Internal Representation of Files

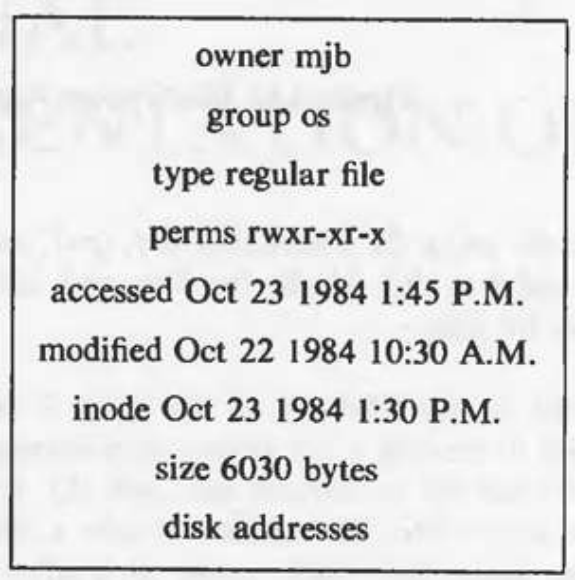
Give the fields of in-core copy of Inode.
What is Inode? Summarize the fields from disk inode

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.

Disk inodes consists of the following fields:

- **Owner information**: ownership is divided into a user and a group of users. Root user has access to all the files.
- **File type**: it states whether a file is a normal file, a directory, a block or character special file, or a device file.
- **File access permissions**: there are 3 types of access permissions: owner, group and others. There are separate permissions for reading, writing and executing. Since execute permission is not applicable to a directory, execute permission for a directory gives the right to search inside the directory.
- **Access times**: the times at which the file was last accessed and last modified, and the time at which the inodes was last modified
- **Number of links**: number of places from which the file is being referred.
- **Array of disk blocks**: even if the users get a logically sequential representation of data in files, the actual data is scattered across the disk. This array keeps the addresses of the disk blocks on which the data is scattered.
- **File size**: the addressing of the file begins from location 0 from relative to the starting location and the size of the file is the maximum offset of the file + 1. For example, if a user creates a file and writes a byte at offset 999, the size of the file is 1000.
  The inode does not specify the pathname/pathnames that access the file.

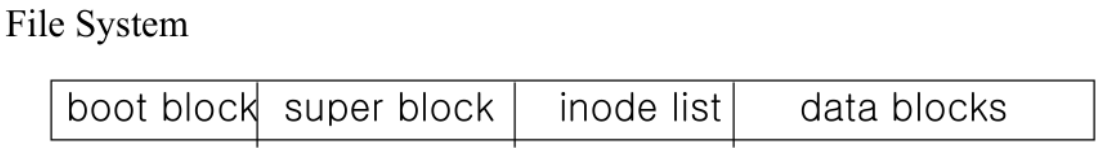


The in-core inodes contain the following fields in additional to the fields of the disk inode:

- **Status of the inode**
    i. Locked.
    ii. A process is (or many processes are) waiting for it to be unlocked.
    iii. The data in the inode differs from the disk inode due to change in the inode data.
    iv. The data in the inode differs from the disk inode due to change in the file data.
    v. The file is a mount point (discussed later).
- The device number of the logical device of the file system on which the file resides.
- **Inode number**: the disk inodes are placed in an array. So the number of the inode is nothing but the index of the inode in the array. That is why the disk copy does not need to store the inode number.
- **Points to inodes**: just like the buffer cache, the in-core inodes are nothing but a cache for disk inodes (but with some extra information which is deterministic). In-core inodes also have hash queues and a free list and the lists behave in a very similar way to the buffer lists. The inode has next and previous pointers to the inodes in the hash queue and free lists. The hash queues have a hash function based on the device number and inode number.
- **Reference count**: it gives the number of instances of files that are active currently.

What is superblock? List and explain various fields of superblock.

The superblock in a Unix file system is a critical component that contains essential metadata about the file system, such as its size, status, and information about other metadata structures. It is vital for file system integrity and accessibility, and redundancy measures are typically implemented to ensure its availability in case of corruption. Below are the various fields of the superblock and their explanations:



1. **Size of the File System**:
   Represents the total size of the file system, including all blocks and inodes.
2. **Number of Free Blocks**:
   Indicates the count of blocks currently available for allocation to files.
3. **List of Free Blocks**:
   A collection of blocks that are currently free and available for allocation. This list allows efficient management of available storage space.
4. **Index of the Next Free Block**:
   Specifies the index or location of the next free block in the list of free blocks. It helps the kernel quickly identify the next available block for allocation.
5. **Size of the Inode List**:
   Denotes the total number of inodes available in the file system. Inodes contain metadata about files and directories.
6. **Number of Free Inodes**:
   Represents the count of inodes currently available for allocation to files.
7. **List of Free Inodes**:
   A collection of inodes that are currently free and available for allocation. This list enables efficient management of available inode resources.
8. **Index of the Next Free Inode**:
   Indicates the index or location of the next free inode in the list of free inodes. It helps the kernel quickly identify the next available inode for allocation.
9. **Lock Fields for Free Block and Free Inode Lists**:
   These fields are used to manage concurrent access to the lists of free blocks and inodes. Locking mechanisms ensure data integrity and prevent conflicts when multiple processes attempt to modify these lists simultaneously.
10. **Flag Indicating Super Block Modification**:
    A flag that indicates whether the superblock has been modified since the last time it was written to disk. This flag helps the kernel determine whether it needs to update the superblock on disk, ensuring that file system metadata remains up-to-date and consistent.

Explain the algorithm for conversion of pathname to Inode.

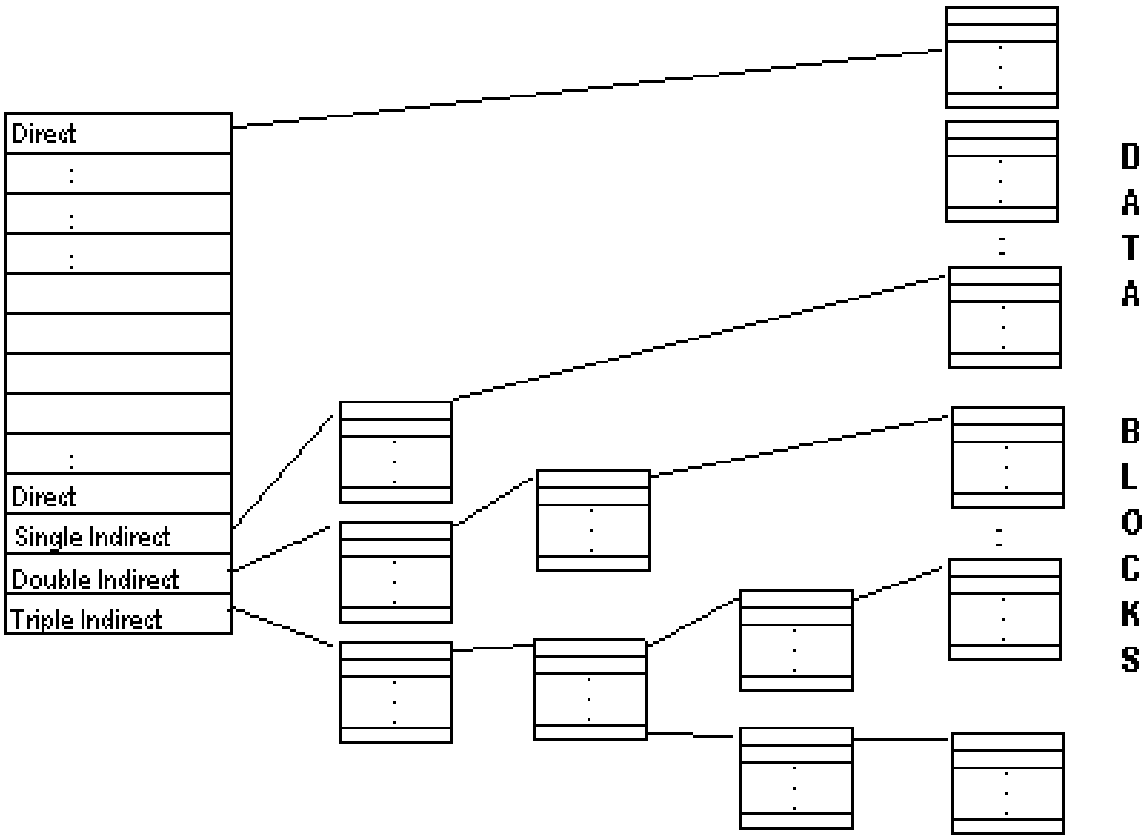
The algorithm for converting a pathname to an inode is crucial for navigating and accessing files within a Unix-like file system. Here's an explanation of the algorithm:

```
algorithm namei /*conven path name to inode */
input: path name
output: locked inode
{
    if (path name starts from root)
        working inode - roo t 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
          brnap. bread and brel.sc;
     if (component matches an entry in directory (working inode))
     {
          get inode number for matched component;
          release working inode (algorithm iput);
          working inode - inode of matclted component (algo iget);
     }
     else  /* component not in directory */
          return (no inode);
}
          return (working inode);
```

1. **Initialization**:
    - The algorithm, named **namei**, begins with the initialization phase. It takes the pathname as input and aims to output the locked inode corresponding to the provided pathname.
2. **Starting Inode**:
    - If the pathname starts from the root ("/"), the algorithm assigns the root inode to the working inode. This root inode is obtained using the **iget** algorithm.
    - If the pathname does not start from the root, the algorithm assigns the current directory inode to the working inode.
3. **Parsing Pathname**:
    - The algorithm iterates through each component of the pathname.
    - It reads the next component from the input pathname.
    - It verifies that the working inode is a directory and that the access permissions are appropriate for traversing the directory.
4. **Directory Traversal**:
    - If the working inode is the root inode and the current component is "..", the algorithm checks for more components in the pathname. If there are more components, it continues the loop.
    - Otherwise, it reads the directory contents of the working inode by repeatedly using algorithms like **bmap**, **bread**, and **brelse**.
    - If the component matches an entry in the directory, the algorithm retrieves the inode number for the matched component.
    - It then releases the working inode and assigns the inode of the matched component as the new working inode.
    - If the component is not found in the directory, the algorithm returns with no inode found.
5. **Returning Result**:
    - After traversing all components of the pathname, the algorithm returns the working inode, which is now locked and corresponds to the final component of the pathname.

Explain the structure of a Regular file.



The structure of a regular file in a Unix-like operating system, such as System V UNIX, is based on the organization of data blocks on the disk and the use of inodes to manage file metadata. Here's an explanation of the structure of a regular file:

1. Data Block Organization:

- Each regular file consists of a sequence of data blocks, where each block contains a fixed amount of data.
- The data blocks are linked together to form the content of the file.

2. Inode Management:

- Inodes are data structures that store metadata about files, such as ownership, permissions, access times, and file size.
- The inode for a regular file contains a table of contents that locates the file's data blocks on the disk.

3. Allocation of Data Blocks:

- The data blocks of a regular file are allocated dynamically as needed.
- The kernel allocates file space one block at a time, allowing the data in a file to be spread across the file system.

4. Block Layout:

- The block layout for a regular file typically includes:
    - **Direct blocks**: Contain the actual data of the file.
    - **Indirect blocks:** Used if the file size exceeds the capacity of direct blocks. They contain pointers to additional data blocks.
    - **Double indirect blocks:** Point to multiple single indirect blocks, further expanding the file's storage capacity.
    - **Triple indirect blocks:** Provide access to double indirect blocks, allowing for significant file size and data storage.

5. Accessing Data:

- Processes access data within a regular file by byte offset, viewing it as a stream of bytes.
- The kernel accesses the inode of the file and converts logical file blocks into appropriate disk blocks for data retrieval.

6. Block Indexing:

- The logical block numbers in the inode are used to index the file's data blocks on the disk.
- The kernel calculates the physical disk block corresponding to a given byte offset in the file using the **bmap** algorithm.

Example:

- For example, if a process wants to access byte offset 9000 in a file, the kernel calculates that the byte is in direct block 8. It then accesses the corresponding disk block number to retrieve the data.

Unused Blocks:

- Unused blocks in the file are indicated by block entries in the inode with a value of 0, meaning that they contain no data.
- No disk space is wasted for such blocks, and the kernel efficiently manages read and write operations involving these blocks.