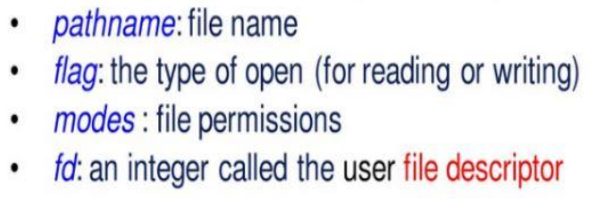
Explain OPEN System call in brief.

- The open system call is the first step a process must take to access the data in a file. The open system call is used to provide access to a file in a file system.
- It allocates resources to the file and provides a handle that the process uses to refer to the file.
- A file can be opened by multiple processes at the same time or be restricted to one process, depending on the file organization and file system.
- The syntax for the open system call is

fd = open (pathname, flags, modes); where,



- *pathname*: file name
- *flag*: the type of open (for reading or writing)
- *modes* : file permissions
- *fd*: an integer called the user file descriptor

- Executing open system call will return an integer which is a file descriptor.

algorithm: open (algorithm for opening a file)

input: file name type of open file permissions (for creation type of open)

output: file descriptor

```
{
 convert file name to inode (algorithm namei);
if (file does not exist or not permitted access)
 return(error)
 allocate file table entry for inode, initialize count, offset;
 allocate user file descriptor entry, set pointer to file table entry;
if (type of open specifies truncate file)
free all file blocks (algorithm free);
 unlock(inode); /* locked above in namei */
return(user file descriptor);
```

The algorithm for opening a file involves the following steps:

1. The kernel searches the file system for the file name parameter using algorithm "namei".
2. Once the file is found, the kernel checks the permissions for opening the file.
3. If the file exists and the process has the necessary permissions to access it, the kernel allocates an entry in the file table for the open file.
4. The file table entry contains a pointer to the inode of the open file and a field that indicates the byte offset in the file where the kernel expects the next read or write to begin.
5. The kernel then returns the user file descriptor, which is used in subsequent file operations.

- The above pic shows the relationship between the inode table, file table and user file descriptor data structures :

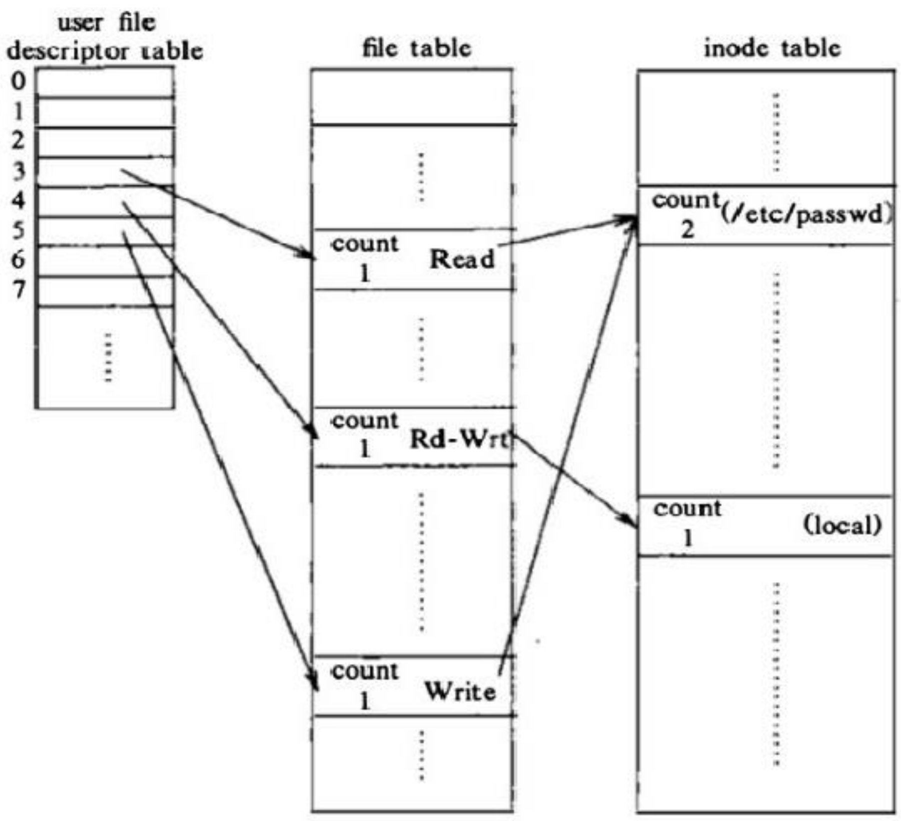


Figure 5.3. Data Structures after Open

- Each open returns a file descriptor to the process, and the corresponding entry in the user file descriptor table points to a unique entry in the kernel file table even through one file ("/etc/passwd") is opened twice.
- The file table entries of all instances of an open file point to one entry in the in-core inode table.
- The process can read or write the file "/etc/passwd" read or write but only through file descriptor 3 and 5 as shown in the following pic.
- The kernel notes the capability to read or write the file in the file table entry allocated during the open call.

Write algorithm for Reading a file.

- The **read** system call is a fundamental operation in file input/output (I/O) operations within Unix-like operating systems.
- Syntax of the read call is as follows –

  number = read (fd, buffer, count); Where,

  fd is the descriptor returned by *open*.

  buffer is the address of the data structure where the data will be read.

  count is the number of bytes to be read.

  And it returns how many bytes were successfully read.
- Read() tells the operating system to read "size" bytes from the file opened in file descriptor "fd", and to put those bytes into the location pointed to by "buf". It returns how many bytes were actually read.
- The algorithm is given below:

```
/*  Algorithm: read
 *  Input: user file descriptor
 *        address of buffer in user process
 *        number of bytes to be read
 *  Output: count of bytes copied into user space
 */

{
        get file table entry from user file descriptor table;
        check file accessibility;
        set parameters in u-area for user address, byte count, I/O to user;
        get inode from file table;
        lock inode;
        set byte offset in u-area from file table offset;
        while (count not satisfied)
        {
            convert file offset to disk block (algorithm: bmap);
            calculate offset into block, number of bytes to read;
            if (number of bytes to read is 0)
                    break;                  // trying to read End Of File (EOF)
            read block (algorithm: bread or breada whichever applicable);
            copy data from system buffer to user address;
            update u-area fields for file byte offset, read count, address to write into user space;
            release buffer;         // locked in bread
        }
        unlock inode;
        update file table offset for next read;
        return (total number of bytes read);
}
```

After getting the file table entry from user file descriptor table, the kernel sets some parameters in the u-area and eliminates the need to pass them as function parameters. The parameters in the u-area:

- mode: indicates read or write
- count: count of bytes to read or write
- offset: byte offset in file
- address: target address to copy data in user or kernel memory
- flag: indicates if address is in user or kernel memory

- If a process reads two blocks sequentially, the kernel assumes that all subsequent reads will be sequential until proven otherwise.
- During each iteration through the loop, the kernel saves the next logical block number in the in-core inode and during the next iteration, compares the current logical block number to the value previously saved.
- If they are equal, the kernel calculates the physical block number for read-ahead and saves its value in the u-area for use in the *breada* algorithm.

Write a note on WRITE system call.

- The write system call in an operating system is used to write data from a user process to a file.
- It is the counterpart of the read system call, which is used to read data from a file.
  The write system call takes the same arguments as read: the user file descriptor, the address in the user space where the data is located, and the number of bytes to be written.
- The algorithm for the write system call is as follows:

```
/* Algorithm: write
 *  Input: user file descriptor
 *       address in the user space from where data is to be written
 *       number of bytes to be written
 *  Output: count of bytes written in the file
 */
{
  get file table entry from user file descriptor table;
  check file accessibility;
  set parameters in u-area for user address, bytes count, I/O to user;
  get inode from file table;
  lock inode;
  set bytes offset in u-area from file table offset;
  while (count not satisfied)
  {
    if (file offset is larger than the file size)
    {
      while (indirection is required)
      {
        allocate a new block for indirection (algorithm: alloc);
        write the block number in the parent block;
      }
      allocate a new block for data (algorithm: alloc);
      write the block number in the inode/indirect block;
    }
    else
      convert file offset to disk block (algorithm: bmap);
    calculate the offset into block, number of bytes to write;
    if (only part of the block to be written)
    {
      read the buffer (algorithm: bread);
    }
    write data in the buffer;
    write block (algorithm: bwrite);
    update u-area fields for file byte offset, write count, address to read from user space;
  }
  update the file size if required;
  unlock inode;
  update file table offset for next write;
  return (total number of bytes read)
}
```

- The algorithm is similar to the read algorithm, but the actions are reversed. When the file offset is larger than the file size, the kernel has to allocate a new data block and indirect blocks if necessary.
- When a block is to be written partially, the kernel has to read the block, copy the contents that are not going to be overwritten, and then write the block.
- Finally, the file size is updated if required, the inode is unlocked, and the file table offset is updated for the next write. The total number of bytes written is returned.

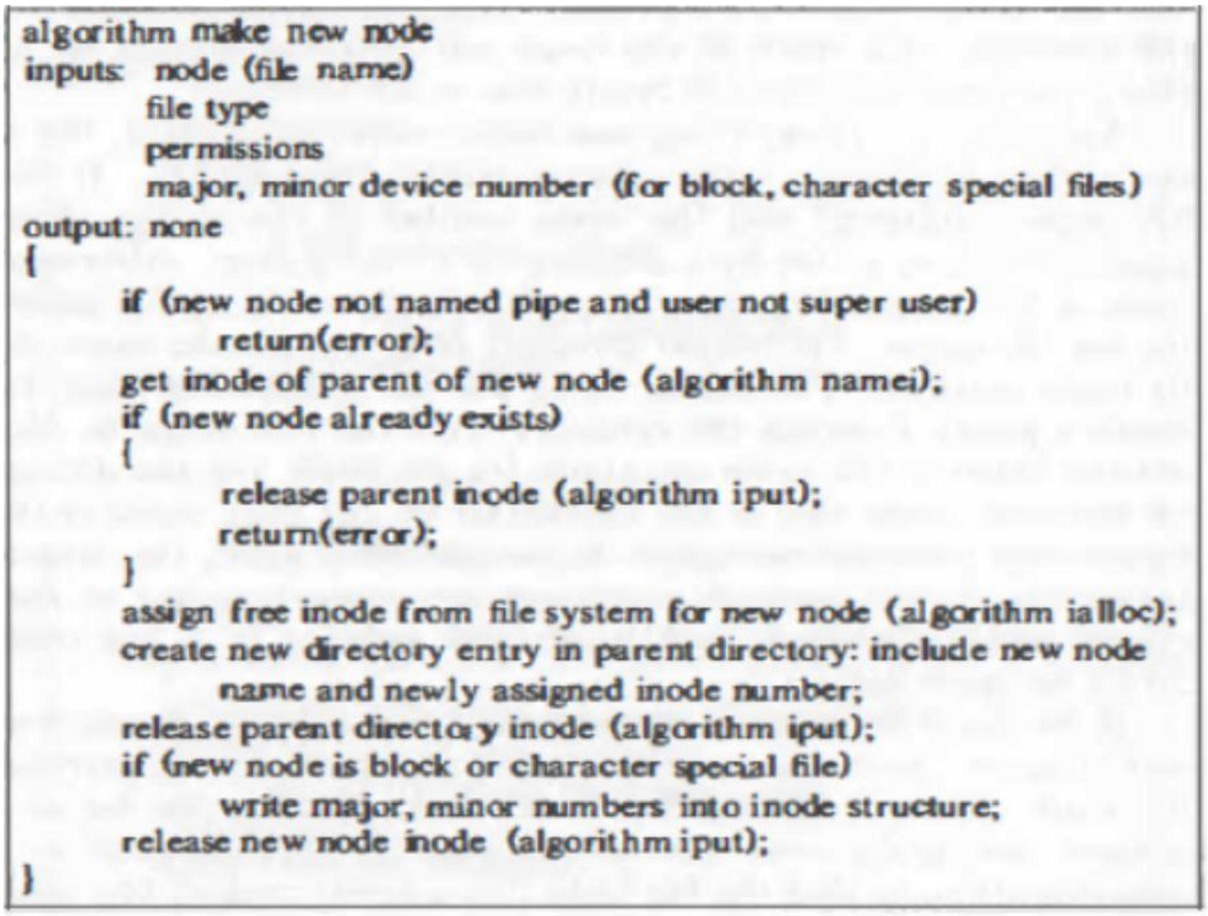
Write and explain algorithm for Making new node.

- The algorithm takes the file name, file type, permissions, and device numbers as input. It first checks if the new node is not a named pipe and if the user is not a super user.
- The system call mknod creates special files in the system, including named pipes, device files, and directories. It is similar to creat in that the kernel allocates an inode for the file.
- The syntax of the mknod system call is:

mknod(pathname, type and permissions, dev);

Where ,

- **pathname** is the name of the node to be created**.**
- **type and permissions** give the node type (directory, for example) and access permissions for the new file to be created.
- **dev** specifies the major and minor device numbers for block and character special files.
- **The algorithm mknod for making a new node :**



```
algorithm make new node
inputs: node (file name)
        file type
        permissions
        major, minor device number (for block, character special files)
output: none
{
    if (new node not named pipe and user not super user)
        return(error);
    get inode of parent of new node (algorithm namei);
    if (new node already exists)
    {
        release parent inode (algorithm iput);
        return(error);
    }
    assign free inode from file system for new node (algorithm ialloc);
    create new directory entry in parent directory: include new node
        name and newly assigned inode number;
    release parent directory inode (algorithm iput);
    if (new node is block or character special file)
        write major, minor numbers into inode structure;
    release new node inode (algorithm iput);
}
```

1. **Check for Permissions**: Before creating a new node, the algorithm checks if the user has sufficient privileges to perform the operation. If the new node is not a named pipe and the user is not a superuser (root), the algorithm returns an error.

2. **Get Inode of Parent Directory**: The algorithm retrieves the inode of the parent directory of the new node using the **get_inode_of_parent** function. This is necessary to add a new directory entry for the new node.

3. **Check for Existing Node**: It verifies if the new node already exists in the parent directory. If it does, the algorithm releases the parent directory inode and returns an error.

4. **Allocate Free Inode**: The algorithm allocates a free inode from the file system for the new node using the **allocate_free_inode** function.
5. **Create Directory Entry**: It creates a new directory entry in the parent directory, including the name of the new node and the inode number assigned to it, using the **create_directory_entry** function.
6. **Release Parent Directory Inode**: The algorithm releases the parent directory inode, as it is no longer needed for further operations.
7. **Write Major and Minor Numbers**: If the new node is a block or character special file, the algorithm writes the major and minor device numbers into the inode structure of the new node.
8. **Release New Node Inode**: Finally, the algorithm releases the inode of the new node, as the creation process is complete.

Write note on Creation of Special files.

- The creation of special files in an operating system involves the use of the mknod system call, which creates special files such as named pipes, device files, and directories.
- The syntax of the mknod system call is as follows:

mknod(pathname, type and permissions, dev);

where:

- **pathname** is the name of the node to be created.
- **type and permissions** give the node type (directory, for example) and access permissions for the new file to be created.
- **dev** specifies the major and minor device numbers for block and character special files.

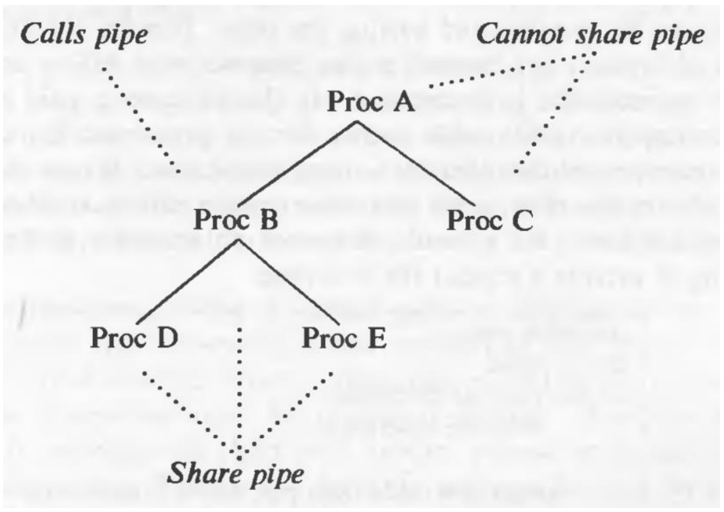
The algorithm for creating a special file using mknod is given below:

```
/*  Algorithm: mknod
 *  Input: pathname
 *       file type
 *       permissions
 *       major, minor device numbers (for block and character special files)
 *  Output: none
 */
{
        if (new node not named pipe and user not super user)
        return (error);
    get inode of parent of new node (algorithm: namei);
    if (new node already exists)
      {
         release parent inode (algorithm: iput);
        return (error);
      }
     assign free inode from file system of new node (algorithm: ialloc);
    create new directory entry in parent directory with new node name and new inode number;
     release parent directory inode (algorithm: iput);
    if (new node is block or character special file)
    write major, minor numbers into inode structure;
     release new node inode (algorithm: iput);
}
```

- The algorithm checks if the new node is not a named pipe and if the user is not a super user. If these conditions are not met, an error is returned.
- It also sets the file type field to indicate if its a pipe, directory or a special file. If a directory is being created, proper format for the directory is set (such as setting the "." and ".." entries).
- It then gets the inode of the parent directory of the new node using the namei algorithm. If the new node already exists, the parent inode is released, and an error is returned.
- This algorithm ensures that a special file is created with the correct permissions, file type, and device numbers, and that it is properly linked to the parent directory and file system.

Explain in brief PIPES.

- Pipes are a mechanism in operating systems that allow transfer of data between processes in a first-in-first-out (FIFO) manner.
- There are two kinds of pipes: *named pipes* and *unnamed pipes*. Both are identical, except the way processes initially accesses them.
- Named pipes are created with *open* and unnamed pipes are created with the *pipe* system call. Afterwards, *read*, *write*, and *close* are used for further operations on pipes.
- Access to unnamed pipes is shared only for child processes. For example, in the following diagram, process B has called *pipe* so that pipe can be used only by process B, D, and E. But process A and C cannot access the pipe.
- However, named pipes can be shared between any processes.



- The Pipe System Call syntax

  pipe (fdptr); where,

  fdptr is the pointer to an integer array that will contain the two file descriptors for *reading* and *writing* the pipe
- Pipes use the same data structures as used by normal files, the user file descriptor table, and the in-core inode table; as a result, the interface to read/write files remains consistent and the processes do not need to know whether they are reading a normal file or a pipe.
- **The algorithm is given below:**

```
/*  Algorithm: pipe
 *  Input: none
 *  Output: read file descriptor
 *                    write file descriptor
 */

{
        assign new inode from pipe device (algorithm: ialloc);
        allocate a file table entry for reading and another for writing;
        initialize file table entries to point to new inode;
        allocate user file descriptor for reading, another for writing and point to respective file table
entries;
        set inode reference count to 2;
        initialize count of inode readers, writers to 1;
}
```

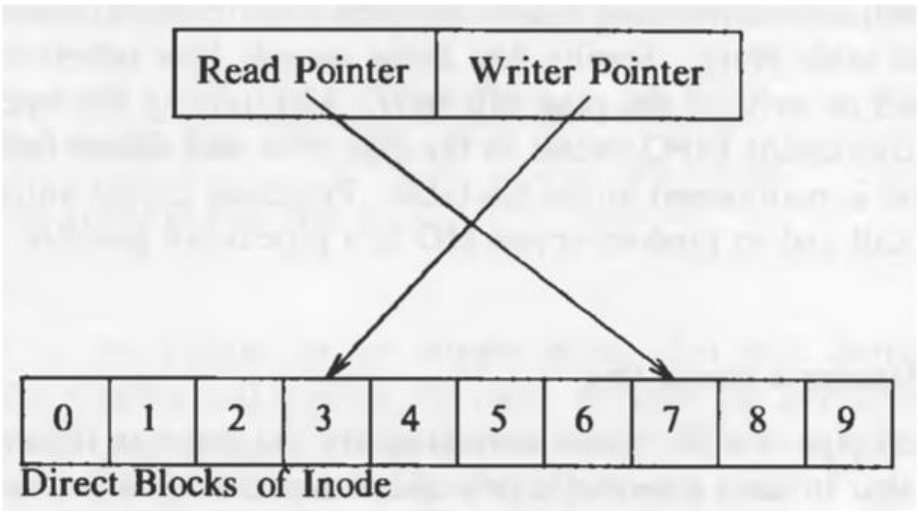
- A *pipe device* is just a file system from which the kernel can assign inodes and data blocks for pipes. System admins specify a pipe device during system configuration.
- One big difference between normal files and pipes is that the byte offset for normal files is in the file table entry, but for pipe files, it is in the inode. And the *lseek* system call cannot adjust the byte offsets in the inode; random access I/O to a pipe is not possible.

Opening a Named Pipe –

- Opening a named pipe is similar to opening a regular file, but the kernel increments the read or write counts in the inode to indicate the number of processes that have the named pipe open for reading or writing.
- A process that opens a named pipe for reading will sleep until another process opens the named pipe for writing, and vice versa.

Reading and Writing Pipes –

- Reading and writing pipes should be viewed as if processes write on one end of the pipe and read from the other end.
- The number of processes reading from a pipe does not necessarily equal the number of processes writing the pipe.
- The kernel manipulates the direct blocks of the inode as a circular queue. It maintains read and write pointers internally to preserve the FIFO order, as shown in the figure:



Closing Pipes –

- Closing pipes involves special processing by the kernel before releasing the inode. The kernel decrements the number of pipe readers or writers, depending on the type.
- If no reader or writer process accesses the pipe, the kernel frees all its data blocks and adjusts the inode to indicate that the pipe is empty.
- When releasing the inode of an ordinary pipe, the kernel frees the disk copy for reassignment.

Draw the diagram for file system tree before and after mount and describe mounting file systems.

- File systems are used to organize and store data on a computer's storage devices. A physical disk may have multiple logical partitions, each with its own file system.
- if each partition has a file system on it, it means that it will have its own boot block, super block, inode list, and data blocks.
- The mount system call allows you to attach a file system to a specified location on the current file system, making its data accessible as files instead of just disk blocks.
- Following is the syntax of mount system call –

  mount (special pathname, directory pathname, options); where,
  - **special pathname** is the name of the device file of the disk section whose file system is to be mounted.
  - The **directory pathname** is the path in existing file system where the new file system will be mounted.
  - The **options** indicate whether to mount in a "read-only" manner.
- The kernel has a mount table which has entries for each mounted file system. The contents of each entry are:
  - The device number of the file system.
  - Pointer to a buffer containing the super block of the mounted file system.
  - Pointer to a buffer containing the root inode of the mounted file system.
  - Pointer to a buffer containing the inode of the mount point on the existing file system.

The algorithm of the mount system call is given below:

```
/*  Algorithm: mount
 *  Input: file name of the block special file
 *       directory name of mount point
 *       options (read only)
 *  Output: none
 */
{
        if (not super user)
                return (error);
        get inode for block special file (algorithm: namei);
        make legality checks;
        get inode for "mounted on" directory name (algorithm: namei);
        if (not directory, or reference count > 1)
        {
                release inodes (algorithm: iput);
                return (error);
        }
        find empty slot in mount table;
        invoke block device driver open routine;
        get free buffer from buffer cache;
        read super block into free buffer;
        initialize super block fields;
        get root inode of mounted device (algorithm: iget), save in mount table;
        mark inode of "mounted on" directory as mount point;
        release special file inode (algorithm: iput);
        unlock inode of mount point directory;
}
```

- The diagram for file system tree before and after mount is as follows :

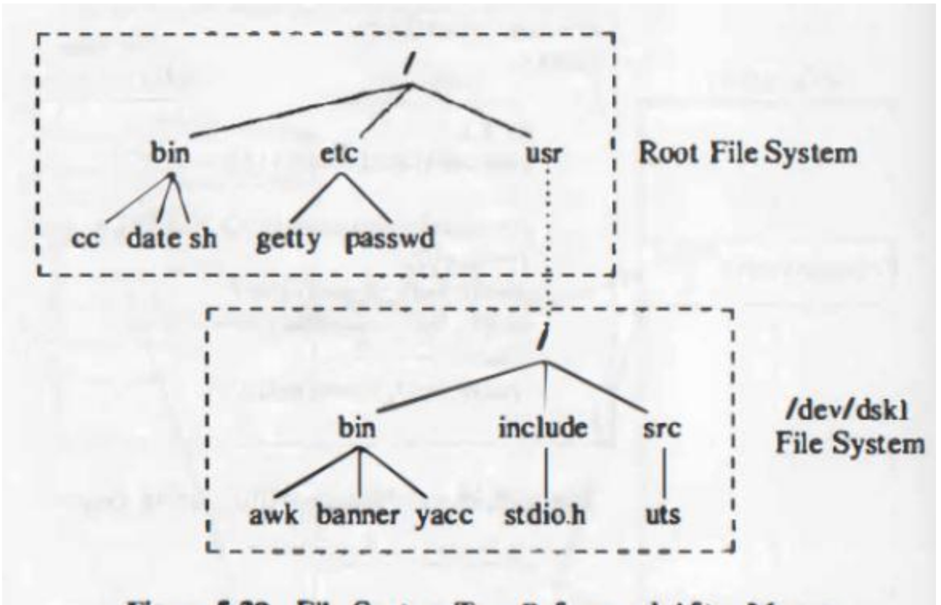


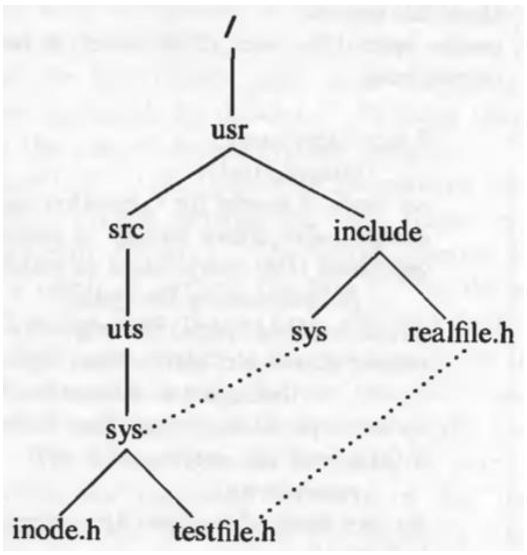
Figure 5.22. File System Tree Before and After Mount

- For example, if this system call is made: **mount ("/dev/dsk1", "/usr", 0);**
- The kernel attaches the file system on "/dev/disk1" on the directory "/usr". The root of the file system on "/dev/dsk1" will be accessed by "/usr". The processes can seamlessly access this newly mounted file system. Only the *link* system call checks if the file system is same for the files being linked.

Write a note on LINK system call.

- The LINK system call in an operating system is used to create a new directory entry for an existing file.
- It links a file to a new name in the directory structure, effectively creating a new hard link to the existing file. The LINK system call takes two arguments:
    - The source file name
    - The target file name.
- Following is the syntax of link system call –

  link (source file name, target file name);
- After linking the files, the kernel does not keep track of which file name was the original one. Therefore, no name is treated specially. For example, after linking with the following code:
  link ("/usr/src/uts/sys", "/usr/include/sys");
  link ("/usr/include/realfile.h", "/usr/src/uts/sys/testfile.h");
- the 3 pathnames refer to the same file: "/usr/src/uts/sys/testfile.h", "/usr/include/sys/testfile.h", "/usr/include/realfile"



- Only a superuser is allowed to link directories. This is done to avoid mistakes by arbitrary users. If a user *links* a directory to a child directory, a program which accesses that pathname will enter in a infinite loop.

The algorithm for link is given below:

```
/*  Algorithm: link
 *  Input: existing file name
 *                    new file name
 *  Output: none
 */

{
        get inode for existing file name (algorithm: namei);
        if (too many links on file or linking directory without super user permission)
        {
                release inode (algorithm: iput);
                return (error);
        }
        increment link count on inode;
        update disk copy of inode;
        unlock inode;
        get parent inode for directory to contain new file name (algorithm: namei);
        if (new file name already exists or existing file, new file on different file systems)
        {
                undo update done above;
                return (error);
        }
        create new directory entry in parent directory of new file name;
                include new file name, inode number of existing file name;
        release parent directory inode (algorithm: iput);
        release inode of existing file (algorithm: iput);
}
```

- **Restrictions**: The LINK system call has some restrictions. It is not allowed to link directories without superuser permission, as it can lead to infinite loops in programs that access the pathnames. It also checks for too many links on the file and for existing files with the same name.

Write note on File system abstraction.

- Unix supports multiple file system types. All the file system types are accessible through the usual system calls (*open*, *read*, *write*, *close*, etc.).
- To make this possible, the kernel has generic inodes, which point to specific inodes of a file system type. The generic inodes contain information like the device number, inode number, file name, file size, owner, reference count.
- The inode is the interface between the abstract file system and the specific file system.
- A generic in-core inode contains data that is independent of particular file systems, and points to a file-system-specific inode that contains file-system-specific data.
- The file-system-specific inode contains information such as access permissions and block layout, but the generic inode contains the device number, inode number, tile type, size, owner, and reference count
- Figure depicts the generic in-core inode table and two tables of file-system-specific inodes, one for System V file system structures and the other for a remote (network) inode.

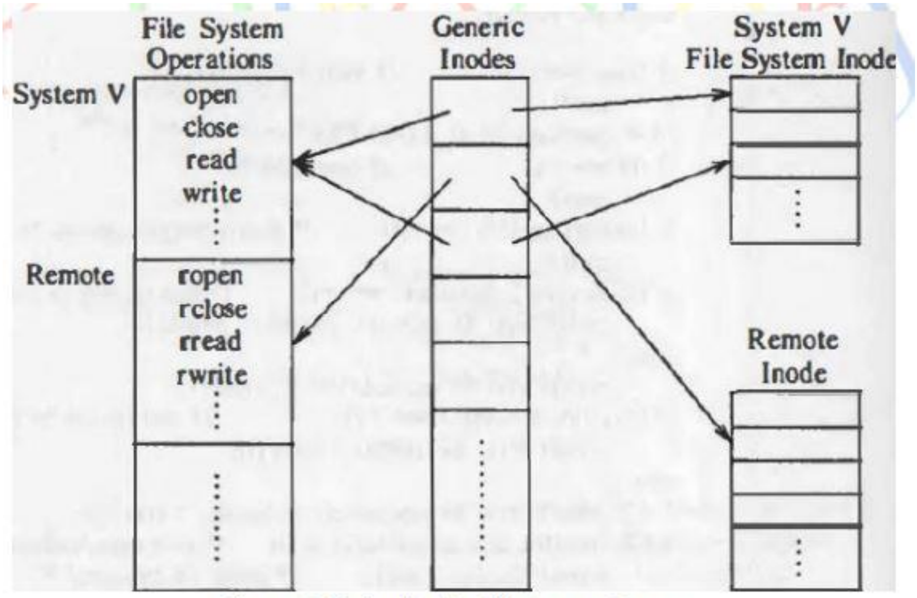


Figure 4.11: Inodes for File System Types

- The latter inode presumably contains enough information to identify a file on a remote system.
- Each file system type has a structure that contains the addresses of functions that perform abstract operations. When the kernel wants to access a file, it makes an indirect function call, based on the file system type and the operation.
- Some abstract operations are to open a file, close it, read or write data, return an inode for a file name component (like namei and iget), release an inode (like iput), and update an inode.
- Additionally, other operations include checking access permissions, setting file attributes (permissions), and mounting and unmounting file systems.