4

INTERNAL REPRESENTATION OF FILES

As observed in Chapter 2, every file on a UNIX system has a unique inode. The inode contains the information necessary for a process to access a file, such as file ownership, access rights, file size, and location of the file's data in the file system. Processes access files by a well defined set of system calls and specify a file by a character string that is the path name. Each path name uniquely specifies a file, and the kernel converts the path name to the file's inode.

This chapter describes the internal structure of files in the UNIX system, and the next chapter describes the system call interface to files. Section 4.1 examines the inode and how the kernel manipulates it, and Section 4.2 examines the internal structure of regular files and how the kernel reads and writes their data. Section 4.3 investigates the structure of directories, the files that allow the kernel to organize the file system as a hierarchy of files, and Section 4.4 presents the algorithm for converting user file names to inodes. Section 4.5 gives the structure of the super block, and Sections 4.6 and 4.7 present the algorithms for assignment of disk inodes and disk blocks to files. Finally, Section 4.8 talks about other file types in the system, namely, pipes and device files.

The algorithms described in this chapter occupy the layer above the buffer cache algorithms explained in the last chapter (Figure 4.1). The algorithm *iget* returns a previously identified inode, possibly reading it from disk via the buffer cache, and the algorithm *iput* releases the inode. The algorithm *bmap* sets kernel parameters for accessing a file. The algorithm *namei* converts a user-level path

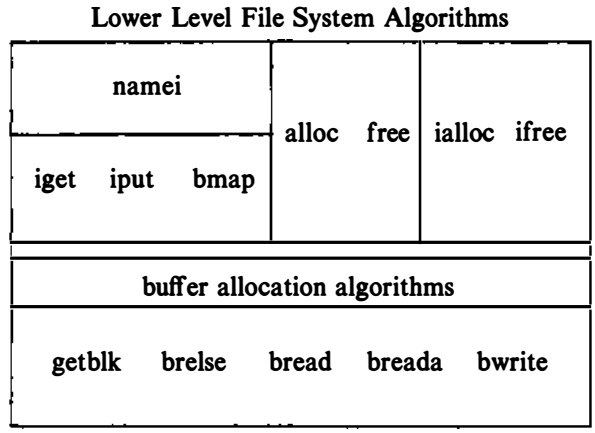

Figure 4.1. File System Algorithms

name to an inode, using the algorithms *iget*, *iput*, and *bmap*. Algorithms *alloc* and *free* allocate and free disk blocks for files, and algorithms *ialloc* and *ifree* assign and free inodes for files.

4.1 INODES

4.1.1 Definition

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

- File owner identifier. Ownership is divided between an individual owner and a "group" owner and defines the set of users who have access rights to a file. The superuser has access rights to all files in the system.
- File type. Files may be of type regular, directory, character or block special, or FIFO (pipes).
- File access permissions. The system protects files according to three classes: the owner and the group owner of the file, and other users; each class has access rights to read, write and execute the file, which can be set individually. Because directories cannot be executed, execution permission for a directory gives the right to search the directory for a file name.
- File access times, giving the time the file was last modified, when it was last accessed, and when the inode was last modified.

- Number of links to the file, representing the number of names the file has in the directory hierarchy. Chapter 5 explains file links in detail.
- Table of contents for the disk addresses of data in a file. Although users treat the data in a file as a logical stream of bytes, the kernel saves the data in discontiguous disk blocks. The inode identifies the disk blocks that contain the file's data.
- File size. Data in a file is addressable by the number of bytes from the beginning of the file, starting from byte offset 0, and the file size is 1 greater than the highest byte offset of data in the file. For example, if a user creates a file and writes only 1 byte of data at byte offset 1000 in the file, the size of the file is 1001 bytes.

The inode does not specify the path name(s) that access the file.

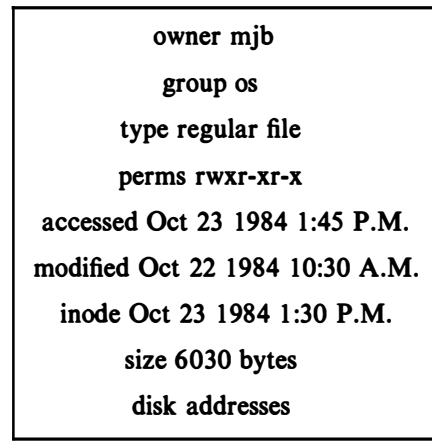

Figure 4.2. Sample Disk Inode

Figure 4.2 shows the disk inode of a sample file. This inode is that of a regular file owned by "mjb," which contains 6030 bytes. The system permits "mjb" to read, write, or execute the file; members of the group "os" and all other users can only read or execute the file, not write it. The last time anyone read the file was on October 23, 1984, at 1:45 in the afternoon, and the last time anyone wrote the file was on October 22, 1984, at 10:30 in the morning. The inode was last changed on October 23, 1984, at 1:30 in the afternoon, although the data in the file was not written at that time. The kernel encodes the above information in the inode. Note the distinction between writing the contents of an inode to disk and writing the contents of a file to disk. The contents of a file change only when *writ*ing it. The contents of an inode change when changing the contents of a file or when changing its owner, permission, or link settings. Changing the contents of a

file automatically implies a change to the inode, but changing the inode does not imply that the contents of the file change.

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

- The status of the in-core inode, indicating whether
  - the inode is locked,
  - a process is waiting for the inode to become unlocked,
  - the in-core representation of the inode differs from the disk copy as a result of a change to the data in the inode,
  - the in-core representation of the file differs from the disk copy as a result of a change to the file data,
  - the file is a mount point (Section 5.15).
- The logical device number of the file system that contains the file.
- The inode number. Since inodes are stored in a linear array on disk (recall Section 2.2.1), the kernel identifies the number of a disk inode by its position in the array. The disk inode does not need this field.
- Pointers to other in-core inodes. The kernel links inodes on hash queues and on a free list in the same way that it links buffers on buffer hash queues and on the buffer free list. A hash queue is identified according to the inode's logical device number and inode number. The kernel can contain at most one in-core copy of a disk inode, but inodes can be simultaneously on a hash queue and on the free list.
- A reference count, indicating the number of instances of the file that are active (such as when *open*ed).

Many fields in the in-core inode are analogous to fields in the buffer header, and the management of inodes is similar to the management of buffers. The inode lock, when set, prevents other processes from accessing the inode; other processes set a flag in the inode when attempting to access it to indicate that they should be awakened when the lock is released. The kernel sets other flags to indicate discrepancies between the disk inode and the in-core copy. When the kernel needs to record changes to the file or to the inode, it writes the in-core copy of the inode to disk after examining these flags.

The most striking difference between an in-core inode and a buffer header is the in-core reference count, which counts the number of active instances of the file. An inode is active when a process allocates it, such as when *open*ing a file. An inode is on the free list only if its reference count is 0, meaning that the kernel can reallocate the in-core inode to another disk inode. The free list of inodes thus serves as a cache of inactive inodes: If a process attempts to access a file whose inode is not currently in the in-core inode pool, the kernel reallocates an in-core inode from the free list for its use. On the other hand, a buffer has no reference count; it is on the free list if and only if it is unlocked.

```
algorithm iget
input:  file system inode number
output: locked inode
{
        while (not done)
        {
                if (inode in inode cache)
                {
                        if (inode locked)
                        {
                                sleep (event inode becomes unlocked);
                                continue;       /* loop back to while */
                        }
                        /* special processing for mount points (Chapter 5) */
                        if (inode on inode free list)
                                remove from free list;
                        increment inode reference count;
                        return (inode);
                }

                /* inode not in inode cache */
                if (no inodes on free list)
                        return(error);
                remove new inode from free list;
                reset inode number and file system;
                remove inode from old hash queue, place on new one;
                read inode from disk (algorithm bread);
                initialize inode (e.g. reference count to 1);
                return(inode);
        }
}
```

Figure 4.3. Algorithm for Allocation of In-Core Inodes

4.1.2 Accessing Inodes

The kernel identifies particular inodes by their file system and inode number and allocates in-core inodes at the request of higher-level algorithms. The algorithm *iget* allocates an in-core copy of an inode (Figure 4.3); it is almost identical to the algorithm *getblk* for finding a disk block in the buffer cache. The kernel maps the device number and inode number into a hash queue and searches the queue for the inode. If it cannot find the inode, it allocates one from the free list and locks it. The kernel then prepares to read the disk copy of the newly accessed inode into the in-core copy. It already knows the inode number and logical device and computes the logical disk block that contains the inode according to how many disk inodes fit into a disk block. The computation follows the formula

block num = ((inode number − 1) / number of inodes per block) +
start block of inode list

where the division operation returns the integer part of the quotient. For example, assuming that block 2 is the beginning of the inode list and that there are 8 inodes per block, then inode number 8 is in disk block 2, and inode number 9 is in disk block 3. If there are 16 inodes in a disk block, then inode numbers 8 and 9 are in disk block 2, and inode number 17 is the first inode in disk block 3.

When the kernel knows the device and disk block number, it reads the block using the algorithm *bread* (Chapter 2), then uses the following formula to compute the byte offset of the inode in the block:

((inode number − 1) modulo (number of inodes per block)) * size of disk inode

For example, if each disk inode occupies 64 bytes and there are 8 inodes per disk block, then inode number 8 starts at byte offset 448 in the disk block. The kernel removes the in-core inode from the free list, places it on the correct hash queue, and sets its in-core reference count to 1. It copies the file type, owner fields, permission settings, link count, file size, and the table of contents from the disk inode to the in-core inode, and returns a locked inode.

The kernel manipulates the inode lock and reference count independently. The lock is set during execution of a system call to prevent other processes from accessing the inode while it is in use (and possibly inconsistent). The kernel releases the lock at the conclusion of the system call: an inode is never locked across system calls. The kernel increments the reference count for every active reference to a file. For example, Section 5.1 will show that it increments the inode reference count when a process *open*s a file. It decrements the reference count only when the reference becomes inactive, for example, when a process *close*s a file. The reference count thus remains set across multiple system calls. The lock is free between system calls to allow processes to share simultaneous access to a file; the reference count remains set between system calls to prevent the kernel from reallocating an active in-core inode. Thus, the kernel can lock and unlock an allocated inode independent of the value of the reference count. System calls other than *open* allocate and release inodes, as will be seen in Chapter 5.

Returning to algorithm *iget*, if the kernel attempts to take an inode from the free list but finds the free list empty, it reports an error. This is different from the philosophy the kernel follows for disk buffers, where a process sleeps until a buffer becomes free: Processes have control over the allocation of inodes at user level via execution of *open* and *close* system calls, and consequently the kernel cannot guarantee when an inode will become available. Therefore, a process that goes to sleep waiting for a free inode to become available may never wake up. Rather than leave such a process "hanging," the kernel fails the system call. However, processes do not have such control over buffers: Because a process cannot keep a buffer locked across system calls, the kernel can guarantee that a buffer will become free soon, and a process therefore sleeps until one is available.

The preceding paragraphs cover the case where the kernel allocated an inode that was not in the inode cache. If the inode is in the cache, the process (A) would find it on its hash queue and check if the inode was currently locked by another process (B). If the inode is locked, process A sleeps, setting a flag in the in-core inode to indicate that it is waiting for the inode to become free. When process B later unlocks the inode, it awakens all processes (including process A) waiting for the inode to become free. When process A is finally able to use the inode, it locks the inode so that other processes cannot allocate it. If the reference count was previously 0, the inode also appears on the free list, so the kernel removes it from there: the inode is no longer free. The kernel increments the inode reference count and returns a locked inode.

To summarize, the *iget* algorithm is used toward the beginning of system calls when a process first accesses a file. The algorithm returns a locked inode structure with reference count 1 greater than it had previously been. The in-core inode contains up-to-date information on the state of the file. The kernel unlocks the inode before returning from the system call so that other system calls can access the inode if they wish. Chapter 5 treats these cases in greater detail.

```
algorithm iput              /* release (put) access to in-core inode */
input:  pointer to in-core inode
output: none
{
        lock inode if not already locked;
        decrement inode reference count;
        if (reference count == 0)
        {
                if (inode link count == 0)
                {
                        free disk blocks for file (algorithm free, section 4.7);
                        set file type to 0;
                        free inode (algorithm ifree, section 4.6);
                }
                if (file accessed or inode changed or file changed)
                        update disk inode;
                put inode on free list;
        }
        release inode lock;
}
```

Figure 4.4. Releasing an Inode

4.1.3 Releasing Inodes

When the kernel releases an inode (algorithm *iput*, Figure 4.4), it decrements its in-core reference count. If the count drops to 0, the kernel writes the inode to disk if the in-core copy differs from the disk copy. They differ if the file data has changed, if the file access time has changed, or if the file owner or access permissions have changed. The kernel places the inode on the free list of inodes, effectively caching the inode in case it is needed again soon. The kernel may also release all data blocks associated with the file and free the inode if the number of links to the file is 0.

4.2 STRUCTURE OF A REGULAR FILE

As mentioned above, the inode contains the table of contents to locate a file's data on disk. Since each block on a disk is addressable by number, the table of contents consists of a set of disk block numbers. If the data in a file were stored in a contiguous section of the disk (that is, the file occupied a linear sequence of disk blocks), then storing the start block address and the file size in the inode would suffice to access all the data in the file. However, such an allocation strategy would not allow for simple expansion and contraction of files in the file system without running the risk of fragmenting free storage area on the disk. Furthermore, the kernel would have to allocate and reserve contiguous space in the file system before allowing operations that would increase the file size.

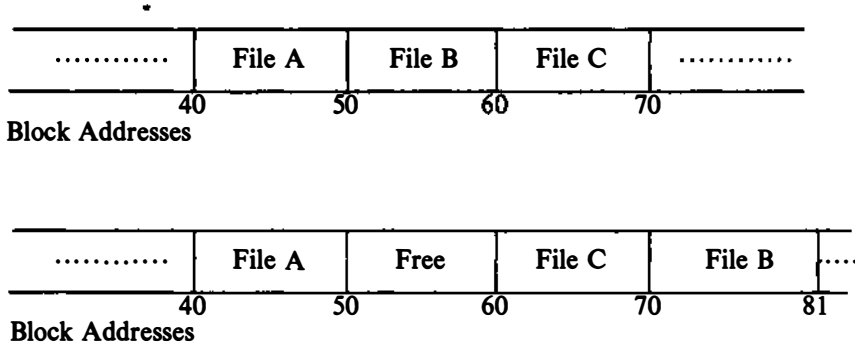

Figure 4.5. Allocation of Contiguous Files and Fragmentation of Free Space

For example, suppose a user creates three files, A, B and C, each consisting of 10 disk blocks of storage, and suppose the system allocated storage for the three files contiguously. If the user then wishes to add 5 blocks of data to the middle file, B, the kernel would have to copy file B to a place in the file system that had room for 15 blocks of storage. Aside from the expense of such an operation, the disk

blocks previously occupied by file B's data would be unusable except for files smaller than 10 blocks (Figure 4.5). The kernel could minimize fragmentation of storage space by periodically running garbage collection procedures to compact available storage, but that would place an added drain on processing power.

For greater flexibility, the kernel allocates file space one block at a time and allows the data in a file to be spread throughout the file system. But this allocation scheme complicates the task of locating the data. The table of contents could consist of a list of block numbers such that the blocks contain the data belonging to the file, but simple calculations show that a linear list of file blocks in the inode is difficult to manage. If a logical block contains 1K bytes, then a file consisting of 10K bytes would require an index of 10 block numbers, but a file containing 100K bytes would require an index of 100 block numbers. Either the size of the inode would vary according to the size of the file, or a relatively low limit would have to be placed on the size of a file.

To keep the inode structure small yet still allow large files, the table of contents of disk blocks conforms to that shown in Figure 4.6. The System V UNIX system runs with 13 entries in the inode table of contents, but the principles are independent of the number of entries. The blocks marked "direct" in the figure contain the numbers of disk blocks that contain real data. The block marked "single indirect" refers to a block that contains a list of direct block numbers. To access the data via the indirect block, the kernel must read the indirect block, find the appropriate direct block entry, and then read the direct block to find the data. The block marked "double indirect" contains a list of indirect block numbers, and the block marked "triple indirect" contains a list of double indirect block numbers.

In principle, the method could be extended to support "quadruple indirect blocks," "quintuple indirect blocks," and so on, but the current structure has sufficed in practice. Assume that a logical block on the file system holds 1K bytes and that a block number is addressable by a 32 bit (4 byte) integer. Then a block can hold up to 256 block numbers. The maximum number of bytes that could be held in a file is calculated (Figure 4.7) at well over 16 gigabytes, using 10 direct blocks and 1 indirect, 1 double indirect, and 1 triple indirect block in the inode. Given that the file size field in the inode is 32 bits, the size of a file is effectively limited to 4 gigabytes (2^{32}).

Processes access data in a file by byte offset. They work in terms of byte counts and view a file as a stream of bytes starting at byte address 0 and going up to the size of the file. The kernel converts the user view of bytes into a view of blocks: The file starts at logical block 0 and continues to a logical block number corresponding to the file size. The kernel accesses the inode and converts the logical file block into the appropriate disk block. Figure 4.8 gives the algorithm *bmap* for converting a file byte offset into a physical disk block.

Consider the block layout for the file in Figure 4.9 and assume that a disk block contains 1024 bytes. If a process wants to access byte offset 9000, the kernel calculates that the byte is in direct block 8 in the file (counting from 0). It then accesses block number 367; the 808th byte in that block (starting from 0) is byte

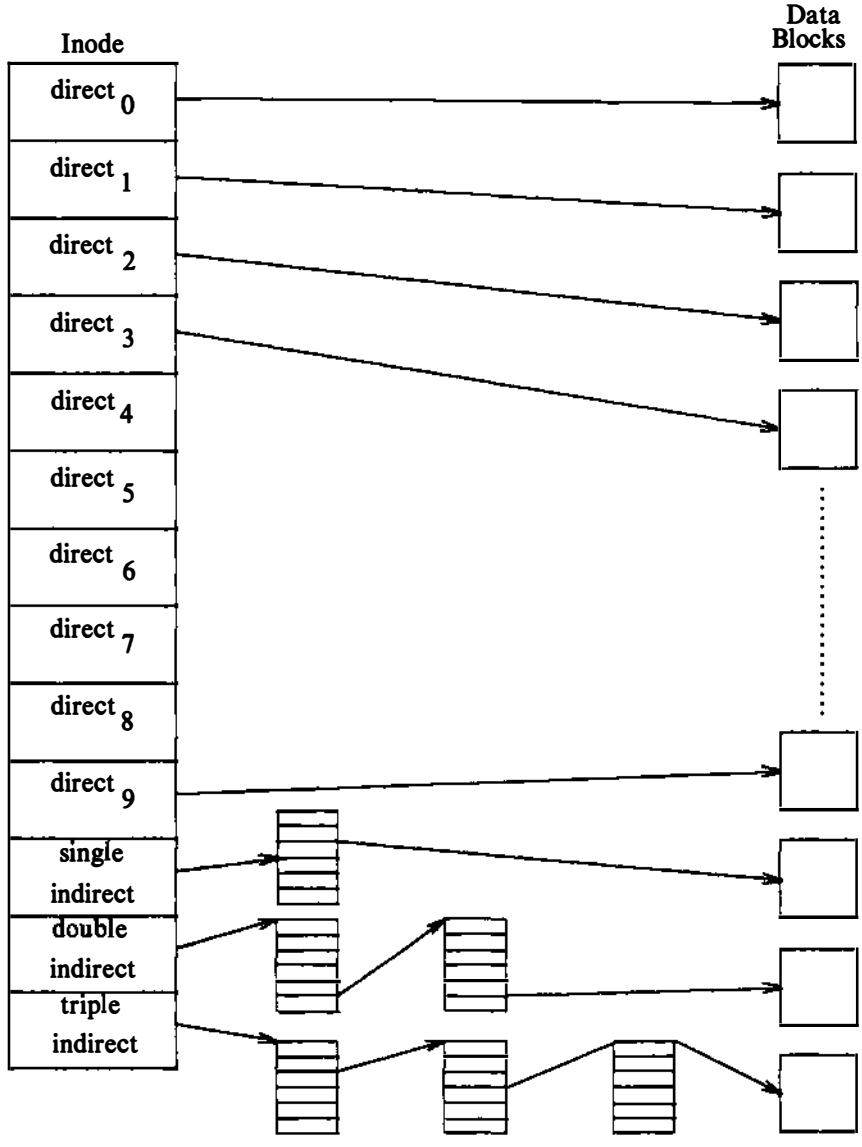

Figure 4.6. Direct and Indirect Blocks in Inode

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

Figure 4.7. Byte Capacity of a File — 1K Bytes Per Block

```
algorithm bmap    /* block map of logical file byte offset to file system block */
input:  (1) inode
        (2) byte offset
output: (1) block number in file system
        (2) byte offset into block
        (3) bytes of I/O in block
        (4) read ahead block number
{
      calculate logical block number in file from byte offset;
      calculate start byte in block for I/O;              /* output 2 */
      calculate number of bytes to copy to user;          /* output 3 */
      check if read-ahead applicable, mark inode;         /* output 4 */
      determine level of indirection;
      while (not at necessary level of indirection)
      {
            calculate index into inode or indirect block from
                        logical block number in file;
            get disk block number from inode or indirect block;
            release buffer from previous disk read, if any (algorithm brelse);
            if (no more levels of indirection)
                  return (block number);
            read indirect disk block (algorithm bread);
            adjust logical block number in file according to level of indirection;
      }
}
```

Figure 4.8. Conversion of Byte Offset to Block Number in File System

9000 in the file. If a process wants to access byte offset 350,000 in the file, it must access a double indirect block, number 9156 in the figure. Since an indirect block has room for 256 block numbers, the first byte accessed via the double indirect block is byte number 272,384 (256K + 10K); byte number 350,000 in a file is therefore byte number 77,616 of the double indirect block. Since each single indirect block accesses 256K bytes, byte number 350,000 must be in the 0th single indirect block of the double indirect block — block number 331. Since each direct block in a single indirect block contains 1K bytes, byte number 77,616 of a single

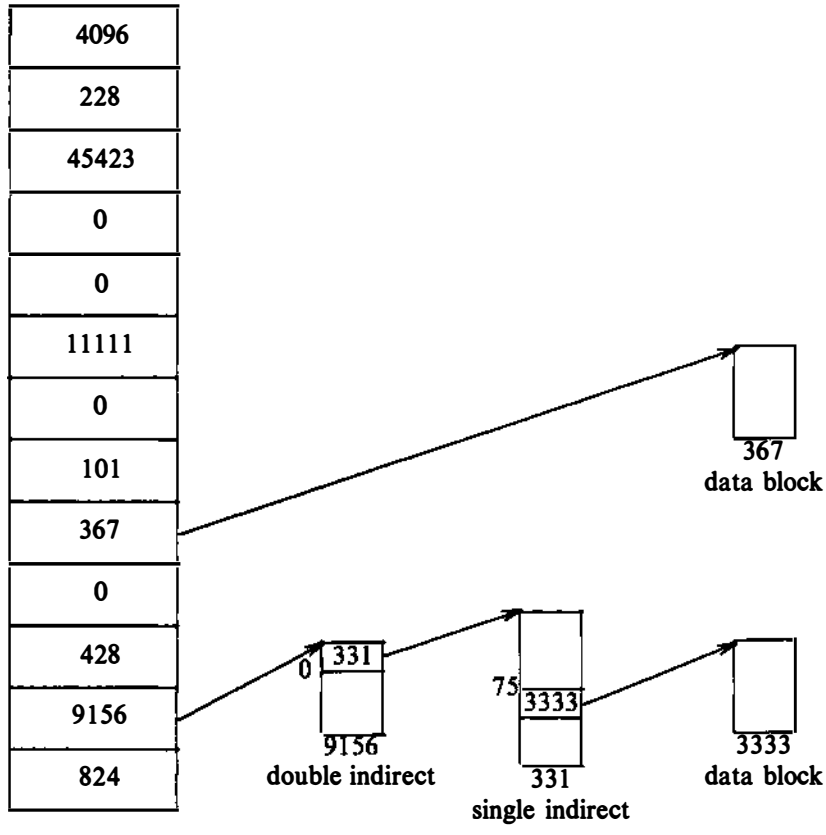

Figure 4.9. Block Layout of a Sample File and its Inode

indirect block is in the 75th direct block in the single indirect block — block number 3333. Finally, byte number 350,000 in the file is at byte number 816 in block 3333.

Examining Figure 4.9 more closely, several block entries in the inode are 0, meaning that the logical block entries contain no data. This happens if no process ever wrote data into the file at any byte offsets corresponding to those blocks and hence the block numbers remain at their initial value, 0. No disk space is wasted for such blocks. Processes can cause such a block layout in a file by using the *lseek* and *write* system calls, as described in the next chapter. The next chapter also describes how the kernel takes care of *read* system calls that access such blocks.

The conversion of a large byte offset, particularly one that is referenced via the triple indirect block, is an arduous procedure that could require the kernel to access three disk blocks in addition to the inode and data block. Even if the kernel finds

the blocks in the buffer cache, the operation is still expensive, because the kernel must make multiple requests of the buffer cache and may have to sleep awaiting locked buffers. How effective is the algorithm in practice? That depends on how the system is used and whether the user community and job mix are such that the kernel accesses large files or small files more frequently. It has been observed [Mullender 84], however, that most files on UNIX systems contain less than 10K bytes, and many contain less than 1K bytes![1] Since 10K bytes of a file are stored in direct blocks, most file data can be accessed with one disk access. So in spite of the fact that accessing large files is an expensive operation, accessing common-sized files is fast.

Two extensions to the inode structure just described attempt to take advantage of file size characteristics. A major principle in the 4.2 BSD file system implementation [McKusick 84] is that the more data the kernel can access on the disk in a single operation, the faster file access becomes. That argues for having larger logical disk blocks, and the Berkeley implementation allows logical disk blocks of 4K or 8K bytes. But having larger block sizes on disk increases block fragmentation, leaving large portions of disk space unused. For instance, if the logical block size is 8K bytes, then a file of size 12K bytes uses 1 complete block and half of a second block. The other half of the second block (4K bytes) is wasted; no other file can use the space for data storage. If the sizes of files are such that the number of bytes in the last block of a file is uniformly distributed, then the average wasted space is half a block per file; the amount of wasted disk space can be as high as 45% for a file system with logical blocks of size 4K bytes [McKusick 84]. The Berkeley implementation remedies the situation by allocating a block *fragment* to contain the last data in a file. One disk block can contain fragments belonging to several files. An exercise in Chapter 5 explores some details of the implementation.

The second extension to the classic inode structure described here is to store file data in the inode (see [Mullender 84]). By expanding the inode to occupy an entire disk block, a small portion of the block can be used for the inode structures and the remainder of the block can store the entire file, in many cases, or the end of a file otherwise. The main advantage is that only one disk access is necessary to get the inode and its data if the file fits in the inode block.

1. For a sample of 19,978 files, Mullender and Tannenbaum say that approximately 85% of the files were smaller than 8K bytes and that 48% were smaller than 1K bytes. Although these percentages will vary from one installation to the next, they are representative of many UNIX systems.

4.3 DIRECTORIES

Recall from Chapter 1 that directories are the files that give the file system its hierarchical structure; they play an important role in conversion of a file name to an inode number. A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file contained in the directory. A path name is a null terminated character string divided into separate components by the slash ("/") character. Each component except the last must be the name of a directory, but the last component may be a non-directory file. UNIX System V restricts component names to a maximum of 14 characters; with a 2 byte entry for the inode number, the size of a directory entry is 16 bytes.

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

Figure 4.10. Directory Layout for /etc

Figure 4.10 depicts the layout of the directory "etc". Every directory contains the file names dot and dot-dot ("." and "..") whose inode numbers are those of the directory and its parent directory, respectively. The inode number of "." in "/etc" is located at offset 0 in the file, and its value is 83. The inode number of ".." is located at offset 16, and its value is 2. Directory entries may be empty, indicated by an inode number of 0. For instance, the entry at address 224 in "/etc" is empty, although it once contained an entry for a file named "crash". The program *mkfs* initializes a file system so that "." and ".." of the root directory have the root inode number of the file system.

The kernel stores data for a directory just as it stores data for an ordinary file, using the inode structure and levels of direct and indirect blocks. Processes may read directories in the same way they read regular files, but the kernel reserves exclusive right to write a directory, thus insuring its correct structure. The access permissions of a directory have the following meaning: read permission on a directory allows a process to read a directory; write permission allows a process to create new directory entries or remove old ones (via the *creat*, *mknod*, *link*, and *unlink* system calls), thereby altering the contents of the directory; execute permission allows a process to search the directory for a file name (it is meaningless to execute a directory). Exercise 4.6 explores the difference between reading and searching a directory.

4.4 CONVERSION OF A PATH NAME TO AN INODE

The initial access to a file is by its path name, as in the *open*, *chdir* (change directory), or *link* system calls. Because the kernel works internally with inodes rather than with path names, it converts the path names to inodes to access files. The algorithm *namei* parses the path name one component at a time, converting each component into an inode based on its name and the directory being searched, and eventually returns the inode of the input path name (Figure 4.11).

Recall from Chapter 2 that every process is associated with (resides in) a current directory; the *u area* contains a pointer to the current directory inode. The current directory of the first process in the system, process 0, is the root directory. The current directory of every other process starts out as the current directory of its parent process at the time it was created (see Section 5.10). Processes change their current directory by executing the *chdir* (change directory) system call. All path name searches start from the current directory of the process unless the path name starts with the slash character, signifying that the search should start from the root directory. In either case, the kernel can easily find the inode where the path name search starts: The current directory is stored in the process *u area*, and the system root inode is stored in a global variable.[2]

Namei uses intermediate inodes as it parses a path name; call them working inodes. The inode where the search starts is the first working inode. During each iteration of the *namei* loop, the kernel makes sure that the working inode is indeed that of a directory. Otherwise, the system would violate the assertion that non-directory files can only be leaf nodes of the file system tree. The process must also have permission to search the directory (read permission is insufficient). The user ID of the process must match the owner or group ID of the file, and execute

2. A process can execute the *chroot* system call to change its notion of the file system root. The changed root is stored in the *u area*.