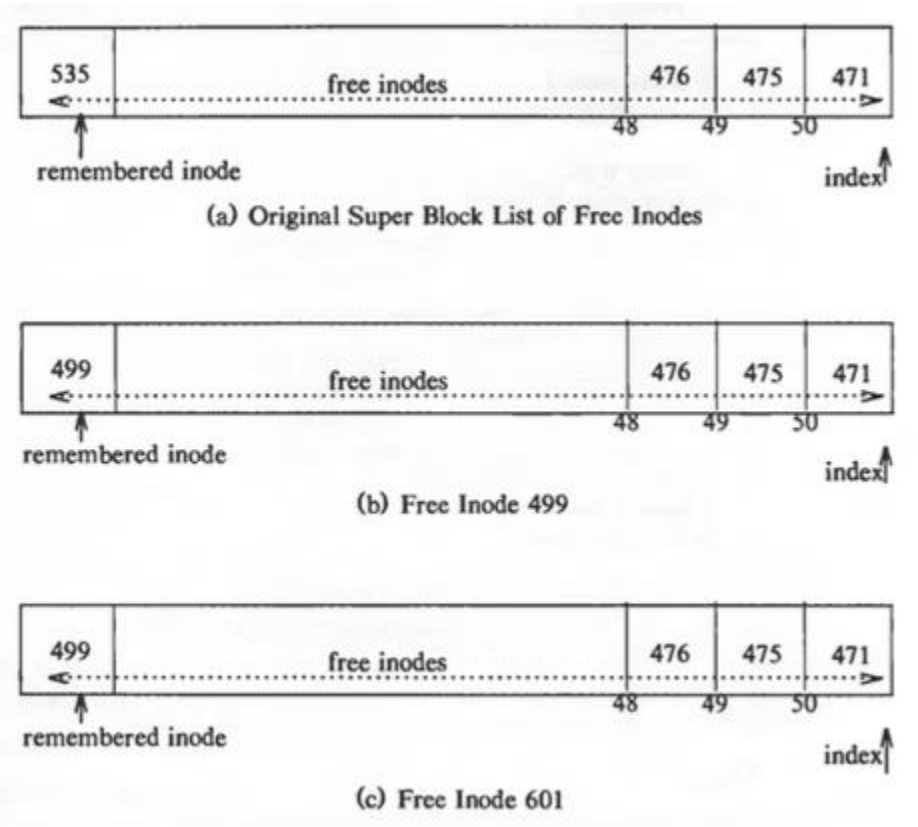
Explain algorithm for freeing Inode.

- The algorithm for freeing inodes (ifree) is a simple one that increments the file system free inode count and updates the inode list accordingly.
- The algorithm checks if the super block is locked and returns if it is. If the inode list is full, the algorithm checks if the input inode number is less than the remembered inode number for free inode search.
- If it is, the remembered inode number is updated to the input inode number. Otherwise, the inode number is stored in the inode list. The algorithm returns after completing the freeing process.

```
/*  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 number less than remembered inode for search)
                        set remembered inode for search = input inode number;
        }
        else
                store inode number in inode list;
        return;
}
```

- If the super block is locked, the algorithm returns, the inode number is not updated in the free list of inode numbers.
- Ideally, there should never be free inodes whose inode number is less than the remembered inode number, but exceptions are possible.
- If an inode is being freed and the super block is locked, in such situation, the it is possible to have an inode number that is free and is less than the remembered inode number.

An example:


535
free inodes
476
475
471
48
49
50
remembered inode
index
(a) Original Super Block List of Free Inodes
499
free inodes
476
475
471
48
49
50
remembered inode
index
(b) Free Inode 499
499
free inodes
476
475
471
48
49
50
remembered inode
index
(c) Free Inode 601

Explain in brief Inode assignment to a new file.

- The algorithm ialloc is utilized to assign an inode to a newly created file within a file system.

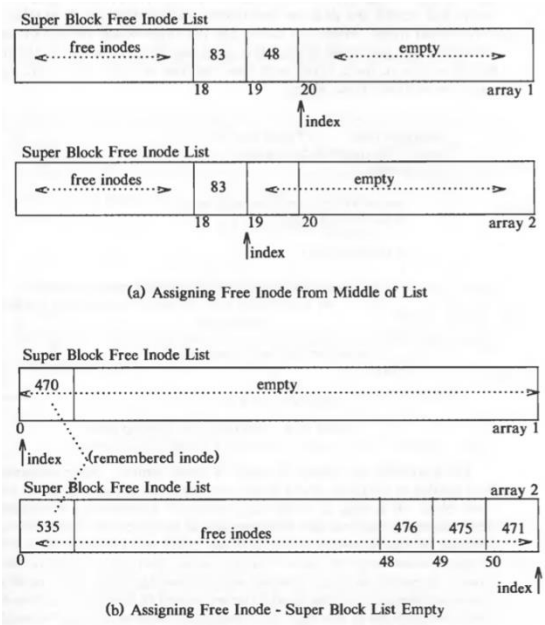
```
/*  Algorithm: ialloc
 *  Input: file system
 *  Output: locked inode
 */

{
        while (not done)
        {
                if (super block locked)
                {
                        sleep (event: super 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
(algorithm: 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 form super block inode list;
                get inode (algorithm: iget);
                if (inode not free after all)
                {
                        write inode to disk;
                        release inode (algorithm: iput);
                        continue;
                }
                // inode is free
                initialize inode;
                write inode to disk;
                decrement file system free inode count;
                return inode;
        }
}
```

1. **Input and Output**:
   - Input: File system.
   - Output: Locked inode for the newly created file.
2. **Initialization and Loop**:
   - The algorithm operates within a loop until the assignment of the inode is completed.
3. **Handling Superblock Lock**:

- If the superblock is locked, indicating that it is currently being accessed or modified by another process, the algorithm sleeps until the superblock becomes free again.

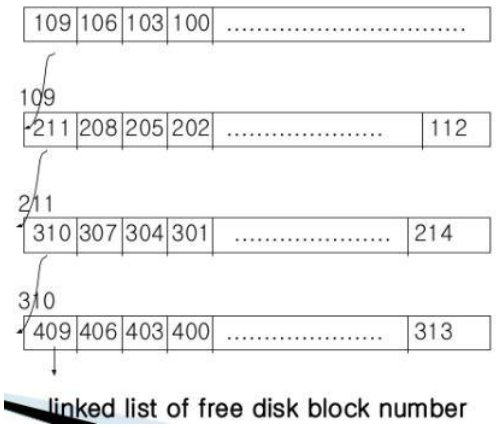
4. **Checking Inode List in Superblock**:
   - If the list of inode numbers in the superblock is empty:
     - The superblock is locked to prevent concurrent access.
     - The algorithm retrieves the remembered inode number for free inode search, which represents the starting point for searching disk blocks for free inodes.
     - It searches disk blocks for free inodes until either the superblock is full or no more free inodes are found.
     - After completing the search, the superblock is unlocked, and the algorithm waits until the superblock becomes free again.
     - If no free inodes are found on the disk, the algorithm returns indicating that there is no available inode.
     - The remembered inode for the next free inode search is updated to continue the search efficiently.
5. **Assigning Inode from Superblock List**:
   - If the list of inode numbers in the superblock is not empty:
     - The algorithm retrieves the next inode number from the superblock inode list.
     - It obtains an in-core inode for the newly assigned disk inode using the iget algorithm.
     - If the inode is found to be not free after all (possibly due to a race condition), it is written back to disk, and the in-core inode is released before continuing the loop.
     - If the inode is free, it is initialized, written back to disk, and the file system's free inode count is decremented.
     - The locked inode for the newly created file is returned.
6. **Optimization with Remembered Inode**:
   - By using the remembered inode from previous searches, the algorithm optimizes the process of searching for free inodes on the disk, ensuring efficiency and avoiding unnecessary reads of disk blocks where no free inodes are expected to exist.

An example:



Explain with diagram 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 indirect blocks as needed.
- The file system super block contains an array that keeps track of the numbers of free disk blocks in the file system.
- The last block in the super block points to a block that contains a list of free blocks.



linked list of free disk block number

- When a file system is created (using mkfs command), the data blocks are organized 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.

Major Steps:

- When the kernel wants to allocate a block from a file system, it allocates the next available block in the super block array. Once allocated, the block cannot be reallocated until it becomes free.
- If the allocated block is the last available block in the super block array, the kernel treats it as a pointer to a block that contains a list of free blocks.
- In such case, it reads the block, populates the super block array with the new list of block numbers, and then proceeds to allocate these block numbers one by one.
- It allocates a buffer for the available free block and clears the buffer's data. (Now, a disk block is allocated and the kernel has a buffer ready to write the data to the disk.)
- If the file system contains no free blocks, the calling process receives an error.
- If a process writes a lot of data to a file, it repeatedly asks the kernel for blocks to store the data, but the kernel assigns only one block at a time.

```
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;
copy block numbers in block into
super block; unlock super block;
```

```
wakeup (event: super block not locked);
}
get buffer for block removed from
super block list; clear buffer contents;
decrement total count of free blocks;

mark super block
modified; return
buffer;
}
```