SYSTEM CALLS FOR THE FILE SYSTEM:

It starts with system calls for accessing existing files, such as open, read, write, /seek, and close, then presents system calls to create new files, namely, creat and mknod. It then examines the system calls that manipulate the inode or that maneuver through the file system: chdir, chroot, chown, chmod, stat, and fstat.

It investigates more advanced system calls: pipe and dup are important for the implementation of pipes in the shell; mount and unmount extend the file system tree visible to users; link and unlink change the structure of the file system hierarchy.

Then, it presents the notion of file system abstractions, allowing the support of various file systems as long as they conform to standard interfaces. Figure 4.1 shows the relationship between the system calls and the algorithms.

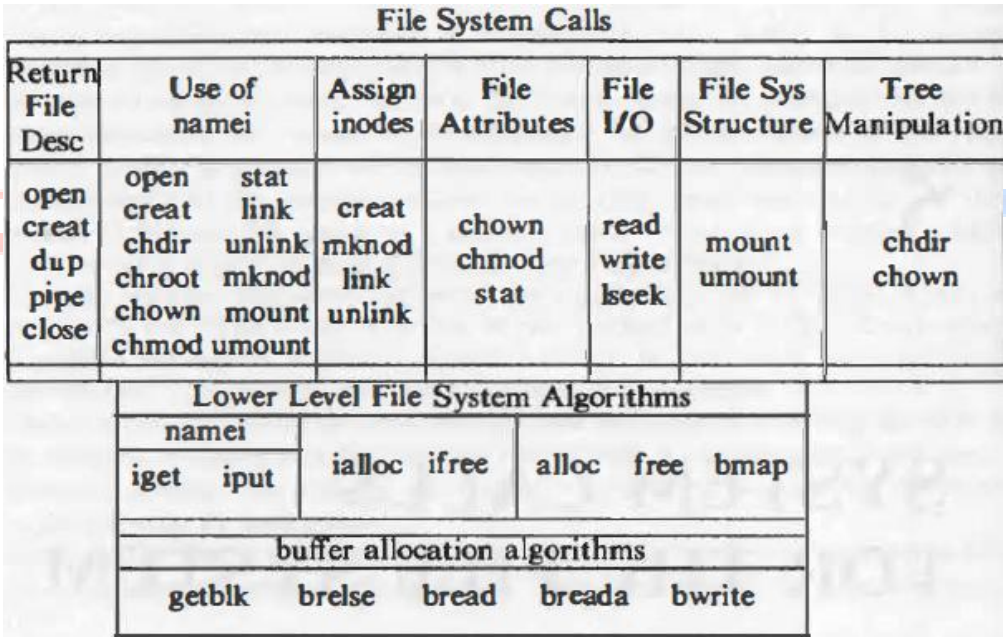

Figure 4.1: File System Calls and Relation to Other Algorithms

It classifies the system calls into several categories, although some system calls appear in more than one category:

- System calls that return file descriptors for use in other system calls
- System calls that use the ***namei*** algorithm to parse a path name
- System calls that assign and free inodes, using algorithms ***ialloc*** and ***ifree***
- System calls that set or change the attributes of a file

- System calls that do I/O to and from a process, using algorithms alloc, free, and the buffer allocation algorithms
- System calls that change the structure of the file system
- System calls that allow a process to change its view of the file system tree

OPEN:

The open system call is the first step a process must take to access the data in a file. The syntax for the open system call is

fd = open(pathname, flags, modes);

Where pathname is a file name, flags indicate the type of open (such as for reading or writing), and modes give the file permissions if the file is being created.

The open system call returns an integer called the user file descriptor. Other file operations, such as reading, writing, seeking, duplicating the file descriptor, setting file I/O parameters, determining file status, and closing the file, use the file descriptor that the open system call returns.

The kernel searches the file system for the file name parameter using algorithm namei (Figure 4.2).

```
algorithm open
inputs:  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);
}
```

Figure 4.2: Algorithm for Opening a File

Suppose a process executes the following code, opening the file "/etc/passwd" twice, once read-only and once write-only, and the file "local" once, for reading and writing.

fd1 = open("/etc/passwd", O_RDONLY);

fd2 = open("local", O_RDWR);

fd3 = open("/etc/passwd", O_WRONLY);

Figure 4.3 shows the relationship between the inode table, file table, and user file descriptor data structures.

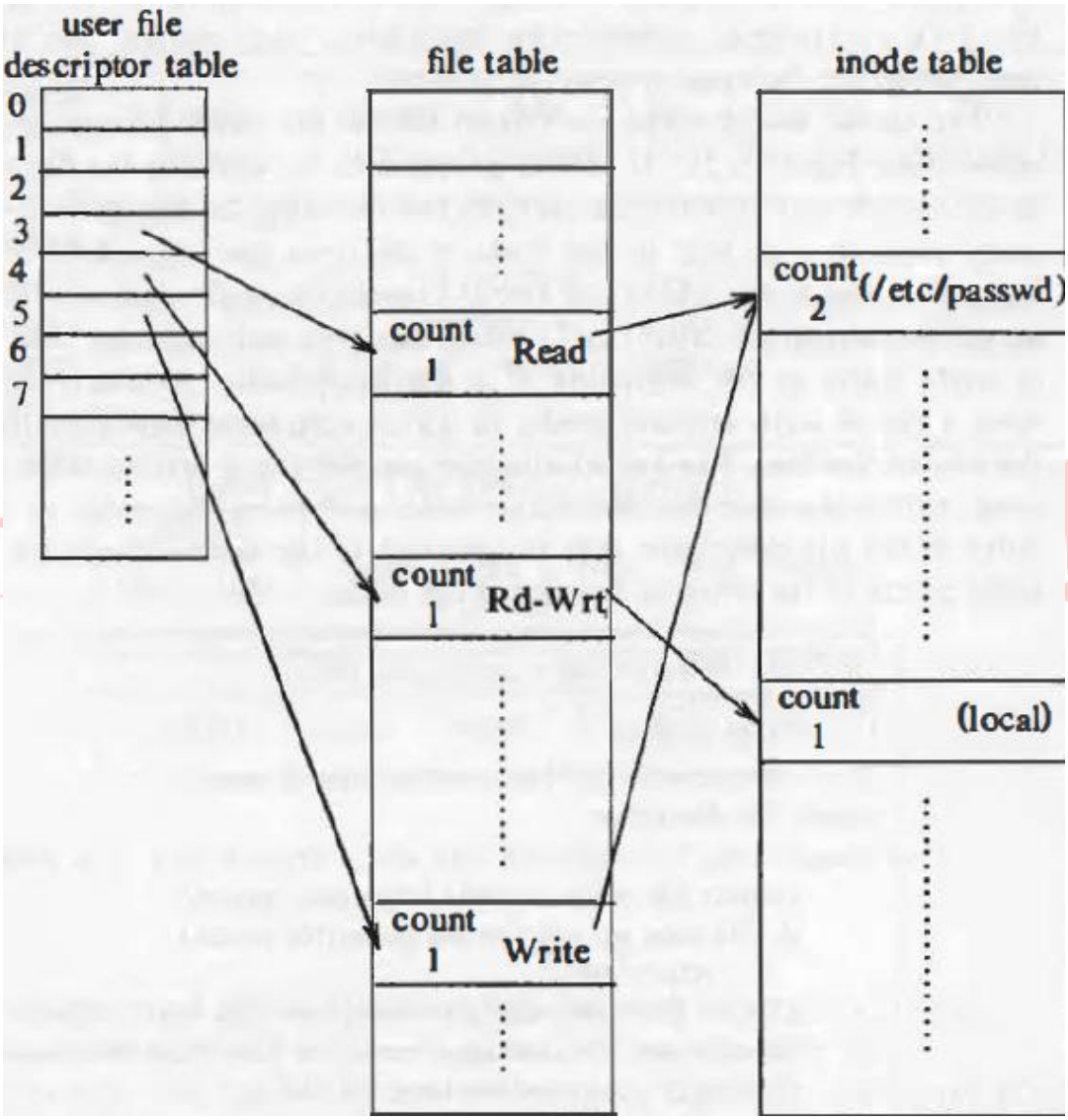

Figure 4.3: Data Structures after Open

READ:

The syntax of the read system call is:

number = read(fd, buffer, count);

Where fd is the file descriptor returned by open, buffer is the address of a data structure in the user process that will contain the read data on successful completion of the call, count is the number of bytes the user wants to read, and number is the number of bytes actually read. Figure 4.4 depicts the algorithm read for reading a regular file.

```
algorithm read
input:   user file descriptor
         address of buffer in user process
         number of bytes to read
output: count of bytes copied into user space
{
      get file table entry from user file descriptor;
      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)
                  /* trying to read end of file */
                  break;              /* out of loop */
            read block (algorithm breada if with read ahead, algorithm
                        bread otherwise);
            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);
}
```

Figure 4.4: Algorithm for Reading a File

WRITE:

The syntax for the write system call is:

number = write(fd, buffer, count);

Where the meaning of the variables fd, buffer, count, and number are the same as they are for the read system call. The algorithm for writing a regular file is similar to that for reading a regular file.

However, if the file does not contain a block that corresponds to the byte offset to be written, the kernel allocates a new block using algorithm alloc and assigns the block number to the correct position in the inode's table of contents.

If the byte offset is that of an indirect block, the kernel may have to allocate several blocks for use as indirect blocks and data blocks.

FILE AND RECORD LOCKING:

The original UNIX system developed by Thompson and Ritchie did not have an internal mechanism by which a process could insure exclusive access to a file. A locking mechanism was considered unnecessary because, as Ritchie notes, "we are not faced with large, single-file databases maintained by independent processes".

To make the UNIX system more attractive to commercial users with database applications, System V now contains file and record locking mechanisms. File locking is the capability to prevent other processes from reading or writing any part of an entire file, and record locking is the capability to prevent other processes from reading or writing particular records (parts of a file between particular byte offsets).

ADJUSTING THE POSITION OF FILE I/O- LSEEK:

The ordinary use of read and write system calls provides sequential access to a file, but processes can use the lseek system call to position the I/O and allow random access to a file.

The syntax for the system call is:

position = lseek(fd, offset, reference);

Where fd is the file descriptor identifying the file, offset is a byte offset, and reference indicates whether offset should be considered from the beginning of the file, from the current position of the read/write offset, or from the end of the file. The return value, position, is the byte offset where the next read or write will start.

CLOSE:

A process closes an open file when it no longer wants to access it. The syntax for the close system call is:

close(fd);

Where fd is the file descriptor for the open file. The kernel does the close operation by manipulating the file descriptor and the corresponding file table and inode table entries. If the reference count of the file table entry is greater than 1 because of dup or fork calls, then other user file descriptors reference the file table entry, as will be seen; the kernel decrements the count and the close completes.

When the close system call completes, the user file descriptor table entry is empty. Attempts by the process to use that file descriptor result in an error until the file descriptor is reassigned as a result of another system call.

FILE CREATION:

The open system call gives a process access to an existing file, but the creat system call creates a new file in the system. The syntax for the creat system call is:

fd = creat(pathname, modes);

Where the variables pathname, modes, and fd mean the same as they do in the open system call. If no such file previously existed, the kernel creates a new file with the specified name and permission modes; if the file already existed, the kernel truncates the file (releases all existing data blocks and sets the file size to 0) subject to suitable file access permissions. Figure 4.5 shows the algorithm for file creation.

```
algorithm creat
input:   file name
         permission settings
output: file descriptor
{
        get inode for file name (algorithm namei);
        if (file already exists)
        {
                if (not permitted access)
                {
                        release inode (algorithm iput);
                        return(error);
                }
        }
        else        /* file does not exist yet */
        {
                assign free inode from file system (algorithm ialloc);
                create new directory entry in parent directory: include
                        new file name and newly assigned inode number;
        }
        allocate file table entry for inode, initialize count;
        if (file did exist at time of create)
                free all file blocks (algorithm free);
        unlock(inode);
        return(user file descriptor);
}
```

Figure 4.5: Algorithm for Creating a File

CREATION OF SPECIAL FILES:

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, and dev specifies the major and minor device numbers for block and character special files.

Figure 4.6 depicts 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);
}
```

Figure 4.6: Algorithm for Making New Node

CHANGE DIRECTORY AND CHANGE ROOT:

When the system is first booted, process 0 makes the file system root its current directory during initialization. It executes the algorithm ***iget*** on the root inode, saves it in the ***u area*** as its current directory, and releases the inode lock.

When a new process is created via the fork system call, the new process inherits the current directory of the old process in its u area, and the kernel increments the inode reference count accordingly.

The algorithm chdir (Figure 4.7) changes the current directory of a process.

```
algorithm change directory
input:   new directory name
output: none
{
        get inode for new directory name (algorithm namei);
        if (inode not that of directory or process not permitted access to file)
        {
                release inode (algorithm iput);
                return (error);
        }
        unlock inode;
        release "old" current directory inode (algorithm iput);
        place new inode into current directory slot in u area;
}
```

Figure 4.7: Algorithm for Changing Current Directory

The syntax for the chdir system call is:

chdir(pathname);

Where pathname is the directory that becomes the new current directory of the process. A process usually uses the global file system root for all path names starting with "/". The kernel contains a global variable that points to the inode of the global root, allocated by iget when the system is booted.

Processes can change their notion of the file system root via the chroot system call. This is useful if a user wants to simulate the usual file system hierarchy and run processes there. Its syntax is:

chroot (pathname);

Where pathname is the directory that the kernel subsequently treats as the process's root directory. When executing the chroot system call, the kernel follows the same algorithm as for changing the current directory.

CHANGE OWNER AND CHANGE MODE:

Changing the owner or mode (access permissions) of a file are operations on the inode. The syntax of the calls is:

chown(pathname, owner, group)

chmod(pathname, mode)

To change the owner of a file, the kernel converts the file name to an inode using algorithm namei.

The process owner must be super user or match that of the file owner (a process cannot give away something that does not belong to it). The kernel then assigns the new owner and group to the file, clears the set user and set group flags, and releases the inode via algorithm ***iput***.

After the change of ownership, the old owner loses "owner" access rights to the file. To change the mode of a file, the kernel follows a similar procedure, changing the mode flags in the inode instead of the owner numbers.

STAT AND FSTAT:

The system calls stat and fstat allow processes to query the status of files, returning information such as the file type, file owner, access permissions, file size, number of links, inode number, and file access times. The syntax for the system calls is:

stat(pathname, statbuffer);

fstat(fd, statbuffer);

Where pathname i s a file name, fd is a file descriptor returned by a previous open call, and statbuffer is the address of a data structure in the user process that will contain the status information of the file on completion of the call. The system calls simply write the fields of the inode into statbuffer.

PIPES:

Pipes allow transfer of data between processes in a first-in-first-out manner (FIFO), and they also allow synchronization of process execution. Their implementation allows processes to communicate even though they do not know what processes are on the other end of the pipe.

The traditional implementation of pipes uses the file system for data storage. There are two kinds of pipes: named pipes and, for lack of a better term, unnamed pipes, which are identical except for the way that a process initially accesses them. Processes use the open system call for named pipes, but the pipe system call to create an unnamed pipe.

Afterwards, processes use the regular system calls for files, such as read, write, and close when manipulating pipes. Only related processes, descendants of a process that issued the pipe call, can share access to unnamed pipes.

In Figure 4.8 for example, if process B creates a pipe and then spawns processes D and E, the three processes share access to the pipe, but processes A and C do not.

However, all processes can access a named pipe regardless of their relationship, subject to the usual file permissions.

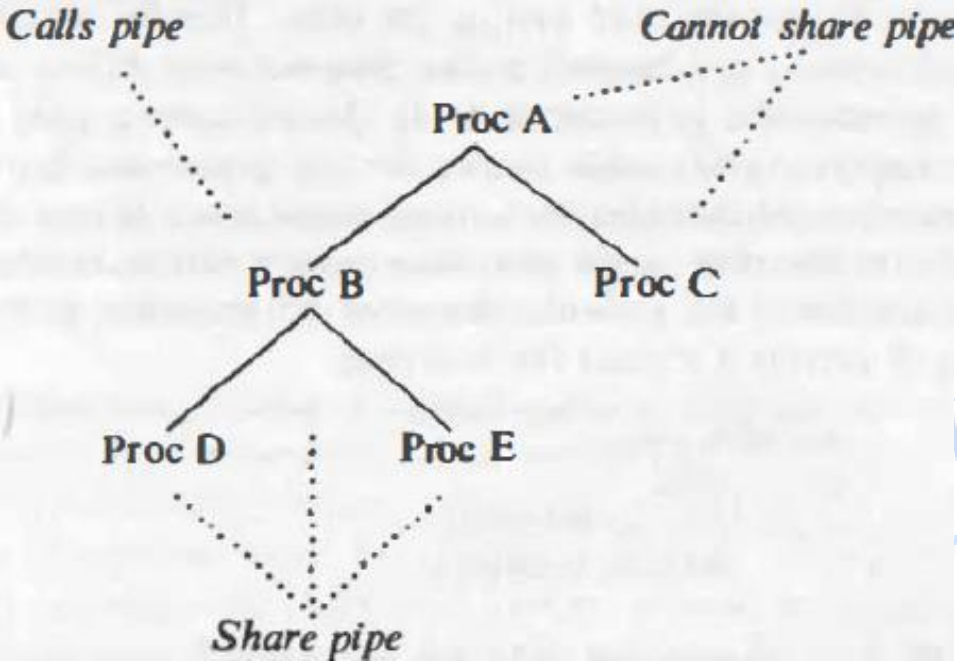

Figure 4.8: Process Tree and Sharing Pipes

Because unnamed pipes are more common, they will be presented first. The Pipe System Call: The syntax for creation of a pipe is

pipe(fdptr);

Where fdptr is the pointer to an integer array that will contain the two file descriptors for reading and writing the pipe; Figure 4.9 shows the algorithm for creating unnamed pipes.

The kernel assigns an inode for a pipe from a file system designated the pipe device using algorithm ialloc. A pipe device is just a file system from which the kernel can assign inodes and data blocks for pipes.

```
algorithm pipe
input:  none
output: read file descriptor
        write file descriptor
{
      assign new inode from pipe device (algorithm ialloc);
      allocate file table entry for reading, another for writing;
      initialize file table entries to point to new inode;
      allocate user file descriptor for reading, another for writing,
            initialize to point to respective file table entries;
      set inode reference count to 2;
      initialize count of inode readers, writers to 1;
}
```

Figure 4.9: Algorithm for Creation of (Unnamed) Pipes

Opening a Named Pipe: A named pipe is a file whose semantics are the same as those of an unnamed pipe, except that it has a directory entry and is accessed by a path name.

Processes open named pipes in the same way that they open regular files and, hence, processes that are not closely related can communicate. Named pipes permanently exist in the file system hierarchy (subject to their removal by the unlink system call), but unnamed pipes are transient: When all processes finish using the pipe, the kernel reclaims its inode.

The algorithm for opening a named pipe is identical to the algorithm for opening a regular file. However, before completing the system call, the kernel increments the read or write counts in the inode, indicating the number of processes that have the named pipe open for reading or writing.

Reading and Writing Pipes:

A pipe should be viewed as if processes write into one end of the pipe and read from the other end. As mentioned above, processes access data from a pipe in FIFO manner, meaning that the order that data is written into a pipe is the order that it is read from the pipe.

The number of processes reading from a pipe does not necessarily equal the number of processes writing the pipe. The kernel accesses the data for a pipe exactly as it accesses data for a regular file: It stores data on the pipe device and assigns blocks to the pipe as needed during write calls.

The difference between storage allocation for a pipe and a regular file is that a pipe uses only the direct blocks of the inode for greater efficiency, although this places a limit on how much data a pipe can hold at a time.

The kernel manipulates the direct blocks of the inode as a circular queue, maintaining read and write pointers internally to preserve the FIFO order (Figure 4.10).

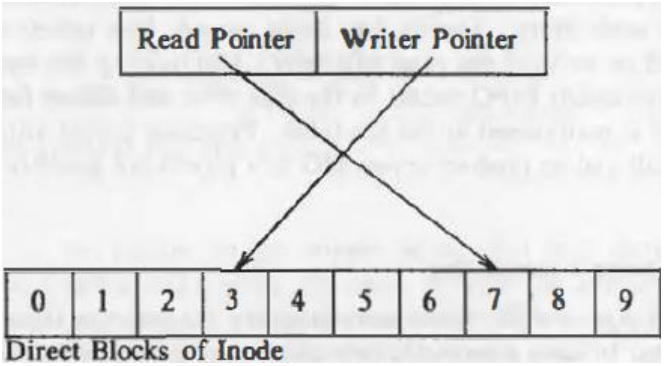

Figure 4.10: Logical View of Reading and Writing a Pipe

Closing Pipes: When closing a pipe, a process follows the same procedure it would follow for closing a regular file, except that the kernel does special processing before releasing the pipe's inode.

The kernel decrement the number of pipe readers or writers, according to the type of the file descriptor. If the count of writer processes drops to 0 and there are processes asleep waiting to read data from the pipe, the kernel awakens them, and they return from their read calls without reading any data If the count of reader processes drops to 0 and there are processes asleep waiting to write data to the pipe, the kernel awakens them and sends them a signal to indicate an error condition.

DUP:

The dup system call copies a file descriptor into the first free slot of the user file descriptor table, returning the new file descriptor to the user. It works for all file types. The syntax of the system call is

newfd = dup(fd);

Where fd is the file descriptor being duped and newfd is the new file descriptor that references the file. Because dup duplicates the file descriptor, it increments the count of the corresponding file table entry, which now has one more file descriptor entry that points to it.

Dup is perhaps an inelegant system call, because it assumes that the user knows that the system will return the lowest-numbered free entry in the user file descriptor table. However, it serves an important purpose in building sophisticated programs from simpler, building-block programs, as exemplified in the construction of shell pipelines.

MOUNTING AND UNMOUNTING FILE SYSTEMS:

A physical disk unit consists of several logical sections, partitioned by the disk driver, and each section has a device file name. Processes can access data in a section by opening the appropriate device file name and then reading and writing the "file" treating it as a sequence of disk blocks.

The mount system call connects the file system in a specified section of a disk to the existing file system hierarchy, and the umount system call disconnects a file system from the hierarchy. The mount system call thus allows users to access data in a disk section as a file system instead of a sequence of disk blocks.

The syntax for the mount system call is

mount(special pathname, directory pathname, options);

Where special pathname is the name of the device special file of the disk section containing the file system to be mounted, directory pathname is the directory in the existing hierarchy where the file system will be mounted (called the mount point), and options indicate whether the file system should be mounted "read-only".

The kernel has a mount table with entries for every mounted file system. Each mount table entry contains:

- a device number that identifies the mounted file system (this is the logical file system number mentioned previously);
- a pointer to a buffer containing the file system super block;
- a pointer to the root inode of the mounted file system;
- a pointer to the inode of the directory that is the mount point

Association of the mount point inode and the root inode of the mounted file system, set up during the mount system call, and allows the kernel to traverse the file system hierarchy gracefully, without special user knowledge.

Crossing Mount Points in File Path Names:

The two cases for crossing a mount point are: crossing from the mounted-on file system to the mounted file system (in the direction from the global system root towards a leaf node) and crossing from the mounted file system to the mounted-on file system.

Unmounting a File System:

The syntax for the umount system call is

umount(special filename);

Where special filename indicates the file system to be unmounted; when unmounting a file system the kernel accesses the inode of the device to be unmounted, retrieves the device number for the special file, releases the inode (algorithm iput), and finds the mount table entry whose device number equals that of the special file.

Before the kernel actually unmounts a file system, it makes sure that no files on that file system are still in use by searching the inode table for all files whose device number equals that of the file system being unmounted.

Active files have a positive reference count and include files that are the current directory of some process, files with shared text that are currently being executed and open files that have not been closed.

If any files from the file system are active, the umount call fails: if it were to succeed, the active files would be inaccessible. The buffer pool may still contain "delayed write" blocks that were not written to disk, so the kernel flushes them from the buffer pool.

The kernel removes shared text entries that are in the region table but not operational writes out all recently modified super blocks to disk, and updates the disk copy of all inodes that need updating.

LINK:

The link system call links a file to a new name in the file system directory structure, creating a new directory entry for an existing inode. The syntax for the link system call is

link(source file name, target file name);

Where source file name is the name of an existing file and target file name is the new (additional) name the file will have after completion of the link call.

The file system contains a path name for each link the file has, and processes can access the file by any of the path names.

The kernel does not know which name was the original file name, so no file name is treated specially.

UNLINK:

The unlink system call removes a directory entry for a file. The syntax for the unlink call is

unlink (pathname);

Where pathname identifies the name of the file to be unlinked from the directory hierarchy; if a process unlinks a given file, no file is accessible by that name until another directory entry with that name is created.

In the following code fragment, for example,

```
unlink("myfile");
fd = open("myfile", O_RDONLY);
```

The open call should fail, because the current directory no longer contains a file called myfile. If the file being unlinked is the last link of the file, the kernel eventually frees its data blocks. However, if the file had several links, it is still accessible by its other names.

File System Consistency:

The kernel orders its writes to disk to minimize file system corruption in event of system failure. For instance, when it removes a file name from its parent directory, it writes the directory synchronously to the disk - before it destroys the contents of the file and frees the inode.

If the system were to crash before the file contents were removed, damage to the file system would be minimal.

Race Conditions:

Race conditions abound in the unlink system call, particularly when unlinking directories. The rmdir command removes a directory after verifying that the directory contains no files (it reads the directory and checks that all directory entries have inode value 0).

But since rmdir runs at user level, the actions of verifying that a directory is empty and removing the directory are not atomic; the system could do a context switch between execution of the read and unlink system calls.

Hence, another process could creat a file in the directory after rmdir determined that the directory was empty. Users can prevent this situation only by use of file and record locking.

FILE SYSTEM ABSTRACTIONS:

Weinberger introduced file system types to support his network file system. File system types allow the kernel to support multiple file systems simultaneously, such as network file systems or even file systems of other operating systems. Processes use the usual UNIX system calls to access files, and the kernel maps a generic set of file operations into operations specific to each file system type.

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. Other data that is file-system-specific includes the super block and directory structures.

Figure 4.11 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.

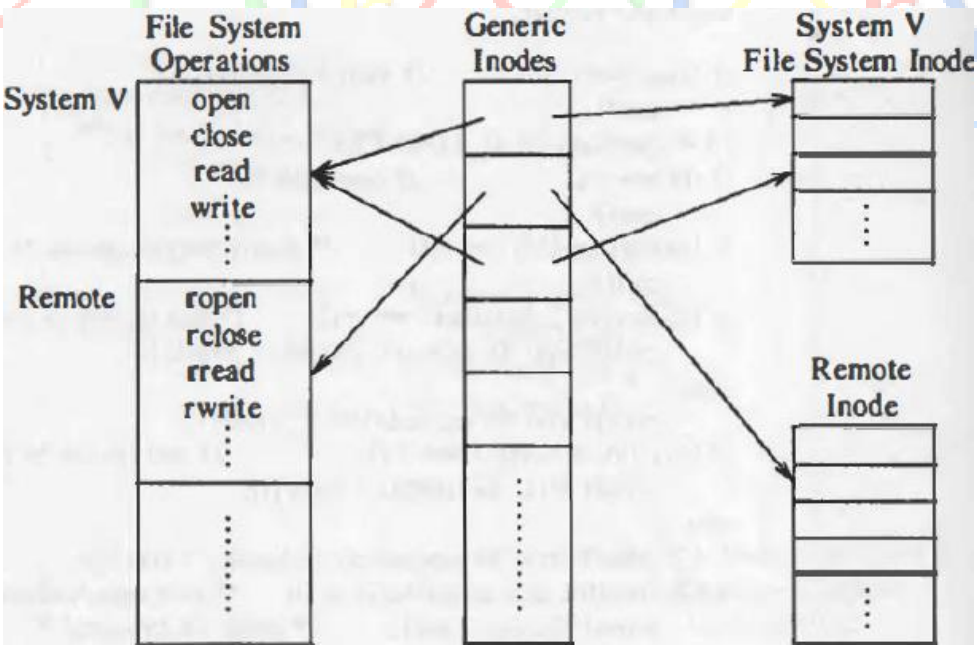

Figure 4.11: Inodes for File System Types

The latter inode presumably contains enough information to identify a file on a remote system. A file system may not have an inode-like structure; but the file-system-specific code manufactures an object that satisfies UNIX file system semantics and allocates its "inode" when the kernel allocates a generic inode.

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), update an inode, check access permissions, set file attributes (permissions), and mount and unmount file systems.

FILE SYSTEM MAINTENANCE:

The kernel maintains consistency of the file system during normal operation. However, extraordinary circumstances such as a power failure may cause a system crash that leaves a file system in an inconsistent state: most of the data in the file system is acceptable for use, but some inconsistencies exist.

The command ***fsck*** checks for such inconsistencies and repairs the file system if necessary. It accesses the file system by its block or raw interface and bypasses the regular file access methods.

A disk block may belong to more than one inode or to the list of free blocks and an inode.

When a file system is originally set up, all disk blocks are on the free list. When a disk block is assigned for use, the kernel removes it from the free list and assigns it to an inode.

The kernel may not reassign the disk block to another inode until the disk block has been returned to the free list. Therefore, a disk block is either on the free list or assigned to a single inode.

Consider the possibilities if the kernel freed a disk block in a file, returning the block number to the in-core copy of the super block, and allocated the disk block to a new file.

If the kernel wrote the inode and blocks of the new file to disk but crashed before updating the inode of the old file to disk, the two inodes would address the same disk block number.

Similarly, if the kernel wrote the super block and its free list to disk and crashed before writing the old inode out, the disk block would appear on the free list and in the old inode.