# 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

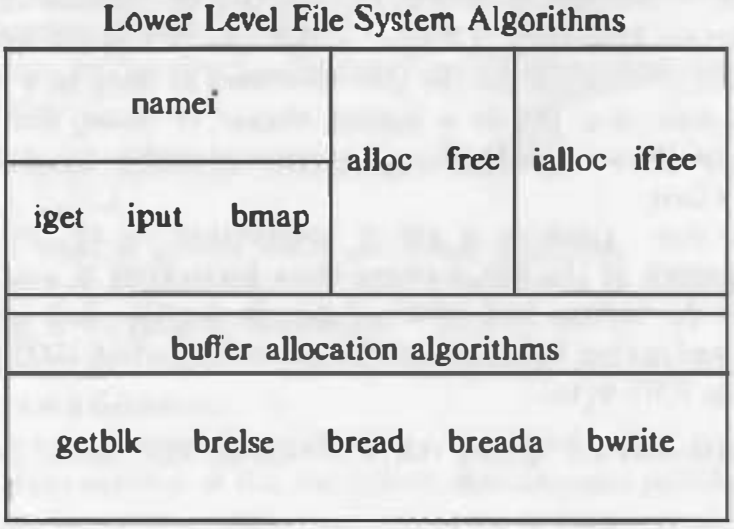



**Figure 4.1.** File System Algorithms

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 *opens* a file. It decrements the reference count only when the reference becomes inactive, for example, when a process *closes* 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.

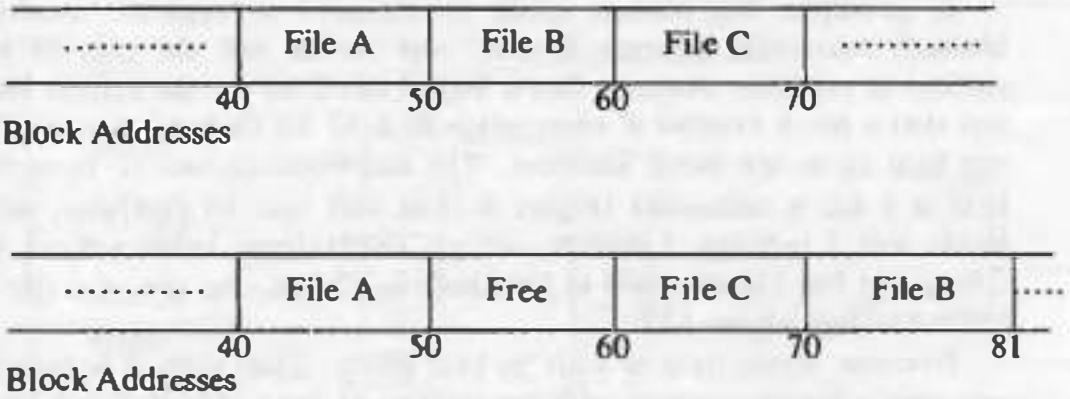



**Figure 4.5.** Allocation of Contiguous Files and Fragmentation of Free Space

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

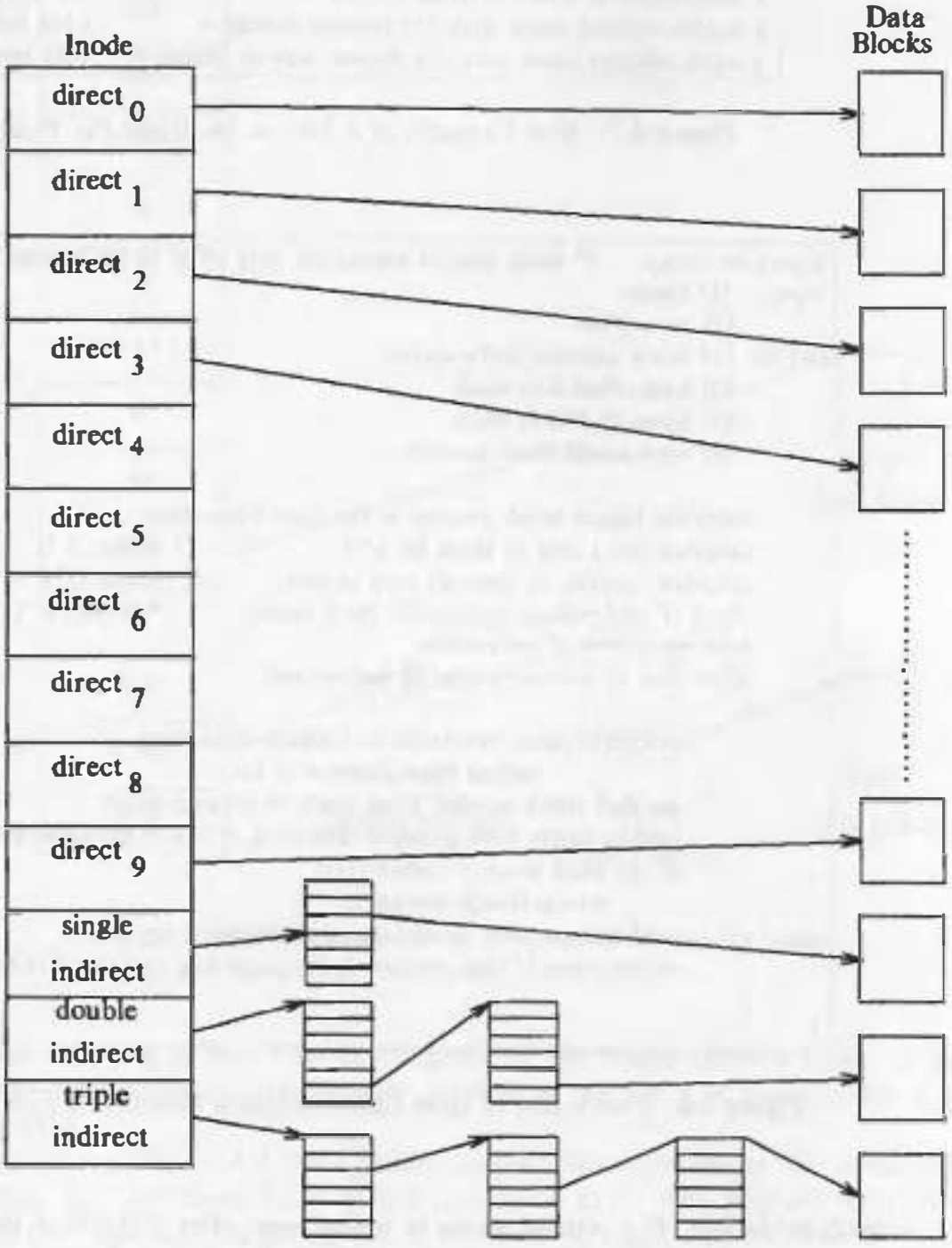



Figure 4.6. Direct and Indirect Blocks in Inode

| | |
|---|---|
| 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

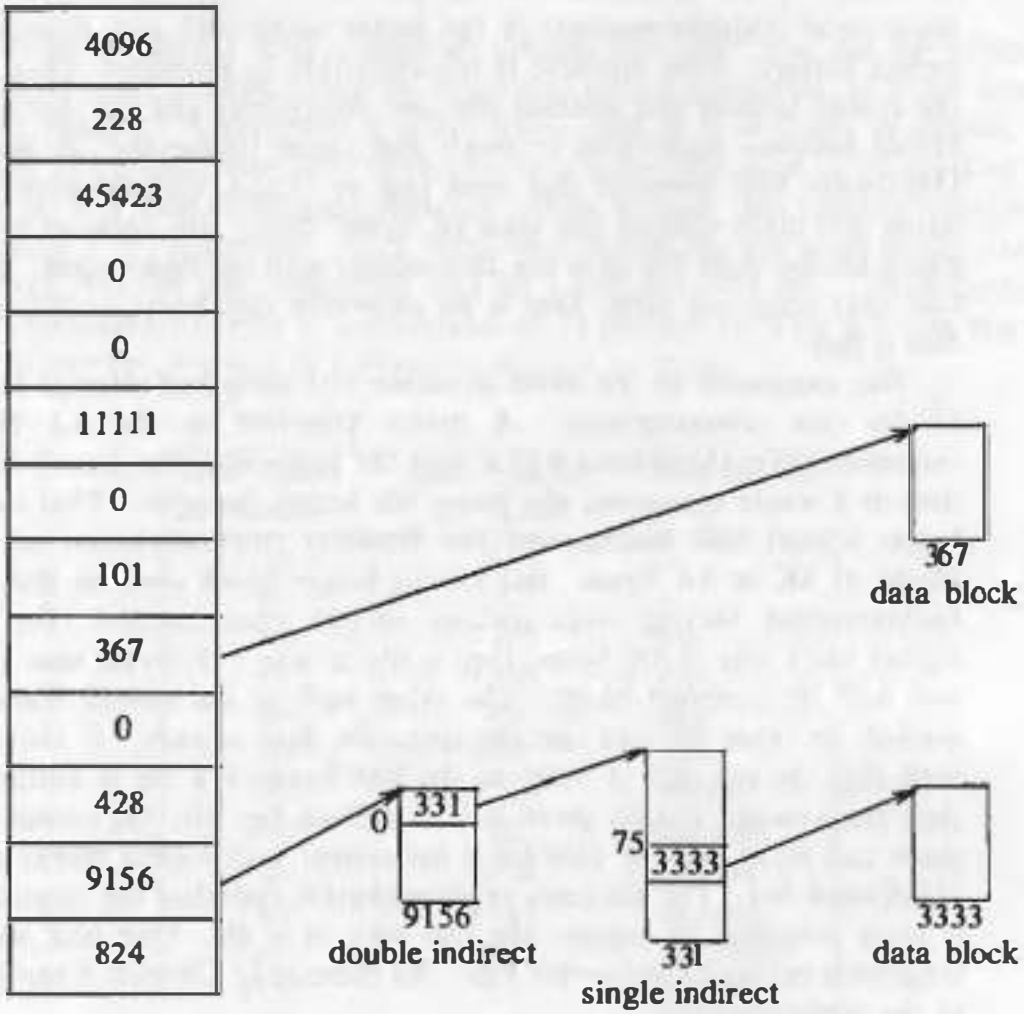



**Figure 4.9.** Block Layout of a Sample File and its Inode

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*.

```
algorithm namei        /* convert path name to inode */
input:   path name
output: locked inode
{
      if (path name starts from root)
            working inode = root inode (algorithm iget);
      else
            working inode = current directory inode (algorithm iget);

      while (there is more path name)
      {
            read next path name component from input;
            verify that working inode is of directory, access permissions OK;
            if (working inode is of root and component is "..")
                  continue;        /* loop back to while */
            read directory (working inode) by repeated use of algorithms
                  bmap, bread and brelse;
            if (component matches an entry in directory (working inode))
            {
                  get inode number for matched component;
                  release working inode (algorithm iput);
                  working inode = inode of matched component (algorithm iget);
            }
            else        /* component not in directory */
                  return (no inode);
      }
      return (working inode);
}
```

**Figure 4.11.** Algorithm for Conversion of a Path Name to an Inode

permission must be granted, or the file must allow search to all users. Otherwise the search fails.

The kernel does a linear search of the directory file associated with the working inode, trying to match the path name component to a directory entry name. Starting at byte offset 0, it converts the byte offset in the directory to the appropriate disk block according to algorithm *bmap* and reads the block using algorithm *bread*. It searches the block for the path name component, treating the contents of the block as a sequence of directory entries. If it finds a match, it records the inode number of the matched directory entry, releases the block (algorithm *brelse*) and the old working inode (algorithm *iput*), and allocates the inode of the matched component (algorithm *iget*). The new inode becomes the working inode. If the kernel does not match the path name with any names in the block, it releases the block, adjusts the byte offset by the number of bytes in a block, converts the new offset to a disk block number (algorithm *bmap*), and reads

the next block. The kernel repeats the procedure until it matches the path name component with a directory entry name, or until it reaches the end of the directory.

For example, suppose a process wants to open the file "/etc/passwd". When the kernel starts parsing the file name, it encounters "/" and gets the system root inode. Making root its current working inode, the kernel gathers in the string "etc". After checking that the current inode is that of a directory ("/") and that the process has the necessary permissions to search it, the kernel searches root for a file whose name is "etc": It accesses the data in the root directory block by block and searches each block one entry at a time until it locates an entry for "etc". On finding the entry, the kernel releases the inode for root (algorithm *iput*) and allocates the inode for "etc" (algorithm *iget*) according to the inode number of the entry just found. After ascertaining that "etc" is a directory and that it has the requisite search permissions, the kernel searches "etc" block by block for a directory structure entry for the file "passwd". Referring to Figure 4.10, it would find the entry for "passwd" as the ninth entry of the directory. On finding it, the kernel releases the inode for "etc", allocates the inode for "passwd", and — since the path name is exhausted — returns that inode.

It is natural to question the efficiency of a linear search of a directory for a path name component. Ritchie points out (see page 1968 of [Ritchie 78b]) that a linear search is efficient because it is bounded by the size of the directory. Furthermore, early UNIX system implementations did not run on machines with large memory space, so there was heavy emphasis on simple algorithms such as linear search schemes. More complicated search schemes could require a different, more complex, directory structure, and would probably run more slowly on small directories than the linear search scheme.

## 4.5 SUPER BLOCK

So far, this chapter has described the structure of a file, assuming that the inode was previously bound to a file and that the disk blocks containing the data were already assigned. The next sections cover how the kernel assigns inodes and disk blocks. To understand those algorithms, let us examine the structure of the super block.

The super block consists of the following fields:

- the size of the file system,
- the number of free blocks in the file system,
- a list of free blocks available on the file system,
- the index of the next free block in the free block list,
- the size of the inode list,
- the number of free inodes in the file system,
- a list of free inodes in the file system,
- the index of the next free inode in the free inode list,

- lock fields for the free block and free inode lists,
- a flag indicating that the super block has been modified.

The remainder of this chapter will explain the use of the arrays, indices and locks. The kernel periodically writes the super block to disk if it had been modified so that it is consistent with the data in the file system.

## 4.6 INODE ASSIGNMENT TO A NEW FILE

The kernel uses algorithm *iget* to allocate a known inode, one whose (file system and) inode number was previously determined. In algorithm *namei* for instance, the kernel determines the inode number by matching a path name component to a name in a directory. Another algorithm, *ialloc*, assigns a disk inode to a newly created file.

The file system contains a linear list of inodes, as mentioned in Chapter 2. An inode is free if its type field is zero. When a process needs a new inode, the kernel could theoretically search the inode list for a free inode. However, such a search would be expensive, requiring at least one read operation (possibly from disk) for every inode. To improve performance, the file system super block contains an array to cache the numbers of free inodes in the file system.

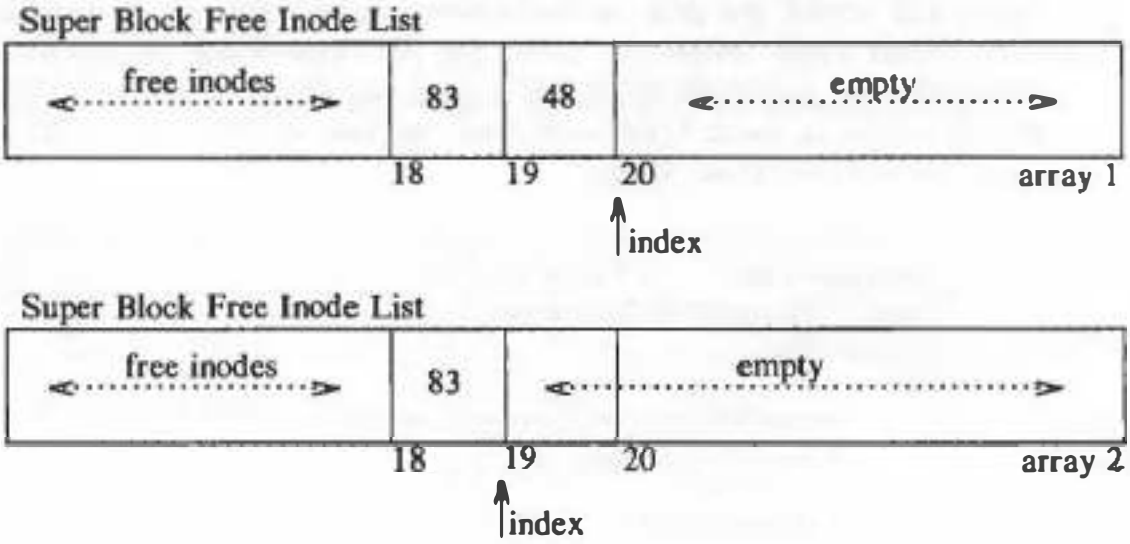
Figure 4.12 shows the algorithm *ialloc* for assigning new inodes. For reasons cited later, the kernel first verifies that no other processes have locked access to the super block free inode list. If the list of inode numbers in the super block is not empty, the kernel assigns the next inode number, allocates a free in-core inode for the newly assigned disk inode using algorithm *iget* (reading the inode from disk if necessary), copies the disk inode to the in-core copy, initializes the fields in the inode, and returns the locked inode. It updates the disk inode to indicate that the inode is now in use: A non-zero file type field indicates that the disk inode is assigned. In the simplest case, the kernel has a good inode, but race conditions exist that necessitate more checking, as will be explained shortly. Loosely defined, a race condition arises when several processes alter common data structures such that the resulting computations depend on the order in which the processes executed, even though all processes obeyed the locking protocol. For example, it is implied here that a process could get a used inode. A race condition is related to the mutual exclusion problem defined in Chapter 2, except that locking schemes solve the mutual exclusion problem there but may not, by themselves, solve all race conditions.

If the super block list of free inodes is empty, the kernel searches the disk and places as many free inode numbers as possible into the super block. The kernel reads the inode list on disk, block by block, and fills the super block list of inode numbers to capacity, remembering the highest-numbered inode that it finds. Call that inode the "remembered" inode; it is the last one saved in the super block. The next time the kernel searches the disk for free inodes, it uses the remembered inode as its starting point, thereby assuring that it wastes no time reading disk blocks

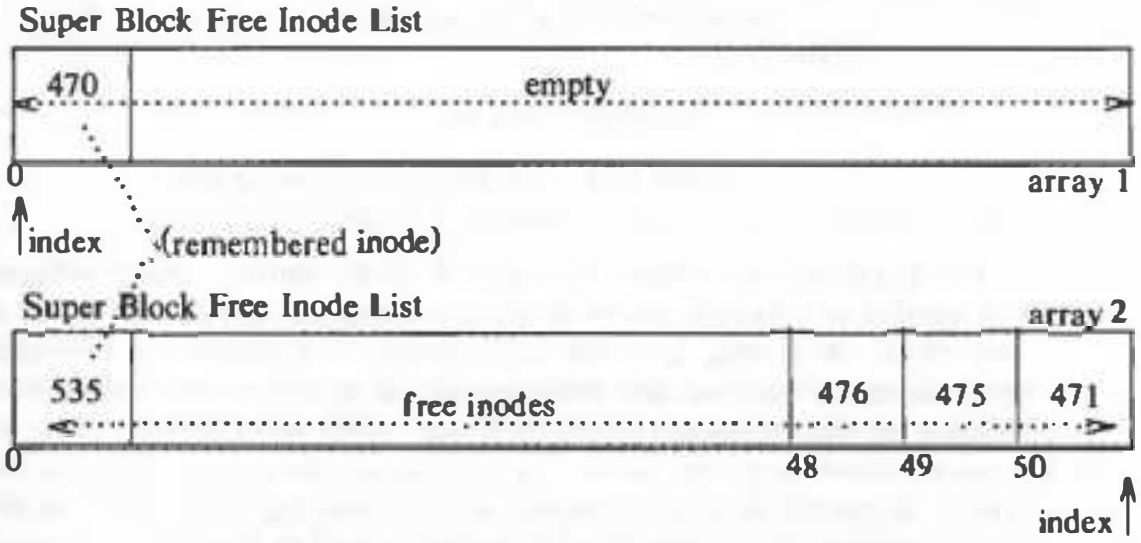
```
algorithm ialloc        /* allocate inode */
input:   file system
output: locked inode
{
        while (not done)
        {
                if (super block locked)
                {
                        sleep (event super block becomes free);
                        continue;       /* while loop */
                }
                if (inode list in super block is empty)
                {
                        lock super block;
                        get remembered inode for free inode search;
                        search disk for free inodes until super block full,
                                or no more free inodes (algorithms bread and brelse);
                        unlock super block;
                        wake up (event super block becomes free);
                        if (no free inodes found on disk)
                                return (no inode);
                        set remembered inode for next free inode search;
                }
                /* there are inodes in super block inode list */
                get inode number from super block inode list;
                get inode (algorithm iget);
                if (inode not free after all)       /* !!! */
                {
                        write inode to disk;
                        release inode (algorithm iput);
                        continue;       /* while loop */
                }
                /* inode is free */
                initialize inode;
                write inode to disk;
                decrement file system free inode count;
                return (inode);
        }
}
```

**Figure 4.12. Algorithm for Assigning New Inodes**

where no free inodes should exist. After gathering a fresh set of free inode numbers, it starts the inode assignment algorithm from the beginning. Whenever the kernel assigns a disk inode, it decrements the free inode count recorded in the super block.




(a) Assigning Free Inode from Middle of List




(b) Assigning Free Inode - Super Block List Empty

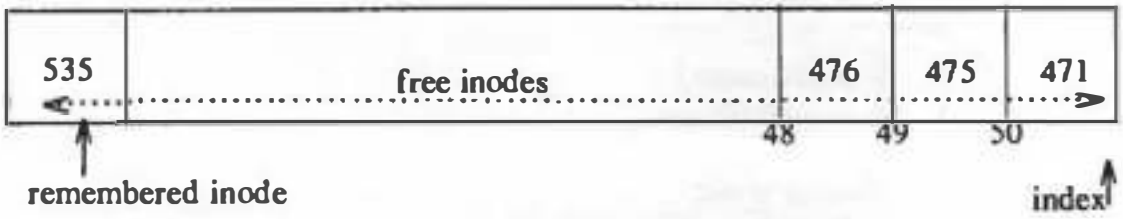
**Figure 4.13.** Two Arrays of Free Inode Numbers

Consider the two pairs of arrays of free inode numbers in Figure 4.13. If the list of free inodes in the super block looks like the first array in Figure 4.13(a) when the kernel assigns an inode, it decrements the index for the next valid inode number to 18 and takes inode number 48. If the list of free inodes in the super block looks like the first array in Figure 4.13(b), it will notice that the array is empty and search the disk for free inodes, starting from inode number 470, the remembered inode. When the kernel fills the super block free list to capacity, it remembers the last inode as the start point for the next search of the disk. The kernel assigns an inode it just took from the disk (number 471 in the figure) and continues whatever it was doing.

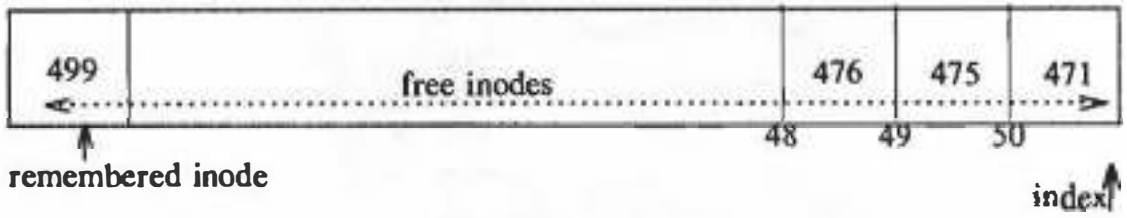
```
algorithm ifree        /* inode free */
input:   file system inode number
output: none
{
        increment file system free inode count;
        if (super block locked)
                return;
        if (inode list full)
        {
                if (inode number less than remembered inode for search)
                        set remembered inode for search = input inode number;
        }
        else
                store inode number in inode list;
        return;
}
```

**Figure 4.14.** Algorithm for Freeing Inode

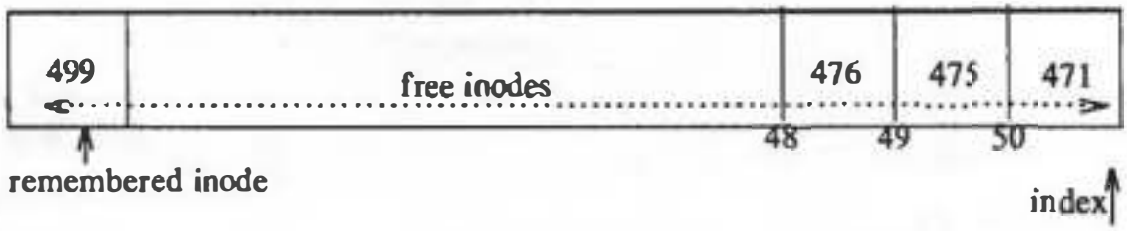
The algorithm for freeing an inode is much simpler. After incrementing the total number of available inodes in the file system, the kernel checks the lock on the super block. If locked, it avoids race conditions by returning immediately: The inode number is not put into the super block, but it can be found on disk and is available for reassignment. If the list is not locked, the kernel checks if it has room for more inode numbers and, if it does, places the inode number in the list and returns. If the list is full, the kernel may not save the newly freed inode there: It compares the number of the freed inode with that of the remembered inode. If the freed inode number is less than the remembered inode number, it "remembers" the newly freed inode number, discarding the old remembered inode number from the super block. The inode is not lost, because the kernel can find it by searching the inode list on disk. The kernel maintains the super block list such that the last inode it dispenses from the list is the remembered inode. Ideally, there should never be free inodes whose inode number is less than the remembered inode number, but




(a) Original Super Block List of Free Inodes




(b) Free Inode 499




(c) Free Inode 601

**Figure 4.15.** Placing Free Inode Numbers into the Super Block

exceptions are possible.

Consider two examples of freeing inodes. If the super block list of free inodes has room for more free inode numbers as in Figure 4.13(a), the kernel places the inode number on the list, increments the index to the next free inode, and proceeds. But if the list of free inodes is full as in Figure 4.15, the kernel compares the inode number it has freed to the remembered inode number that will start the next disk search. Starting with the free inode list in Figure 4.15(a), if the kernel frees inode 499, it makes 499 the remembered inode and evicts number 535 from the free list. If the kernel then frees inode number 601, it does not change the contents of the free list. When it later uses up the inodes in the super block free list, it will search the disk for free inodes starting from inode number 499, and find inodes 535 and 601 again.

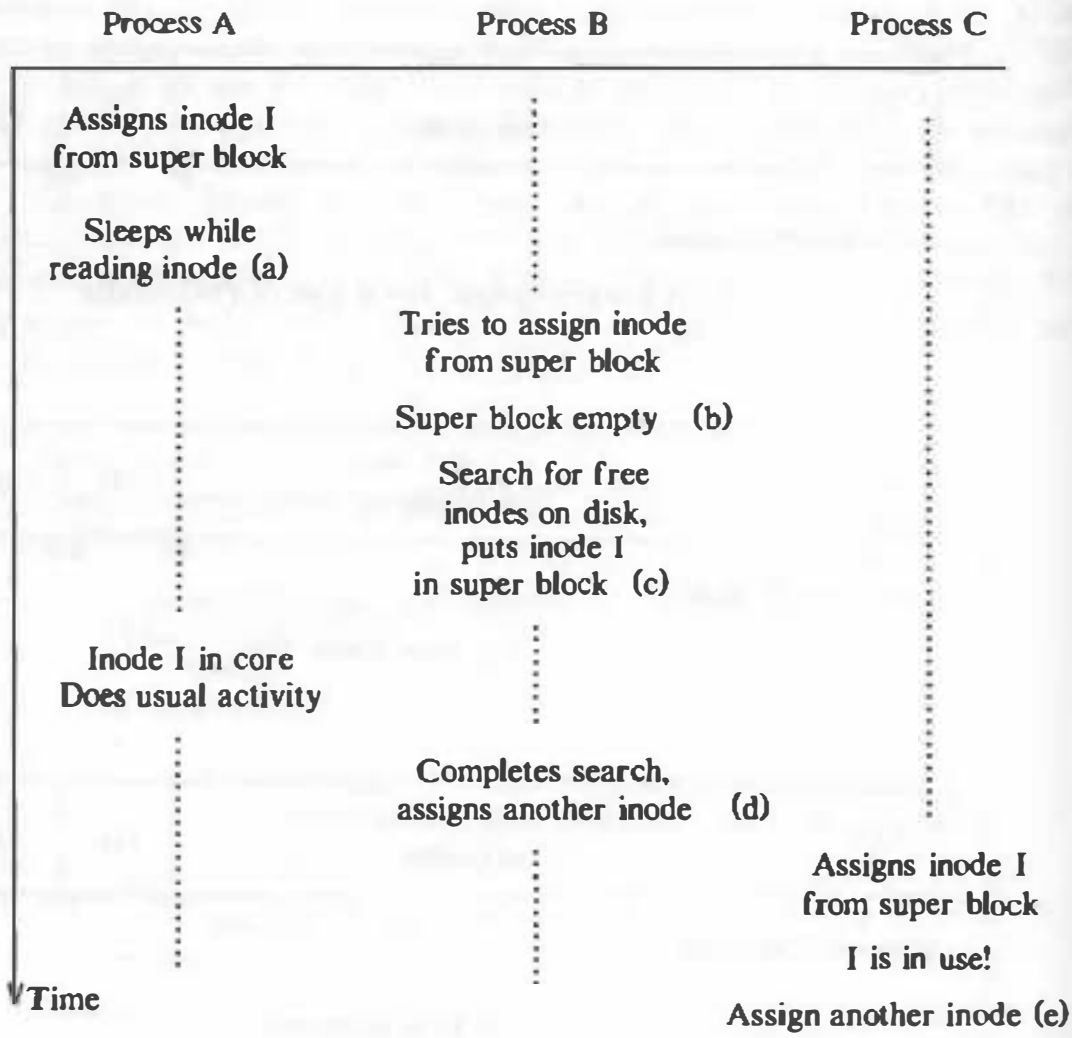



**Figure 4.16.** Race Condition in Assigning Inodes

The preceding paragraph described the simple cases of the algorithms. Now consider the case where the kernel assigns a new inode and then allocates an in-core copy for the inode. The algorithm implies that the kernel could find that the inode had already been assigned. Although rare, the following scenario shows such a case (refer to Figures 4.16 and 4.17). Consider three processes, A, B, and C, and suppose that the kernel, acting on behalf of process A,[3] assigns inode I but goes to sleep before it copies the disk inode into the in-core copy. Algorithms *iget* (invoked

3. As in the last chapter, the term "process" here will mean "the kernel, acting on behalf of a process."

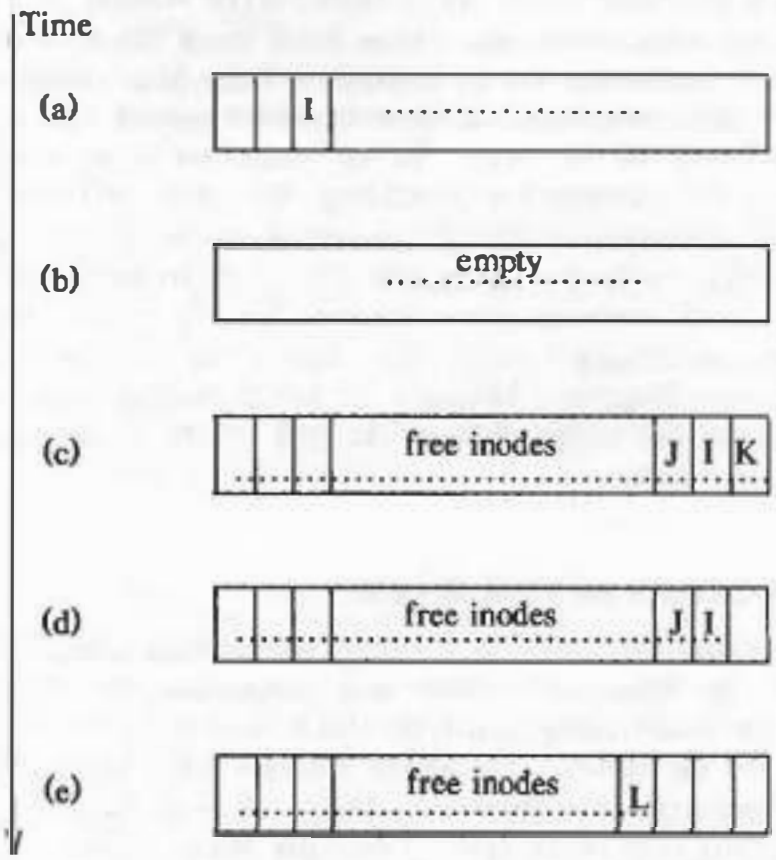



**Figure 4.17.** Race Condition in Assigning Inodes (continued)

by *ialloc*) and *bread* (invoked by *iget*) give process A ample opportunity to go to sleep. While process A is asleep, suppose process B attempts to assign a new inode but discovers that the super block list of free inodes is empty. Process B searches the disk for free inodes, and suppose it starts its search for free inodes at an inode number lower than that of the inode that A is assigning. It is possible for process B to find inode I free on the disk since process A is still asleep, and the kernel does not know that the inode is about to be assigned. Process B, not realizing the danger, completes its search of the disk, fills up the super block with (supposedly) free inodes, assigns an inode, and departs from the scene. However, inode I is in the super block free list of inode numbers. When process A wakes up, it completes the assignment of inode I. Now suppose process C later requests an inode and happens to pick inode I from the super block free list. When it gets the in-core copy of the inode, it will find its file type set, implying that the inode was already assigned. The kernel checks for this condition and, finding that the inode has been assigned, tries to assign a new one. Writing the updated inode to disk immediately after its assignment in *ialloc* makes the chance of the race smaller, because the file type field will mark the inode in use.

Locking the super block list of inodes while reading in a new set from disk prevents other race conditions. If the super block list were not locked, a process could find it empty and try to populate it from disk, occasionally sleeping while waiting for I/O completion. Suppose a second process also tried to assign a new inode and found the list empty. It, too, would try to populate the list from disk. At best, the two processes are duplicating their efforts and wasting CPU power. At worst, race conditions of the type described in the previous paragraph would be more frequent. Similarly, if a process freeing an inode did not check that the list is locked, it could overwrite inode numbers already in the free list while another process was populating it from disk. Again, the race conditions described above would be more frequent. Although the kernel handles them satisfactorily, system performance would suffer. Use of the lock on the super block free list prevents such race conditions.

## 4.7 ALLOCATION OF DISK BLOCKS

When a process writes data to a file, the kernel must allocate disk blocks from the file system for direct data blocks and, sometimes, for indirect blocks. The file system super block contains an array that is used to cache the numbers of free disk blocks in the file system. The utility program *mkfs* (make file system) organizes the data blocks of a file system in a linked list, such that each link of the list is a disk block that contains an array of free disk block numbers, and one array entry is the number of the next block of the linked list. Figure 4.18 shows an example of the linked list, where the first block is the super block free list and later blocks on the linked list contain more free block numbers.

When the kernel wants to allocate a block from a file system (algorithm *alloc*, Figure 4.19), it allocates the next available block in the super block list. Once allocated, the block cannot be reallocated until it becomes free. If the allocated block is the last available block in the super block cache, the kernel treats it as a pointer to a block that contains a list of free blocks. It reads the block, populates the super block array with the new list of block numbers, and then proceeds to use the original block number. It allocates a buffer for the block and clears the buffer's data (zeros it). The disk block has now been assigned, and the kernel has a buffer to work with. If the file system contains no free blocks, the calling process receives an error.

If a process *write*s a lot of data to a file, it repeatedly asks the system for blocks to store the data, but the kernel assigns only one block at a time. The program *mkfs* tries to organize the original linked list of free block numbers so that block numbers dispensed to a file are near each other. This helps performance, because it reduces disk seek time and latency when a process reads a file sequentially. Figure 4.18 depicts block numbers in a regular pattern, presumably based on the disk rotation speed. Unfortunately, the order of block numbers on the free block linked lists breaks down with heavy use as processes *write* files and remove them, because block numbers enter and leave the free list at random. The kernel makes no

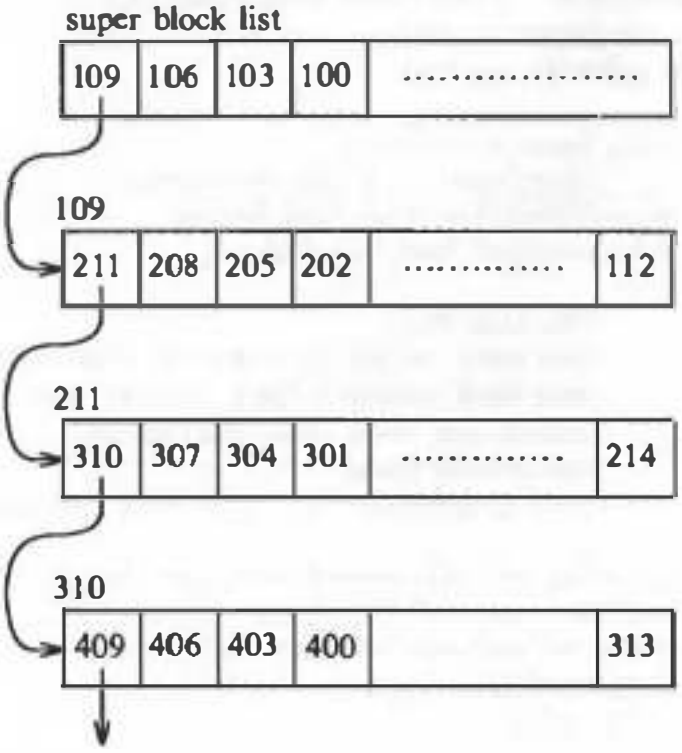



**Figure 4.18.** Linked List of Free Disk Block Numbers

attempt to sort block numbers on the free list.

The algorithm *free* for freeing a block is the reverse of the one for allocating a block. If the super block list is not full, the block number of the newly freed block is placed on the super block list. If, however, the super block list is full, the newly freed block becomes a link block; the kernel writes the super block list into the block and writes the block to disk. It then places the block number of the newly freed block in the super block list: That block number is the only member of the list.

Figure 4.20 shows a sequence of *alloc* and *free* operations, starting with one entry on the super block free list. The kernel frees block 949 and places the block number on the free list. It then allocates a block and removes block number 949 from the free list. Finally, it allocates a block and removes block number 109 from the free list. Because the super block free list is now empty, the kernel replenishes the list by copying in the contents of block 109, the next link on the linked list. Figure 4.20(d) shows the full super block list and the next link block, block 211.

The algorithms for assigning and freeing inodes and disk blocks are similar in that the kernel uses the super block as a cache containing indices of free resources, block numbers, and inode numbers. It maintains a linked list of block numbers such that every free block number in the file system appears in some element of the linked list, but it maintains no such list of free inodes. There are three reasons for

```
algorithm alloc     /* file system block allocation */
input:   file system number
output: buffer for new block
{
        while (super block locked)
                sleep (event super block not locked);
        remove block from super block free list;
        if (removed last block from free list)
        {
                lock super block;
                read block just taken from free list (algorithm bread);
                copy block numbers in block into super block;
                release block buffer (algorithm brelse);
                unlock super block;
                wake up processes (event super block not locked);
        }
        get buffer for block removed from super block list (algorithm getblk);
        zero buffer contents;
        decrement total count of free blocks;
        mark super block modified;
        return buffer;
}
```

**Figure 4.19.** Algorithm for Allocating Disk Block

the different treatment.

1. The kernel can determine whether an inode is free by inspection: If the file type field is clear, the inode is free. The kernel needs no other mechanism to describe free inodes. However, it cannot determine whether a block is free just by looking at it. It could not distinguish between a bit pattern that indicates the block is free and data that happened to have that bit pattern. Hence, the kernel requires an external method to identify free blocks, and traditional implementations have used a linked list.
2. Disk blocks lend themselves to the use of linked lists: A disk block easily holds large lists of free block numbers. But inodes have no convenient place for bulk storage of large lists of free inode numbers.
3. Users tend to consume disk block resources more quickly than they consume inodes, so the apparent lag in performance when searching the disk for free inodes is not as critical as it would be for searching for free disk blocks.

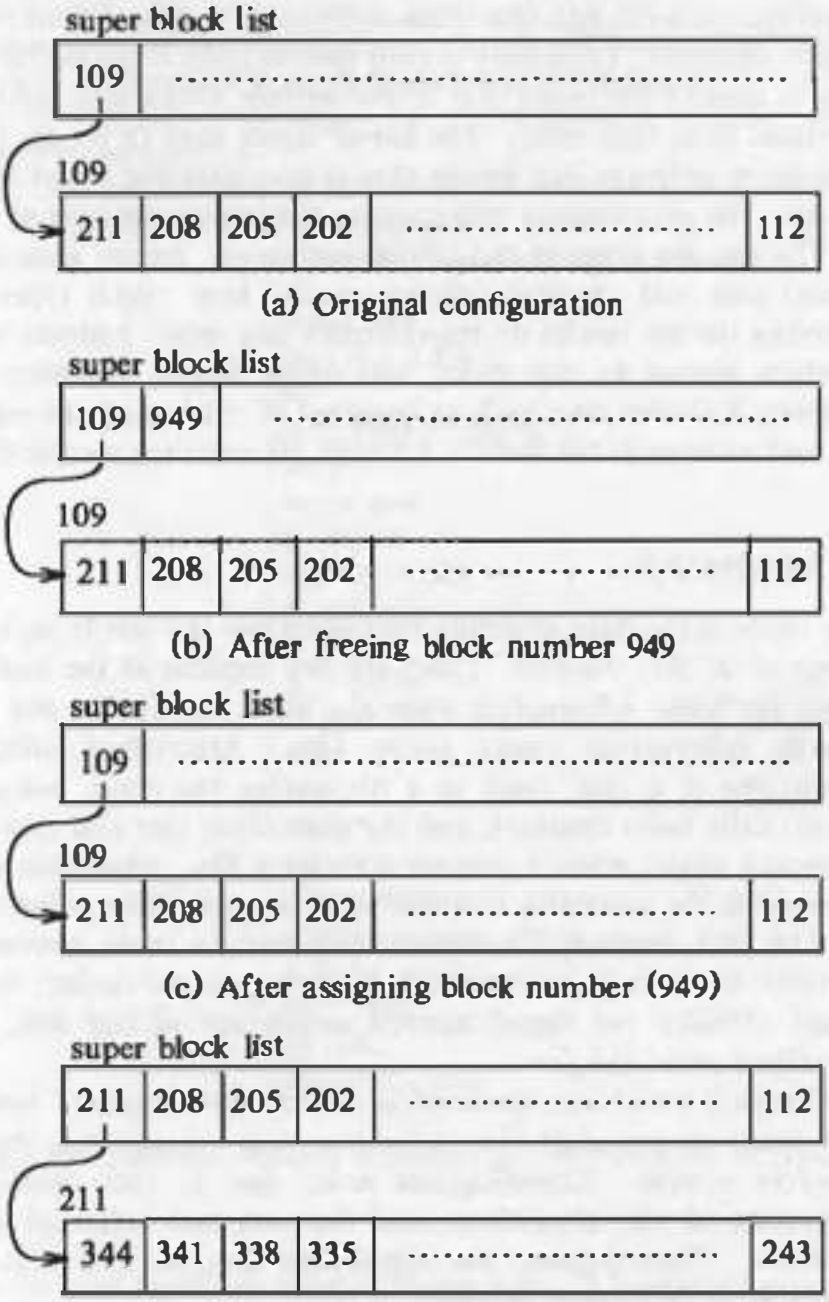



**Figure 4.20.** Requesting and Freeing Disk Blocks

## 4.8 OTHER FILE TYPES

The UNIX system supports two other file types: pipes and special files. A pipe, sometimes called a *fifo* (for "first-in-first-out"), differs from a regular file in that its data is transient: Once data is read from a pipe, it cannot be read again. Also, the data is read in the order that it was written to the pipe, and the system allows no deviation from that order. The kernel stores data in a pipe the same way it stores data in an ordinary file, except that it uses only the direct blocks, not the indirect blocks. The next chapter will examine the implementation of pipes.

The last file types in the UNIX system are special files, including block device special files and character device special files. Both types specify devices, and therefore the file inodes do not reference any data. Instead, the inode contains two numbers known as the major and minor device numbers. The major number indicates a device type such as terminal or disk, and the minor number indicates the unit number of the device. Chapter 10 examines special devices in detail.

## 4.9 SUMMARY

The inode is the data structure that describes the attributes of a file, including the layout of its data on disk. There are two versions of the inode: the disk copy that stores the inode information when the file is not in use and the in-core copy that records information about active files. Algorithms *ialloc* and *ifree* control assignment of a disk inode to a file during the *creat*, *mknod*, *pipe*, and *unlink* system calls (next chapter), and the algorithms *iget* and *iput* control the allocation of in-core inodes when a process accesses a file. Algorithm *bmap* locates the disk blocks of a file, according to a previously supplied byte offset in the file. Directories are files that correlate file name components to inode numbers. Algorithm *namei* converts file names manipulated by processes to inodes, used internally by the kernel. Finally, the kernel controls assignment of new disk blocks to a file using algorithms *alloc* and *free*.

The data structures discussed in this chapter consist of linked lists, hash queues, and linear arrays, and the algorithms that manipulate the data structures are therefore simple. Complications arise due to race conditions caused by the interaction of the algorithms, and the text has indicated some of these timing problems. Nevertheless, the algorithms are not elaborate and illustrate the simplicity of the system design.

The structures and algorithms explained here are internal to the kernel and are not visible to the user. Referring to the overall system architecture (Figure 2.1), the algorithms described in this chapter occupy the lower half of the file subsystem. The next chapter examines the system calls that provide the user interface to the file system, and it describes the upper half of the file subsystem that invokes the internal algorithms described here.

## 4.10 EXERCISES

1. The C language convention counts array indices from 0. Why do inode numbers start from 1 and not 0?
2. If a process sleeps in algorithm *iget* when it finds the inode locked in the cache, why must it start the loop again from the beginning after waking up?
3. Describe an algorithm that takes an in-core inode as input and updates the corresponding disk inode.
4. The algorithms *iget* and *iput* do not require the processor execution level to be raised to block out interrupts. What does this imply?
5. How efficiently can the loop for indirect blocks in *bmap* be encoded?

```
mkdir junk
for i in 1 2 3 4 5
do
echo hello > junk/$i
done
ls -ld junk
ls -l junk
chmod -r junk
ls -ld junk
ls junk
ls -l junk
cd junk
pwd
ls -l
echo *
cd ..
chmod +r junk
chmod -x junk
ls junk
ls -l junk
cd junk
chmod +x junk
```

**Figure 4.21.** Difference between Read and Search Permission on Directories

6. Execute the shell command script in Figure 4.21. It creates a directory "junk" and creates five files in the directory. After doing some control *ls* commands, the *chmod* command turns off read permission for the directory. What happens when the various *ls* commands are executed now? What happens after changing directory into "junk"? After restoring read permission but removing execute (search) permission from "junk", repeat the experiment. What happens? What is happening in the kernel to cause this behavior?
7. Given the current structure of a directory entry on a System V system, what is the maximum number of files a file system can contain?

8. UNIX System V allows a maximum of 14 characters for a path name component. *Namei* truncates extra characters in a component. How should the file system and respective algorithms be redesigned to allow arbitrary length component names?
9. Suppose a user has a private version of the UNIX system but changes it so that a path name component can consist of 30 characters; the private version of the operating system stores the directory entries the same way that the standard operating system does, except that the directory entries are 32 bytes long instead of 16. If the user mounts the private file system on a standard system, what would happen in algorithm *namei* when a process accesses a file on the private file system?
- 10. Consider the algorithm *namei* for converting a path name into an inode. As the search progresses, the kernel checks that the current working inode is that of a directory. Is it possible for another process to remove (*unlink*) the directory? How can the kernel prevent this? The next chapter will come back to this problem.
- 11. Design a directory structure that improves the efficiency of searching for path names by avoiding the linear search. Consider two techniques: hashing and $n$-ary trees.
- 12. Design a scheme that reduces the number of directory searches for file names by caching frequently used names.
- 13. Ideally, a file system should never contain a free inode whose inode number is less than the "remembered" inode used by *ialloc*. How is it possible for this assertion to be false?
14. The super block is a disk block and contains other information besides the free block list, as described in this chapter. Therefore, the super block free list cannot contain as many free block numbers as can be potentially stored in a disk block on the linked list of free disk blocks. What is the optimal number of free block numbers that should be stored in a block on the linked list?
- 15. Discuss a system implementation that keeps track of free disk blocks with a bit map instead of a linked list of blocks. What are the advantages and disadvantages of this scheme?

# 5

# SYSTEM CALLS FOR THE FILE SYSTEM

The last chapter described the internal data structures for the file system and the algorithms that manipulate them. This chapter deals with system calls for the file system, using the concepts explored in the previous chapter. It starts with system calls for accessing existing files, such as *open*, *read*, *write*, *lseek*, and *close*, then presents system calls to create new files, namely, *creat* and *mknod*, and 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 *umount* 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. The last section in the chapter covers file system maintenance. The chapter introduces three kernel data structures: the file table, with one entry allocated for every opened file in the system, the user file descriptor table, with one entry allocated for every file descriptor known to a process, and the mount table, containing information for every active file system.

Figure 5.1 shows the relationship between the system calls and the algorithms described previously. It classifies the system calls into several categories, although some system calls appear in more than one category:

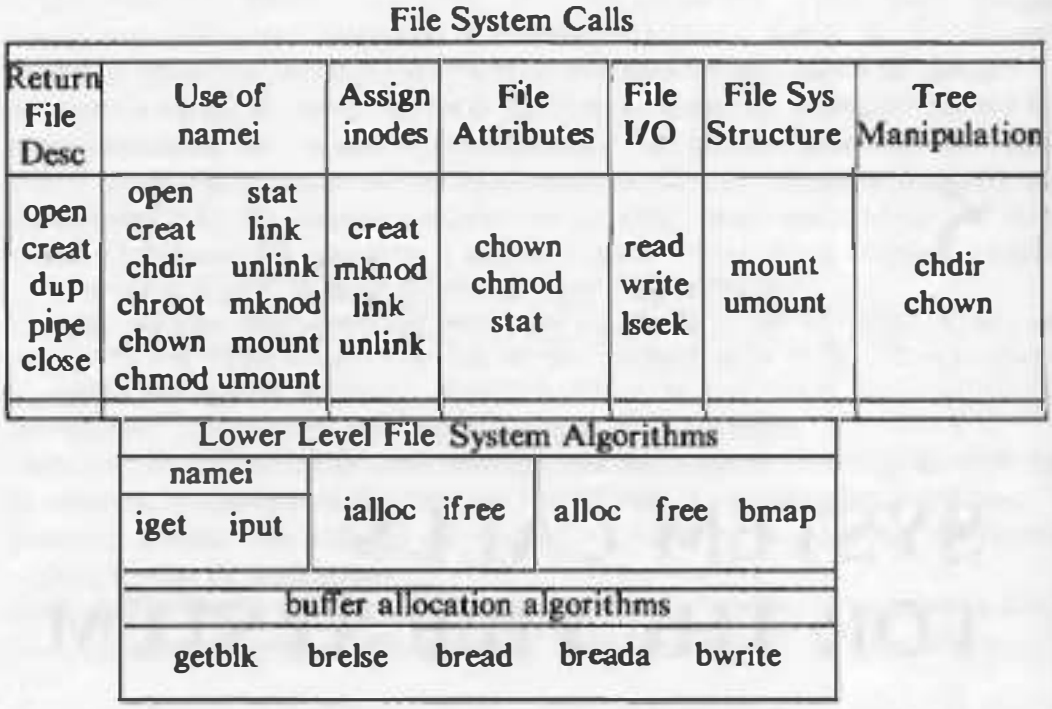



**Figure 5.1.** File System Calls and Relation to Other Algorithms

- 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.

## 5.1 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[1] 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* (see Figure 5.2). It checks permissions for opening the file after it finds the in-core inode and allocates an entry in the file table for the open file. 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. The kernel initializes the offset to 0 during the *open* call, meaning that the initial *read* or *write* starts at the beginning of a file by default. Alternatively, a process can *open* a file in *write-append* mode, in which case the kernel initializes the offset to the size of the file. The kernel allocates an entry in a private table in the process *u area*, called the user file descriptor table, and notes the index of this entry. The index is the file descriptor that is returned to the user. The entry in the user file table points to the entry in the global file table.

```
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 5.2.** 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.[2]

1. All system calls return the value −1 if they fail. The return value −1 will not be explicitly mentioned when discussing the syntax of the system calls.
2. The definition of the *open* system call specifies three parameters (the third is used for the *create* mode of open), but programmers usually use only the first two. The C compiler does not check that the number of parameters is correct. System implementations typically pass the first two parameters and a third "garbage" parameter (whatever happens to be on the stack) to the kernel. The kernel

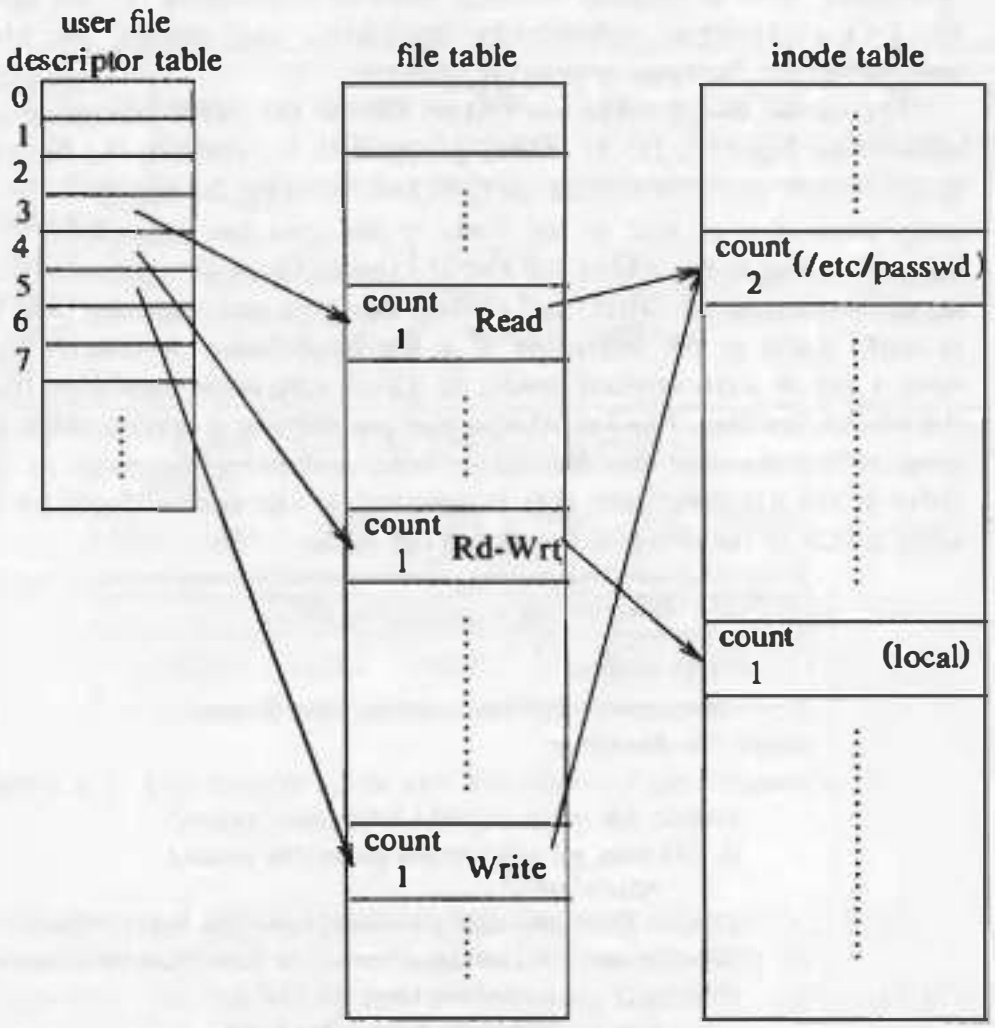



**Figure 5.3.** Data Structures after Open

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_RDWR);
fd3 = open("/etc/passwd", O_WRONLY);
```

Figure 5.3 shows the relationship between the inode table, file table, and user file descriptor data structures. 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

---

does not check the third parameter unless the second parameter indicates that it must, allowing programmers to encode only two parameters.

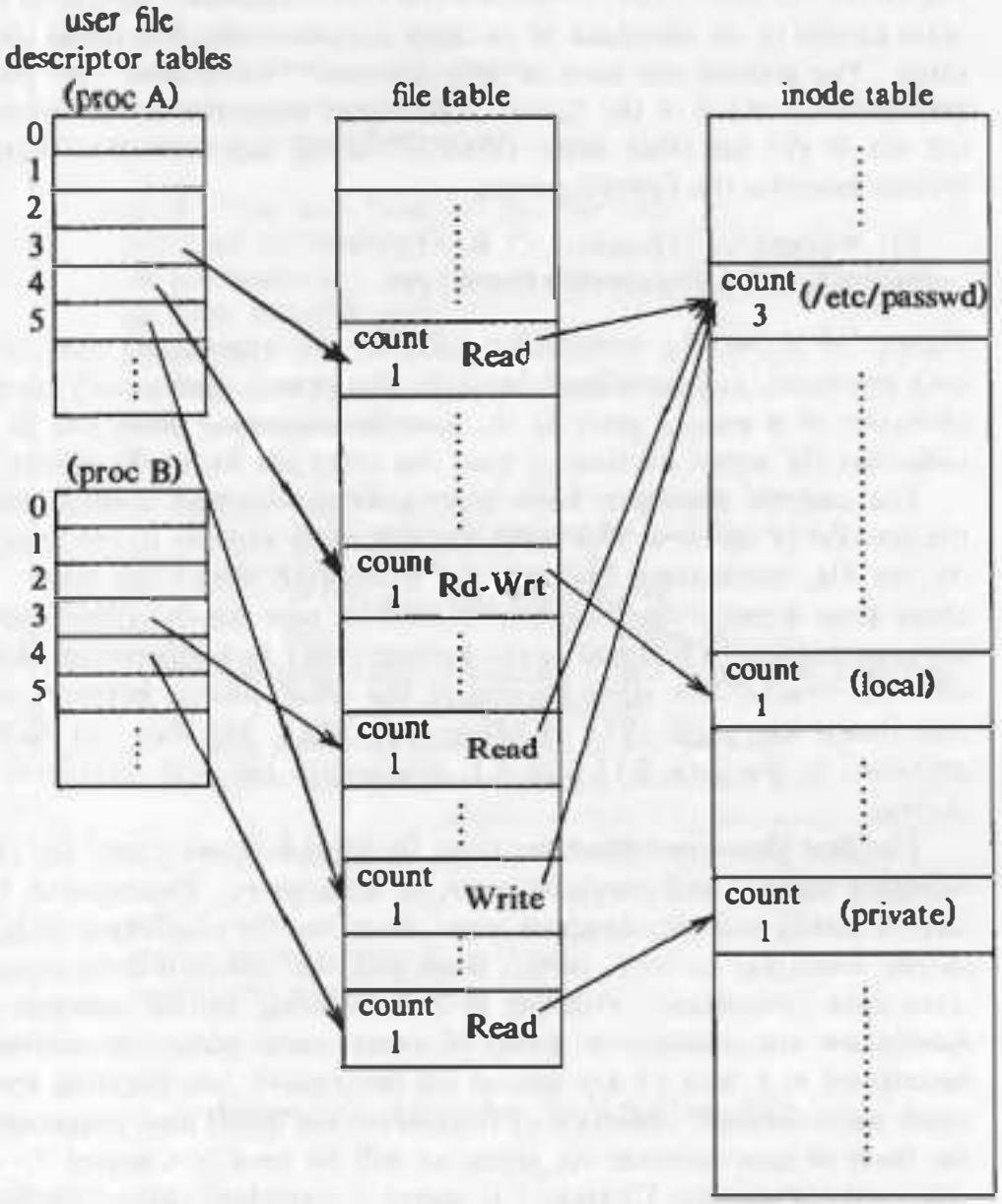



**Figure 5.4.** Data Structures after Two Processes Open Files

the kernel file table even though 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" but only through file descriptors 3 and 5 in the figure. The kernel notes the capability to read or write the file in the file table entry allocated during the *open* call. Suppose a second process executes the following code.

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);
```

Figure 5.4 shows the relationship between the appropriate data structures while both processes (and no others) have the files open. Again, each *open* call results in allocation of a unique entry in the user file descriptor table and in the kernel file table, but the kernel contains at most one entry per file in the in-core inode table.

The user file descriptor table entry could conceivably contain the file offset for the position of the next I/O operation and point directly to the in-core inode entry for the file, eliminating the need for a separate kernel file table. The examples above show a one-to-one relationship between user file descriptor entries and kernel file table entries. Thompson notes, however, that he implemented the file table as a separate structure to allow sharing of the offset pointer between several user file descriptors (see page 1943 of [Thompson 78]). The *dup* and *fork* system calls, explained in Sections 5.13 and 7.1, manipulate the data structures to allow such sharing.

The first three user file descriptors (0, 1, and 2) are called the *standard input*, *standard output*, and *standard error* file descriptors. Processes on UNIX systems conventionally use the standard input descriptor to read input data, the standard output descriptor to write output data, and the standard error descriptor to write error data (messages). Nothing in the operating system assumes that these file descriptors are special. A group of users could adopt the convention that file descriptors 4, 6, and 11 are special file descriptors, but counting from 0 (in C) is much more natural. Adoption of the convention by all user programs makes it easy for them to communicate via *pipes*, as will be seen in Chapter 7. Normally, the control terminal (see Chapter 10) serves as standard input, standard output and standard error.

## 5.2 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 5.5 depicts the algorithm *read* for reading a regular file. The kernel gets the file table entry that corresponds to

```
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 5.5.** Algorithm for Reading a File

the user file descriptor, following the pointer in Figure 5.3. It now sets several I/O parameters in the *u area* (Figure 5.6), eliminating the need to pass them as function parameters. Specifically, it sets the I/O mode to indicate that a read is being done, a flag to indicate that the I/O will go to user address space, a count field to indicate the number of bytes to read, the target address of the user data buffer, and finally, an offset field (from the file table) to indicate the byte offset into the file where the I/O should begin. After the kernel sets the I/O parameters in the *u area*, it follows the pointer from the file table entry to the inode, locking the inode before it reads the file.

The algorithm now goes into a loop until the *read* is satisfied. The kernel converts the file byte offset into a block number, using algorithm *bmap*, and it notes the byte offset in the block where the I/O should begin and how many bytes

| | |
|---|---|
| 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 |

**Figure 5.6.** I/O Parameters Saved in U Area

in the block it should read. After reading the block into a buffer, possibly using block read ahead (algorithms *bread* and *breada*) as will be described, it copies the data from the block to the target address in the user process. It updates the I/O parameters in the *u area* according to the number of bytes it read, incrementing the file byte offset and the address in the user process where the next data should be delivered, and decrementing the count of bytes it needs to read to satisfy the user read request. If the user request is not satisfied, the kernel repeats the entire cycle, converting the file byte offset to a block number, reading the block from disk to a system buffer, copying data from the buffer to the user process, releasing the buffer, and updating I/O parameters in the *u area*. The cycle completes either when the kernel completely satisfies the user request, when the file contains no more data, or if the kernel encounters an error in reading the data from disk or in copying the data to user space. The kernel updates the offset in the file table according to the number of bytes it actually read; consequently, successive *read*s of a file deliver the file data in sequence. The *lseek* system call (Section 5.6) adjusts the value of the file table offset and changes the order in which a process *read*s or *write*s a file.

```
#include <fcntl.h>
main()
{
        int fd;
        char lilbuf[20], bigbuf[1024];

        fd = open("/etc/passwd", O_RDONLY);
        read(fd, lilbuf, 20);
        read(fd, bigbuf, 1024);
        read(fd, lilbuf, 20);
}
```

**Figure 5.7.** Sample Program for Reading a File

Consider the program in Figure 5.7. The *open* returns a file descriptor that the user assigns to the variable *fd* and uses in the subsequent *read* calls. In the *read* system call, the kernel verifies that the file descriptor parameter is legal, and that

the process had previously *opened* the file for reading. It stores the values *lilbuf*, 20, and 0 in the *u area*, corresponding to the address of the user buffer, the byte count, and the starting byte offset in the file. It calculates that byte offset 0 is in the 0th block of the file and retrieves the entry for the 0th block in the inode. Assuming such a block exists, the kernel reads the entire block of 1024 bytes into a buffer but copies only 20 bytes to the user address *lilbuf*. It increments the *u area* byte offset to 20 and decrements the count of data to read to 0. Since the *read* has been satisfied, the kernel resets the file table offset to 20, so that subsequent *read*s on the file descriptor will begin at byte 20 in the file, and the system call returns the number of bytes actually read, 20.

For the second *read* call, the kernel again verifies that the descriptor is legal and that the process had *opened* the file for reading, because it has no way of knowing that the user *read* request is for the same file that was determined to be legal during the last *read*. It stores in the *u area* the user address *bigbuf*, the number of bytes the process wants to read, 1024, and the starting offset in the file, 20, taken from the file table. It converts the file byte offset to the correct disk block, as above, and reads the block. If the time between *read* calls is small, chances are good that the block will be in the buffer cache. But the kernel cannot satisfy the *read* request entirely from the buffer, because only 1004 out of the 1024 bytes for this request are in the buffer. So it copies the last 1004 bytes from the buffer into the user data structure *bigbuf* and updates the parameters in the *u area* to indicate that the next iteration of the read loop starts at byte 1024 in the file, that the data should be copied to byte position 1004 in *bigbuf*, and that the number of bytes to to satisfy the *read* request is 20.

The kernel now cycles to the beginning of the loop in the *read* algorithm. It converts byte offset 1024 to logical block offset 1, looks up the second direct block number in the inode, and finds the correct disk block to read. It reads the block from the buffer cache, reading the block from disk if it is not in the cache. Finally, it copies 20 bytes from the buffer to the correct address in the user process. Before leaving the system call, the kernel sets the offset field in the file table entry to 1044, the byte offset that should be accessed next. For the last *read* call in the example, the kernel proceeds as in the first *read* call, except that it starts reading at byte 1044 in the file, finding that value in the offset field in the file table entry for the descriptor.

The example shows how advantageous it is for I/O requests to start on file system block boundaries and to be multiples of the block size. Doing so allows the kernel to avoid an extra iteration in the *read* algorithm loop, with the consequent expense of accessing the inode to find the correct block number for the data and competing with other processes for access to the buffer pool. The standard I/O library was written to hide knowledge of the kernel buffer size from users; its use avoids the performance penalties inherent in processes that nibble at the file system inefficiently (see exercise 5.4).

As the kernel goes through the *read* loop, it determines whether a file is subject to read-ahead: if a process *read*s 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. Of course, if a process does not *read* to the end of a block, the kernel does not invoke read-ahead for the next block.

Recall from Figure 4.9 that it is possible for some block numbers in an inode or in indirect blocks to have the value 0, even though later blocks have nonzero value. If a process attempts to *read* data from such a block, the kernel satisfies the request by allocating an arbitrary buffer in the *read* loop, clearing its contents to 0, and copying it to the user address. This case is different from the case where a process encounters the end of a file, meaning that no data was ever written to any location beyond the current point. When encountering end of file, the kernel returns no data to the process (see exercise 5.1).

When a process invokes the *read* system call, the kernel locks the inode for the duration of the call. Afterwards, it could go to sleep reading a buffer associated with data or with indirect blocks of the inode. If another process were allowed to change the file while the first process was sleeping, *read* could return inconsistent data. For example, a process may *read* several blocks of a file; if it slept while reading the first block and a second process were to *write* the other blocks, the returned data would contain a mixture of old and new data. Hence, the inode is left locked for the duration of the *read* call, affording the process a consistent view of the file as it existed at the start of the call.

The kernel can preempt a *read*ing process between system calls in user mode and schedule other processes to run. Since the inode is unlocked at the end of a system call, nothing prevents other processes from accessing the file and changing its contents. It would be unfair for the system to keep an inode locked from the time a process *open*ed the file until it *close*d the file, because one process could keep a file open and thus prevent other processes from ever accessing it. If the file was "/etc/passwd", used by the login process to check a user's password, then one malicious (or, perhaps, just errant) user could prevent all other users from logging in. To avoid such problems, the kernel unlocks the inode at the end of each system call that uses it. If another process changes the file between the two *read* system calls by the first process, the first process may *read* unexpected data, but the kernel data structures are consistent.

For example, suppose the kernel executes the two processes in Figure 5.8 concurrently. Assuming both processes complete their *open* calls before either one starts its *read* or *write* calls, the kernel could execute the *read* and *write* calls in any of six sequences: *read1, read2, write1, write2*, or *read1, write1, read2, write2*, or *read1, write1, write2, read2*, and so on. The data that process A *read*s depends on the order that the system executes the system calls of the two processes; the system does not guarantee that the data in the file remains the same after *open*ing the file. Use of the *file and record locking* feature (Section 5.4) allows a process to

```
#include  <fcntl.h>
/* process A */
main()
{
        int fd;
        char buf[512];
        fd = open("/etc/passwd", O_RDONLY);
        read(fd, buf, sizeof(buf));        /* read1 */
        read(fd, buf, sizeof(buf));        /* read2 */
}

/* process B */
main()
{
        int fd, i;
        char buf[512];
        for (i = 0;  i < sizeof(buf);  i++)
                buf[i] = 'a';
        fd = open("/etc/passwd", O_WRONLY);
        write(fd, buf, sizeof(buf));       /* write1 */
        write(fd, buf, sizeof(buf));       /* write2 */
}
```

**Figure 5.8.** A Reader and a Writer Process

guarantee file consistency while it has a file *open*.

Finally, the program in Figure 5.9 shows how a process can *open* a file more than once and *read* it via different file descriptors. The kernel manipulates the file table offsets associated with the two file descriptors independently, and hence, the arrays *buf1* and *buf2* should be identical when the process completes, assuming no other process *writes* "/etc/passwd" in the meantime.

## 5.3 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

```
#include <fcntl.h>
main()
{
        int fd1, fd2;
        char buf1[512], buf2[512];

        fd1 = open("/etc/passwd", O_RDONLY);
        fd2 = open("/etc/passwd", O_RDONLY);
        read(fd1, buf1, sizeof(buf1));
        read(fd2, buf2, sizeof(buf2));
}
```

**Figure 5.9.** Reading a File via Two File Descriptors

have to allocate several blocks for use as indirect blocks and data blocks. The inode is locked for the duration of the *write*, because the kernel may change the inode when allocating new blocks; allowing other processes access to the file could corrupt the inode if several processes allocate blocks simultaneously for the same byte offsets. When the write is complete, the kernel updates the file size entry in the inode if the file has grown larger.

For example, suppose a process writes byte number 10,240 to a file, the highest-numbered byte yet written to the file. When accessing the byte in the file using algorithm *bmap*, the kernel will find not only that the file does not contain a block for that byte but also that it does not contain the necessary indirect block. It assigns a disk block for the indirect block and writes the block number in the in-core inode. Then it assigns a disk block for the data block and writes its block number into the first position in the newly assigned indirect block.

The kernel goes through an internal loop, as in the *read* algorithm, writing one block to disk during each iteration. During each iteration, it determines whether it will write the entire block or only part of it. If it writes only part of a block, it must first read the block from disk so as not to overwrite the parts that will remain the same, but if it writes the whole block, it need not read the block, since it will overwrite its previous contents anyway. The write proceeds block by block, but the kernel uses a *delayed write* (Section 3.4) to write the data to disk, caching it in case another process should *read* or *write* it soon and avoiding extra disk operations. Delayed write is probably most effective for pipes, because another process is reading the pipe and removing its data (Section 5.12). But even for regular files, delayed write is effective if the file is created temporarily and will be read soon. For example, many programs, such as editors and mail, create temporary files in the directory "/tmp" and quickly remove them. Use of delayed write can reduce

the number of disk writes for temporary files.

## 5.4 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" (see [Ritchie 81]). 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 *read*ing or *writ*ing any part of an entire file, and record locking is the capability to prevent other processes from *read*ing or *writ*ing particular records (parts of a file between particular byte offsets). Exercise 5.9 explores the implementation of file and record locking.

## 5.5 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. In the program in Figure 5.10, for example, a process *open*s a file, *read*s a byte, then invokes *lseek* to advance the file table offset value by 1023 (with *reference* 1), and loops. Thus, the program *read*s every 1024th byte of the file. If the value of *reference* is 0, the kernel seeks from the beginning of the file, and if its value is 2, the kernel seeks beyond the end of the file. The *lseek* system call has nothing to do with the seek operation that positions a disk arm over a particular disk sector. To implement *lseek*, the kernel simply adjusts the offset value in the file table; subsequent *read* or *write* system calls use the file table offset as their starting byte offset.

## 5.6 CLOSE

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

```
#include <fcntl.h>
main(argc, argv)
        int argc;
        char *argv[];
{
        int fd, skval;
        char c;

        if (argc != 2)
                exit();
        fd = open(argv[1], O_RDONLY);
        if (fd == -1)
                exit();
        while ((skval = read(fd, &c, 1)) == 1)
        {
                printf("char %c\n", c);
                skval = lseek(fd, 1023L, 1);
                printf("new seek val %d\n", skval);
        }
}
```

**Figure 5.10.** Program with Lseek System Call

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. If the file table reference count is 1, the kernel frees the entry and releases the in-core inode originally allocated in the *open* system call (algorithm *iput*). If other processes still reference the inode, the kernel decrements the inode reference count but leaves it allocated; otherwise, the inode is free for reallocation because its reference count is 0. 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. When a process *exits*, the kernel examines its active user file descriptors and internally *closes* each one. Hence, no process can keep a file open after it terminates.

For example, Figure 5.11 shows the relevant table entries of Figure 5.4, after the second process *closes* its files. The entries for file descriptors 3 and 4 in the user file descriptor table are empty. The count fields of the file table entries are now 0, and the entries are empty. The inode reference count for the files "/etc/passwd" and "private" are also decremented. The inode entry for "private" is on the free list because its reference count is 0, but its entry is not empty. If

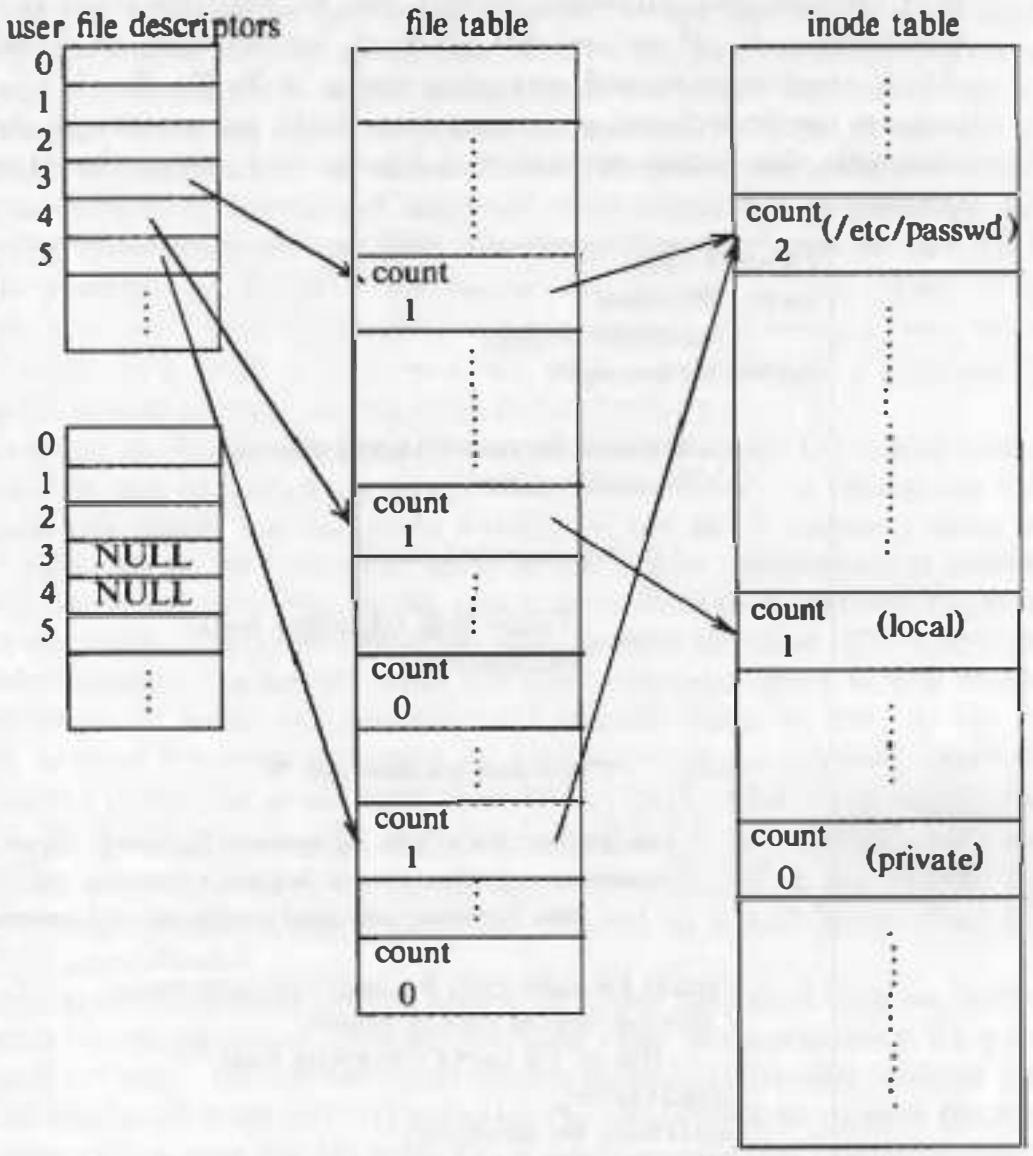



**Figure 5.11.** Tables after Closing a File

another process accesses the file "private" while the inode is still on the free list, the kernel will reclaim the inode, as explained in Section 4.1.2.

## 5.7 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.[3] Figure 5.12 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 5.12.** Algorithm for Creating a File

The kernel parses the path name using algorithm *namei*, following the algorithm literally while parsing directory names. However, when it arrives at the last component of the path name, namely, the file name that it will create, *namei*

3. The *open* system call specifies two flags, *O_CREAT* (create) and *O_TRUNC* (truncate): If a process specifies the *O_CREAT* flag on an *open* and the file does not exist, the kernel will create the file. If the file already exists, it will not be truncated unless the *O_TRUNC* flag is also set.

notes the byte offset of the first empty directory slot in the directory and saves the offset in the *u area*. If the kernel does not find the path name component in the directory, it will eventually write the name into the empty slot just found. If the directory has no empty slots, the kernel remembers the offset of the end of the directory and creates a new slot there. It also remembers the inode of the directory being searched in its *u area* and keeps the inode locked; the directory will become the parent directory of the new file. The kernel does not write the new file name into the directory yet, so that it has less to undo in event of later errors. It checks that the directory allows the process write permission: Because a process will write the directory as a result of the *creat* call, write permission for a directory means that processes are allowed to create files in the directory.

Assuming no file by the given name previously existed, the kernel assigns an inode for the new file, using algorithm *ialloc* (Section 4.6). It then writes the new file name component and the inode number of the newly allocated inode in the parent directory, at the byte offset saved in the *u area*. Afterwards, it releases the inode of the parent directory, having held it from the time it searched the directory for the file name. The parent directory now contains the name of the new file and its inode number. The kernel writes the newly allocated inode to disk (algorithm *bwrite*) before it writes the directory with the new name to disk. If the system crashes between the write operations for the inode and the directory, there will be an allocated inode that is not referenced by any path name in the system but the system will function normally. If, on the other hand, the directory were written before the newly allocated inode and the system crashed in the middle, the file system would contain a path name that referred to a bad inode. (See Section 5.16.1 for more detail.)

If the given file already existed before the *creat*, the kernel finds its inode while searching for the file name. The old file must allow write permission for a process to create a "new" file by the same name, because the kernel changes the file contents during the *creat* call: It truncates the file, freeing all its data blocks using algorithm *free*, so that the file looks like a newly created file. However, the owner and permission modes of the file are the same as they were for the original file: The kernel does not reassign ownership to the owner of the process, and it ignores the permission modes specified by the process. Finally, the kernel does not check that the parent directory of the existing file allows write permission, because it will not change the directory contents.

The *creat* system call proceeds according to the same algorithm as the *open* system call. The kernel allocates an entry in the file table for the created file so that the process can *write* the file, allocates an entry in the user file descriptor table, and eventually returns the index to the latter entry as the user file descriptor.

## 5.8 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 (Chapter 10). Figure 5.13 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 5.13.** Algorithm for Making New Node

The kernel searches the file system for the file name it is about to create. If the file does not yet exist, the kernel assigns a new inode on the disk and writes the new file name and inode number into the parent directory. It sets the file type field in the inode to indicate that the file type is a pipe, directory or special file. Finally, if the file is a *character special* or *block special* device file, it writes the major and minor device numbers into the inode. If the *mknod* call is creating a directory node, the node will exist after the system call completes but its contents will be in the wrong format (there are no directory entries for "." and ".."). Exercise 5.33 considers the other steps needed to put a directory into the correct format.

```
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 5.14.** Algorithm for Changing Current Directory

## 5.9 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 5.14) changes the current directory of a process. The syntax for the *chdir* system call is

```
chdir(pathname);
```

where *pathname* is the directory that becomes the new current directory of the process. The kernel parses the name of the target directory using algorithm *namei* and checks that the target file is a directory and that the process owner has access permission to the directory. It releases the lock to the new inode but keeps the inode allocated and its reference count incremented, releases the inode of the old current directory (algorithm *iput*) stored in the *u area*, and stores the new inode in the *u area*. After a process changes its current directory, algorithm *namei* uses the inode for the start directory to search for all path names that do not begin from root. After execution of the *chdir* system call, the inode reference count of the new directory is at least one, and the inode reference count of the previous current directory may be 0. In this respect, *chdir* is similar to the *open* system call, because both system calls access a file and leave its inode allocated. The inode allocated during the *chdir* system call is released only when the process executes another *chdir* call or when it *exits*.

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. It stores the new root inode in the process *u area*, unlocking the inode on completion of the system call. However, since the default root for the kernel is stored in a global variable, it does not release the inode of the old root automatically, but only if it or an ancestor process had executed the *chroot* system call. The new inode is now the logical root of the file system for the process (and all its children), meaning that all path name searches in algorithm *namei* that start from root ("/") start from this inode, and that all attempts to use ".." over the root will leave the working directory of the process in the new root. A process bestows new child processes with its changed root, just as it bestows them with its current directory.

## 5.10 CHANGE OWNER AND CHANGE MODE

Changing the owner or mode (access permissions) of a file are operations on the inode, not on the file per se. 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 superuser 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 (see Section 7.5), 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.

## 5.11 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* is 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*. The program in Figure 5.33 will illustrate the use of *stat* and *fstat*.

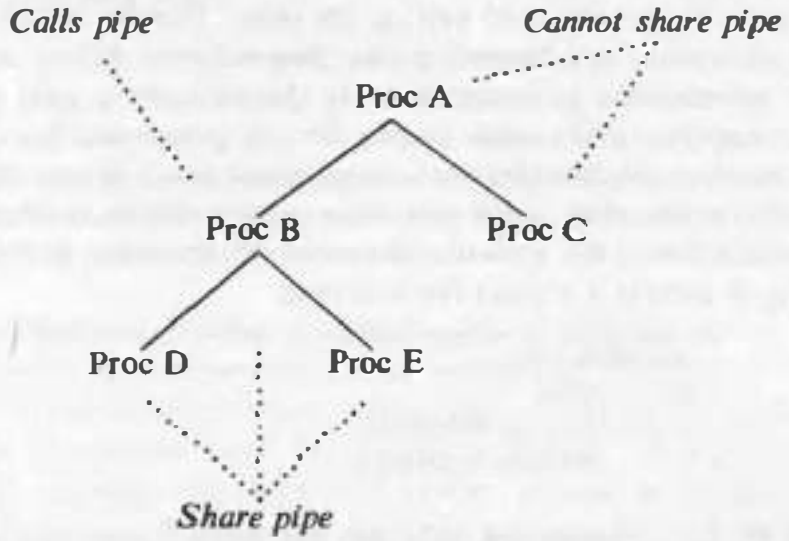



**Figure 5.15.** Process Tree and Sharing Pipes

## 5.12 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 5.15 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.

Because unnamed pipes are more common, they will be presented first.

### 5.12.1 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 *read*ing and *writ*ing the pipe. Because the kernel implements pipes in the file system and because a pipe does not exist before its use, the kernel must assign an inode for it on creation. It also allocates a pair of user file descriptors and corresponding file table entries for the pipe: one file descriptor for *read*ing from the pipe and the other for *writ*ing to the pipe. It uses the file table so that the interface for the *read*, *write* and other system calls is consistent with the interface for regular files. As a result, processes do not have to know whether they are *read*ing or *writ*ing a regular file or a pipe.

```
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 5.16.** Algorithm for Creation of (Unnamed) Pipes

Figure 5.16 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. System administrators specify a pipe device during system configuration, and it may be identical to another file system. While a pipe is active, the kernel cannot reassign the pipe inode and data blocks to another file.

The kernel then allocates two file table entries for the read and write descriptors, respectively, and updates the bookkeeping information in the in-core inode. Each file table entry records how many instances of the pipe are open for reading or writing, initially 1 for each file table entry, and the inode reference

count indicates how many times the pipe was "opened," initially two — one for each file table entry. Finally, the inode records byte offsets in the pipe where the next read or write of the pipe will start. Maintaining the byte offsets in the inode allows convenient FIFO access to the pipe data and differs from regular files where the offset is maintained in the file table. Processes cannot adjust them via the *lseek* system call and so random access I/O to a pipe is not possible.

### 5.12.2 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. A process that *opens* the named pipe for reading will sleep until another process opens the named pipe for writing, and vice versa. It makes no sense for a pipe to be open for reading if there is no hope for it to receive data; the same is true for writing. Depending on whether the process *opens* the named pipe for reading or writing, the kernel awakens other processes that were asleep, waiting for a writer or reader process (respectively) on the named pipe.

If a process *opens* a named pipe for reading and a writing process exists, the *open* call completes. Or if a process *opens* a named pipe with the *no delay* option, the *open* returns immediately, even if there are no writing processes. But if neither condition is true, the process sleeps until a writer process *opens* the pipe. Similar rules hold for a process that *opens* a pipe for writing.

### 5.12.3 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 *read*ing from a pipe do not necessarily equal the number of processes *writ*ing the pipe; if the number of readers or writers is greater than 1, they must coordinate use of the pipe with other mechanisms. 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

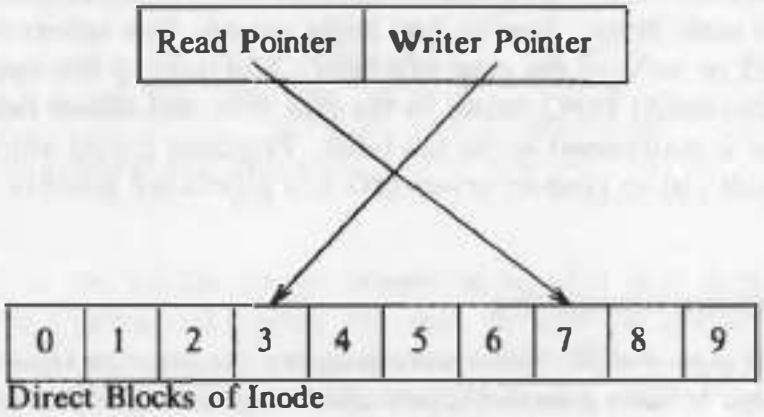



**Figure 5.17.** Logical View of Reading and Writing a Pipe

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 5.17).

Consider four cases for *read*ing and *writ*ing pipes: *writ*ing a pipe that has room for the data being written, *read*ing from a pipe that contains enough data to satisfy the *read*, *read*ing from a pipe that does not contain enough data to satisfy the *read*, and finally, *writ*ing a pipe that does not have room for the data being written.

Consider first the case that a process is writing a pipe and assume that the pipe has room for the data being written: The sum of the number of bytes being written and the number of bytes already in the pipe is less than or equal to the pipe's capacity. The kernel follows the algorithm for writing a regular file, except that it increments the pipe size automatically after every *write*, since by definition the amount of data in the pipe grows with every *write*. This differs from the growth of a regular file where the process increments the file size only when it *write*s data beyond the current end of file. If the next byte offset in the pipe were to require use of an indirect block, the kernel adjusts the file offset value in the *u area* to point to the beginning of the pipe (byte offset 0). The kernel never overwrites data in the pipe; it can reset the byte offset to 0 because it has already determined that the data will not overflow the pipe's capacity. When the writer process has written all its data into the pipe, the kernel updates the pipe's (inode) write pointer so that the next process to *write* the pipe will proceed from where the last *write* stopped. The kernel then awakens all other processes that fell asleep waiting to read data from the pipe.

When a process *read*s a pipe, it checks if the pipe is empty or not. If the pipe contains data, the kernel *read*s the data from the pipe as if the pipe were a regular file, following the regular algorithm for *read*. However, its initial offset is the pipe

read pointer stored in the inode, indicating the extent of the previous *read*. After *read*ing each block, the kernel decrements the size of the pipe according to the number of bytes it read, and it adjusts the *u area* offset value to wrap around to the beginning of the pipe, if necessary. When the *read* system call completes, the kernel awakens all sleeping writer processes and saves the current read offset in the inode (not in the file table entry).

If a process attempts to *read* more data than is in the pipe, the *read* will complete successfully after returning all data currently in the pipe, even though it does not satisfy the user count. If the pipe is empty, the process will typically sleep until another process *write*s data into the pipe, at which time all sleeping processes that were waiting for data wake up and race to *read* the pipe. If, however, a process *open*s a named pipe with the *no delay* option, it will return immediately from a *read* if the pipe contains no data. The semantics of reading and writing pipes are similar to the semantics of reading and writing terminal devices (Chapter 10), allowing programs to ignore the type of file they are dealing with.

If a process *write*s a pipe and the pipe cannot hold all the data, the kernel marks the inode and goes to sleep waiting for data to drain from the pipe. When another process subsequently *read*s from the pipe, the kernel will notice that processes are asleep waiting for data to drain from the pipe, and it will awaken them, as explained above. The exception to this statement is when a process *write*s an amount of data greater than the pipe capacity (that is, the amount of data that can be stored in the inode direct blocks); here, the kernel *write*s as much data as possible to the pipe and puts the process to sleep until more room becomes available. Thus, it is possible that written data will not be contiguous in the pipe if other processes write their data to the pipe before this process resumes its write.

Analyzing the implementation of pipes, the process interface is consistent with that of regular files, but the implementation differs because the kernel stores the read and write offsets in the inode instead of in the file table. The kernel must store the offsets in the inode for named pipes so that processes can share their values: They cannot share values stored in file table entries because a process gets a new file table entry for each *open* call. However, the sharing of read and write offsets in the inode predates the implementation of named pipes. Processes with access to unnamed pipes share access to the pipe through common file table entries, so they could conceivably store the read and write offsets in the file table entry, as is done for regular files. This was not done, because the low-level routines in the kernel no longer have access to the file table entry: The code is simpler because the processes share offsets stored in the inode.

### 5.12.4 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 decrements 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 (Chapter 7) to indicate an error condition. In both cases, it makes no sense to allow the processes to continue sleeping when there is no hope that the state of the pipe will ever change. For example, if a process is waiting to read an unnamed pipe and there are no more writer processes, there will never be a writer process. Although it *is* possible to get new reader or writer processes for named pipes, the kernel treats them consistently with unnamed pipes. If no reader or writer processes access the pipe, the kernel frees all its data blocks and adjusts the inode to indicate that the pipe is empty. When it releases the inode of an ordinary pipe, it frees the disk copy for reassignment.

```
char string[] = "hello";
main()
{
        char buf[1024];
        char *cp1, *cp2;
        int fds[2];

        cp1 = string;
        cp2 = buf;
        while (*cp1)
                *cp2++ = *cp1++;
        pipe(fds);
        for (;;)
        {
                write(fds[1], buf, 6);
                read(fds[0], buf, 6);
        }
}
```

**Figure 5.18.** Reading and Writing a Pipe

### 5.12.5 Examples

The program in Figure 5.18 illustrates an artificial use of pipes. The process creates a pipe and goes into an infinite loop, *writ*ing the string "hello" to the pipe and *read*ing it from the pipe. The kernel does not know nor does it care that the process that writes the pipe is the same process that reads the pipe.

A process executing the program in Figure 5.19 creates a named pipe node called "fifo". If invoked with a second (dummy) argument, it continually *write*s

```
#include <fcntl.h>
char string[] = "hello";
main(argc, argv)
        int argc;
        char *argv[];
{
        int fd;
        char buf[256];

        /* create named pipe with read/write permission for all users */
        mknod("fifo", 010777, 0);
        if (argc == 2)
                fd = open("fifo", O_WRONLY);
        else
                fd = open("fifo", O_RDONLY);
        for (;;)
                if (argc == 2)
                        write(fd, string, 6);
                else
                        read(fd, buf, 6);
}
```

**Figure 5.19.** Reading and Writing a Named Pipe

the string "hello" into the pipe; if invoked without a second argument, it *read*s the named pipe. The two processes are invocations of the identical program and have secretly agreed to communicate through the named pipe "fifo", but they need not be related. Other users could execute the program and participate in (or interfere with) the conversation.

## 5.13 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 *dup*ed 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. For example, examination of the data structures depicted in Figure 5.20 indicates that the process did the following sequence of system calls: It *open*ed the file "/etc/passwd" (file descriptor 3), then *open*ed the file "local" (file descriptor 4), *open*ed the file "/etc/passwd" again (file descriptor 5), and finally,

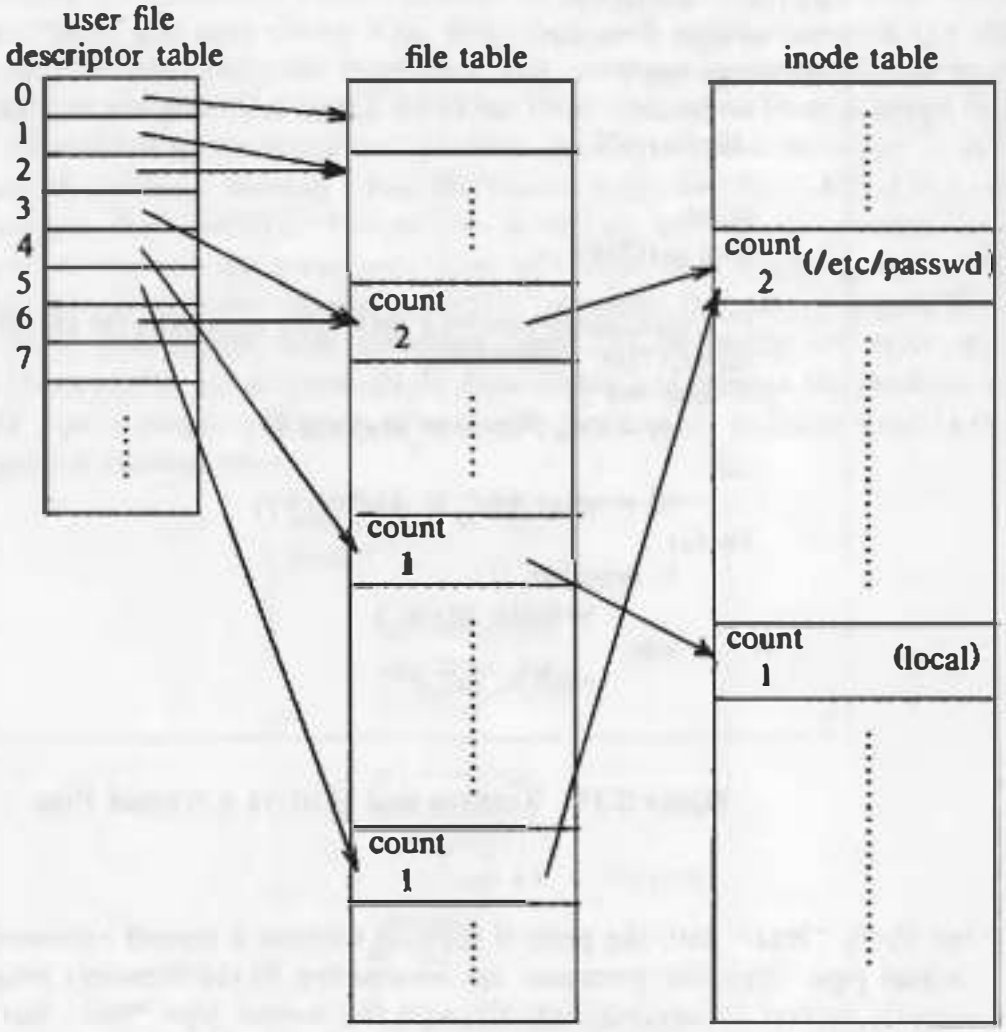



**Figure 5.20.** Data Structures after Dup

*dup*ed file descriptor 3, returning file descriptor 6.

*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 (Chapter 7).

Consider the program in Figure 5.21. The variable *i* contains the file descriptor that the system returns as a result of opening the file "etc/passwd," and the variable *j* contains the file descriptor that the system returns as a result of *dup*ing the file descriptor *i*. In the *u area* of the process, the two user file descriptor entries represented by the user variables *i* and *j* point to one file table entry and therefore use the same file offset. The first two *read*s in the process thus read the data in sequence, and the two buffers, *buf1* and *buf2*, do not contain the same data.

```
#include <fcntl.h>
main()
{
        int i, j;
        char buf1[512], buf2[512];

        i = open("/etc/passwd", O_RDONLY);
        j = dup(i);
        read(i, buf1, sizeof(buf1));
        read(j, buf2, sizeof(buf2));
        close(i);
        read(j, buf2, sizeof(buf2));
}
```

**Figure 5.21.** C Program Illustrating Dup

This differs from the case where a process *opens* the same file twice and *reads* the same data twice (Section 5.2). A process can *close* either file descriptor if it wants, but I/O continues normally on the other file descriptor, as illustrated in the example. In particular, a process can *close* its standard output file descriptor (file descriptor 1), *dup* another file descriptor so that it becomes file descriptor 1, then treat the file as its standard output. Chapter 7 presents a more realistic example of the use of *pipe* and *dup* when it describes the implementation of the shell.

## 5.14 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 *open*ing the appropriate device file name and then *read*ing and *writ*ing the "file," treating it as a sequence of disk blocks. Chapter 10 gives details on this interface. A section of a disk may contain a logical file system, consisting of a boot block, super block, inode list, and data blocks, as described in Chapter 2. 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"

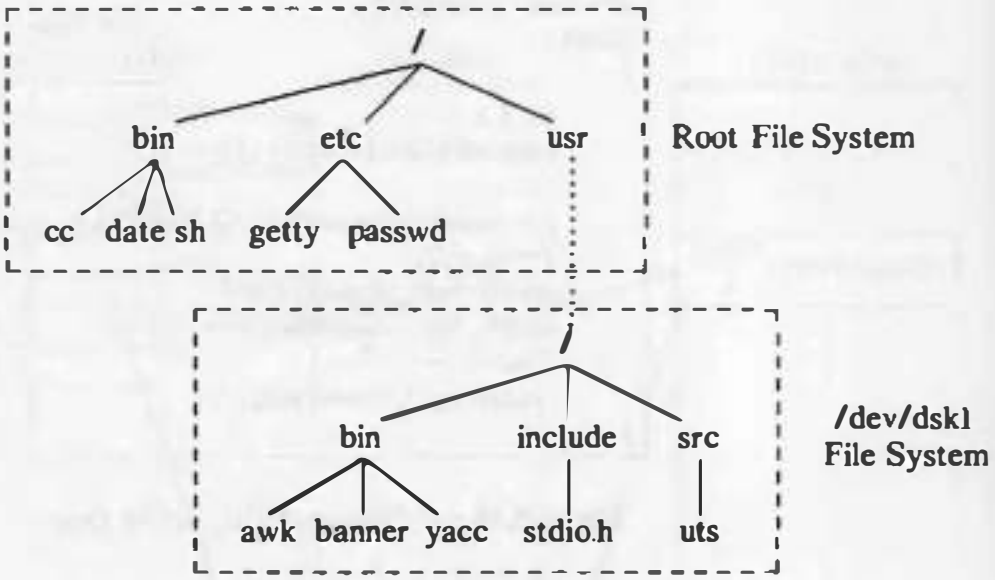



**Figure 5.22.** File System Tree Before and After Mount

(system calls such as *write* and *creat* that write the file system will fail). For example, if a process issues the system call

```
mount("/dev/dsk1", "/usr", 0);
```

the kernel attaches the file system contained in the portion of the disk called "/dev/dsk1" to directory "/usr" in the existing file system tree (see Figure 5.22). The file "/dev/dsk1" is a block special file, meaning that it is the name of a block device, typically a portion of a disk. The kernel assumes that the indicated portion of the disk contains a file system with a super block, inode list, and root inode. After completion of the *mount* system call, the root of the mounted file system is accessed by the name "/usr". Processes can access files on the mounted file system and ignore the fact that it is detachable. Only the *link* system call checks the file system of a file, because System V does not allow file links to span multiple file systems (see Section 5.15).

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 ("/" of the "/dev/dsk1" file system in Figure 5.22);
- a pointer to the inode of the directory that is the mount point ("usr" of the root file system in Figure 5.22).

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

```
algorithm mount
inputs:  file name of 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;
}
```

**Figure 5.23.** Algorithm for Mounting a File System

Figure 5.23 depicts the algorithm for mounting a file system. The kernel only allows processes owned by a superuser to *mount* or *umount* file systems. Yielding permission for *mount* and *umount* to the entire user community would allow malicious (or not so malicious) users to wreak havoc on the file system. Superusers should wreak havoc only by accident.

The kernel finds the inode of the special file that represents the file system to be mounted, extracts the major and minor numbers that identify the appropriate disk section, and finds the inode of the directory on which the file system will be mounted. The reference count of the directory inode must not be greater than 1 (it must be at least 1 — why?), because of potentially dangerous side effects (see exercise 5.27). The kernel then allocates a free slot in the mount table, marks the slot in use, and assigns the device number field in the mount table. The above

assignments are done immediately because the calling process could go to sleep in the ensuing device *open* procedure or in reading the file system super block, and another process could attempt to *mount* a file system. By having marked the mount table entry in use, the kernel prevents two *mount*s from using the same entry. By noting the device number of the attempted *mount*, the kernel can prevent other processes from *mount*ing the same file system again, because strange things could happen if a double mount were allowed (see exercise 5.26).

The kernel calls the *open* procedure for the block device containing the file system in the same way it invokes the procedure when opening the block device directly (Chapter 10). The device *open* procedure typically checks that the device is legal, sometimes initializing driver data structures and sending initialization commands to the hardware. The kernel then allocates a free buffer from the buffer pool (a variation of algorithm *getblk*) to hold the super block of the mounted file system and reads the super block using a variation of algorithm *read*. The kernel stores a pointer to the inode of the mounted-on directory of the original file tree to allow file path names containing ".." to traverse the mount point, as will be seen. It finds the root inode of the *mount*ed file system and stores a pointer to the inode in the mount table. To the user, the mounted-on directory and the root of the mounted file system are logically equivalent, and the kernel establishes their equivalence by their coexistence in the mount table entry. Processes can no longer access the inode of the mounted-on directory.

The kernel initializes fields in the file system super block, clearing the lock fields for the free block list and free inode list and setting the number of free inodes in the super block to 0. The purpose of the initializations is to minimize the danger of file system corruption when mounting the file system after a system crash: Making the kernel think that there are no free inodes in the super block forces algorithm *ialloc* to search the disk for free inodes. Unfortunately, if the linked list of free disk blocks is corrupt, the kernel does not fix the list internally (see Section 5.17 for file system maintenance). If the user *mount*s the file system *read-only* to disallow all write operations to the file system, the kernel sets a flag in the super block. Finally, the kernel marks the mounted-on inode as a mount point, so other processes can later identify it. Figure 5.24 depicts the various data structures at the conclusion of the *mount* call.

### 5.14.1 Crossing Mount Points in File Path Names

Let us reconsider algorithms *namei* and *iget* for the cases where a path name crosses a mount point. 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. The following sequence of shell commands illustrates the two cases.

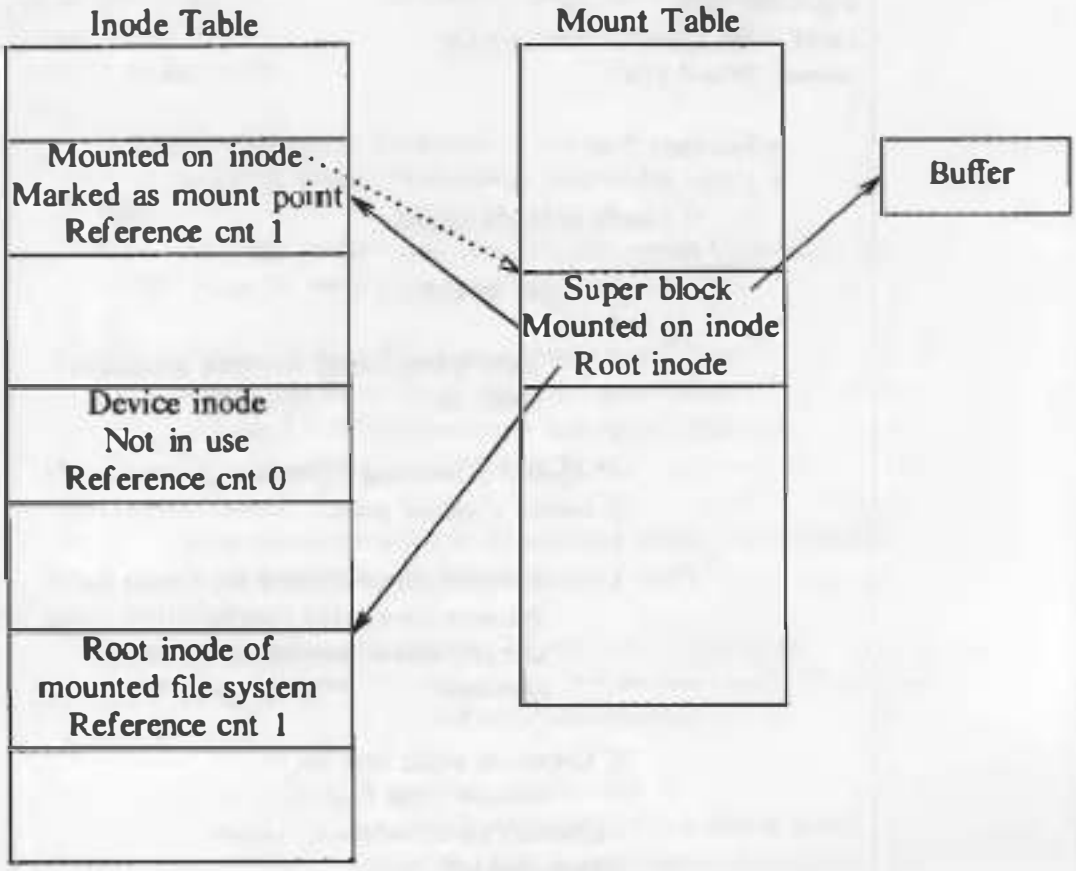



**Figure 5.24.** Data Structures after Mount

```
mount /dev/dsk1 /usr
cd /usr/src/uts
cd ../../..
```

The *mount* command invokes the *mount* system call after doing some consistency checks and mounts the file system in the disk section identified by "/dev/dsk1" onto the directory "/usr". The first *cd* (change directory) command causes the shell to execute the *chdir* system call, and the kernel parses the path name, crossing the mount point at "/usr". The second *cd* command results in the kernel parsing the path name and crossing the mount point at the third ".." in the path name.

For the case of crossing the mount point from the mounted-on file system to the mounted file system, consider the revised algorithm for *iget* in Figure 5.25, which is identical to that of Figure 4.3, except that it checks if the inode is a mount point: If the inode is marked "mounted-on," the kernel knows that it is a mount point. It finds the mount table entry whose mounted-on inode is the one just accessed and notes the device number of the mounted file system. Using the device number and the inode number for root, which is common to all file systems, it then accesses the

```
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 */
                        }
                        /* special processing for mount points---*/
                        if (inode a mount point)
                        {
                                find mount table entry for mount point;
                                get new file system number from mount table;
                                use root inode number in search;
                                continue;       /* loop again */
                        }
                        if (inode on inode free list)
                                remove from free list;
                        increment inode reference count;
                        return (inode);
                }

                /* inode not in inode cache */
                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 5.25.** Revised Algorithm for Accessing an Inode

root inode of the mounted device and returns that inode. In the first change directory example above, the kernel first accesses the inode for "/usr" in the mounted-on file system, finds that the inode is marked "mounted-on," finds the root inode of the mounted file system in the mount table, and accesses the root inode of the mounted file system.

```
algorithm namei                 /* convert path name to inode */
input:  path name
output: locked inode
{
    if (path name starts from root)
        working inode = root inode (algorithm iget);
    else
        working inode = current directory inode (algorithm iget);
    while (there is more path name)
    {
        read next path name component from input;
        verify that inode is of directory, permissions;
        if (inode is of changed root and component is "..")
            continue;       /* loop */
component search:
        read inode (directory) (algorithms bmap, bread, brelse);
        if (component matches a directory entry)
        {
            get inode number for matched component;
            if (found inode of root and working inode is root and
                    and component name is "..")
            {
                /* crossing mount point */
                get mount table entry for working inode;
                release working inode (algorithm iput);
                working inode = mounted on inode;
                lock mounted on inode;
                increment reference count of working inode;
                go to component search (for "..");
            }
            release working inode (algorithm iput);
            working inode = inode for new inode number (algorithm iget);
        }
        else        /* component not in directory */
            return (no inode);
    }
    return (working inode);
}
```

**Figure 5.26.** Revised Algorithm for Parsing a File Name

For the second case of crossing the mount point from the mounted file system to the mounted-on file system, consider the revised algorithm for *namei* in Figure 5.26. It is similar to that of Figure 4.11. However, after finding the inode number for a path name component in a directory, the kernel checks if the inode number is the root inode of a file system. If it is, and if the inode of the current working inode is

also root, and the path name component is dot-dot (".."), the kernel identifies the inode as a mount point. It finds the mount table entry whose device number equals the device number of the last found inode, gets the inode of the mounted-on directory, and continues its search for dot-dot ("..") using the mounted-on inode as the working inode. At the root of the file system, however, ".." is the root.

In the example above (cd "../../.."), assume the starting current directory of the process is "/usr/src/uts". When parsing the path name in *namei*, the starting working inode is the current directory. The kernel changes the working inode to that of "/usr/src" as a result of parsing the first ".." in the path name. Then, it parses the second ".." in the path name, finds the root inode of the (previously) mounted file system, "usr", and makes it the working inode in *namei*. Finally, it parses the third ".." in the path name: It finds that the inode number for ".." is the root inode number, its working inode is the root inode, and ".." is the current path name component. The kernel finds the mount table entry for the "usr" mount point, releases the current working inode (the root of the "usr" file system), and allocates the mounted-on inode (the inode for directory "usr" in the root file system) as the new working inode. It then searches the directory structures in the mounted-on "/usr" for ".." and finds the inode number for the root of the file system ("/"). The *chdir* system call then completes as usual; the calling process is oblivious to the fact that it crossed a mount point.

### 5.14.2 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 (Figure 5.27), 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 (Chapter 7), 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 (see Chapter 7 for detail), writes out all recently modified super blocks to disk, and updates the disk copy of all inodes that need updating. It would suffice for the kernel to update the disk blocks, super block, and inodes for the unmounting file system only, but for

```
algorithm umount
input:  special file name of file system to be unmounted
output: none
{
        if (not super user)
                return(error);
        get inode of special file (algorithm namei);
        extract major, minor number of device being unmounted;
        get mount table entry, based on major, minor number.
                for unmounting file system;
        release inode of special file (algorithm iput);
        remove shared text entries from region table for files
                        belonging to file system; /* chap 7xxx */
        update super block, inodes, flush buffers;
        if (files from file system still in use)
                return(error);
        get root inode of mounted file system from mount table;
        lock inode;
        release inode (algorithm iput);  /* iget was in mount */
        invoke close routine for special device;
        invalidate buffers in pool from unmounted file system;
        get inode of mount point from mount table;
        lock inode;
        clear flag marking it as mount point;
        release inode (algorithm iput);          /* iget in mount */
        free buffer used for super block;
        free mount table slot;
}
```

**Figure 5.27.** Algorithm for Unmounting a File System

historical reasons it does so for all file systems. The kernel then releases the root inode of the mounted file system, held since its original access during the *mount* system call, and invokes the driver of the device that contains the file system to close the device. Afterwards, it goes through the buffers in the buffer cache and invalidates buffers for blocks on the now unmounted file system; there is no need to cache data in those blocks any longer. When invalidating the buffers, it moves the buffers to the beginning of the buffer free list, so that valid blocks remain in the buffer cache longer. It clears the "mounted-on" flag in the mounted-on inode set during the *mount* call and releases the inode. After marking the mount table entry free for general use, the *umount* call completes.

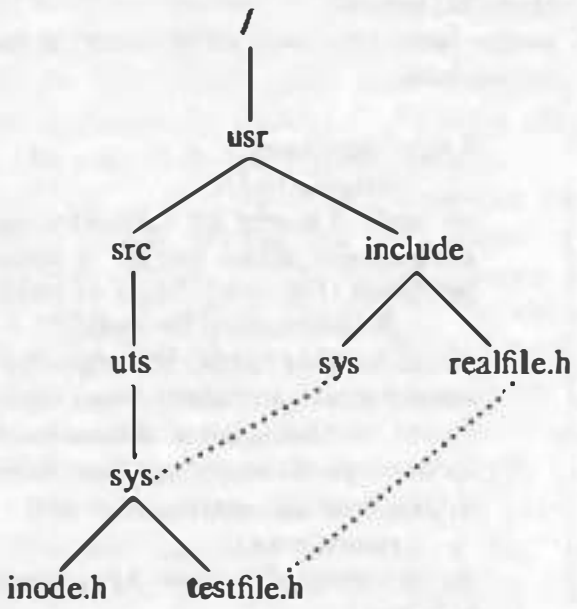



**Figure 5.28.** Linked Files in File System Tree

## 5.15 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. For example, after executing the system calls

link("/usr/src/uts/sys", "/usr/include/sys");
link("/usr/include/realfile.h", "/usr/src/uts/sys/testfile.h");

the following three path names refer to the same file: "/usr/src/uts/sys/testfile.h", "/usr/include/sys/testfile.h", and "/usr/include/realfile" (see Figure 5.28).

The kernel allows only a superuser to *link* directories, simplifying the coding of programs that traverse the file system tree. If arbitrary users could *link* directories, programs designed to traverse the file hierarchy would have to worry about getting into an infinite loop if a user were to *link* a directory to a node name below it in the hierarchy. Superusers are presumably more careful about making such *links*. The capability to link directories had to be supported on early versions of the

system, because the implementation of the *mkdir* command, which creates a new directory, relies on the capability to link directories. Inclusion of the *mkdir* system call eliminates the need to link directories.

```
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);
}
```

**Figure 5.29.** Algorithm for Linking Files

Figure 5.29 shows the algorithm for *link*. The kernel first locates the inode for the source file using algorithm *namei*, increments its link count, updates the disk copy of the inode (for consistency, as will be seen), and unlocks the inode. It then searches for the target file; if the file is present, the *link* call fails, and the kernel decrements the link count incremented earlier. Otherwise, it notes the location of an empty slot in the parent directory of the target file, writes the target file name and the source file inode number into that slot, and releases the inode of the target file parent directory via algorithm *iput*. Since the target file did not originally exist, there is no other inode to release. The kernel concludes by releasing the source file inode: Its link count is 1 greater than it was at the beginning of the call, and another name in the file system allows access to it. The link count keeps count of the directory entries that refer to the file and is thus distinct from the inode

reference count. If no other processes access the file at the conclusion of the *link* call, the inode reference count of the file is 0, and the link count of the file is at least 2.

For example, when executing

```
link("source", "dir/target");
```

the kernel locates the inode for file "source", increments its link count, remembers its inode number, say 74, and unlocks the inode. It locates the inode of "dir", the parent directory of "target", finds an empty directory slot in "dir", and writes the file name "target" and the inode number 74 into the empty directory slot. Finally, it releases the inode for "source" via algorithm *iput*. If the link count of "source" had been 1, it is now 2.

Two deadlock possibilities are worthy of note, both concerning the reason the process unlocks the source file inode after incrementing its link count. If the kernel did not unlock the inode, two processes could deadlock by executing the following system calls simultaneously.

```
process A:    link("a/b/c/d", "e/f/g");
process B:    link("e/f", "a/b/c/d/ee");
```

Suppose process A finds the inode for file "a/b/c/d" at the same time that process B finds the inode for "e/f". The phrase *at the same time* means that the system arrives at a state where each process has allocated its inode. Figure 5.30 illustrates an execution scenario. When process A now attempts to find the inode for directory "e/f", it would sleep awaiting the event that the inode for "f" becomes free. But when process B attempts to find the inode for directory "a/b/c/d", it would sleep awaiting the event that the inode for "d" becomes free. Process A would be holding a locked inode that process B wants, and process B would be holding a locked inode that process A wants. The kernel avoids this classic example of deadlock by releasing the source file's inode after incrementing its link count. Since the first resource (inode) is free when accessing the next resource, no deadlock can occur.

The last example showed how two processes could deadlock each other if the inode lock were not released. A single process could also deadlock itself. If it executed

```
link("a/b/c", "a/b/c/d");
```

it would allocate the inode for file "c" in the first part of the algorithm; if the kernel did not release the inode lock, it would deadlock when encountering the inode "c" in searching for the file "d". If two processes, or even one process, could not continue executing because of deadlock, what would be the effect on the system? Since inodes are finitely allocatable resources, receipt of a signal cannot awaken the process from its sleep (Chapter 7). Hence, the system could not break the deadlock without rebooting. If no other processes accessed the files over which the processes deadlock, no other processes in the system would be affected.

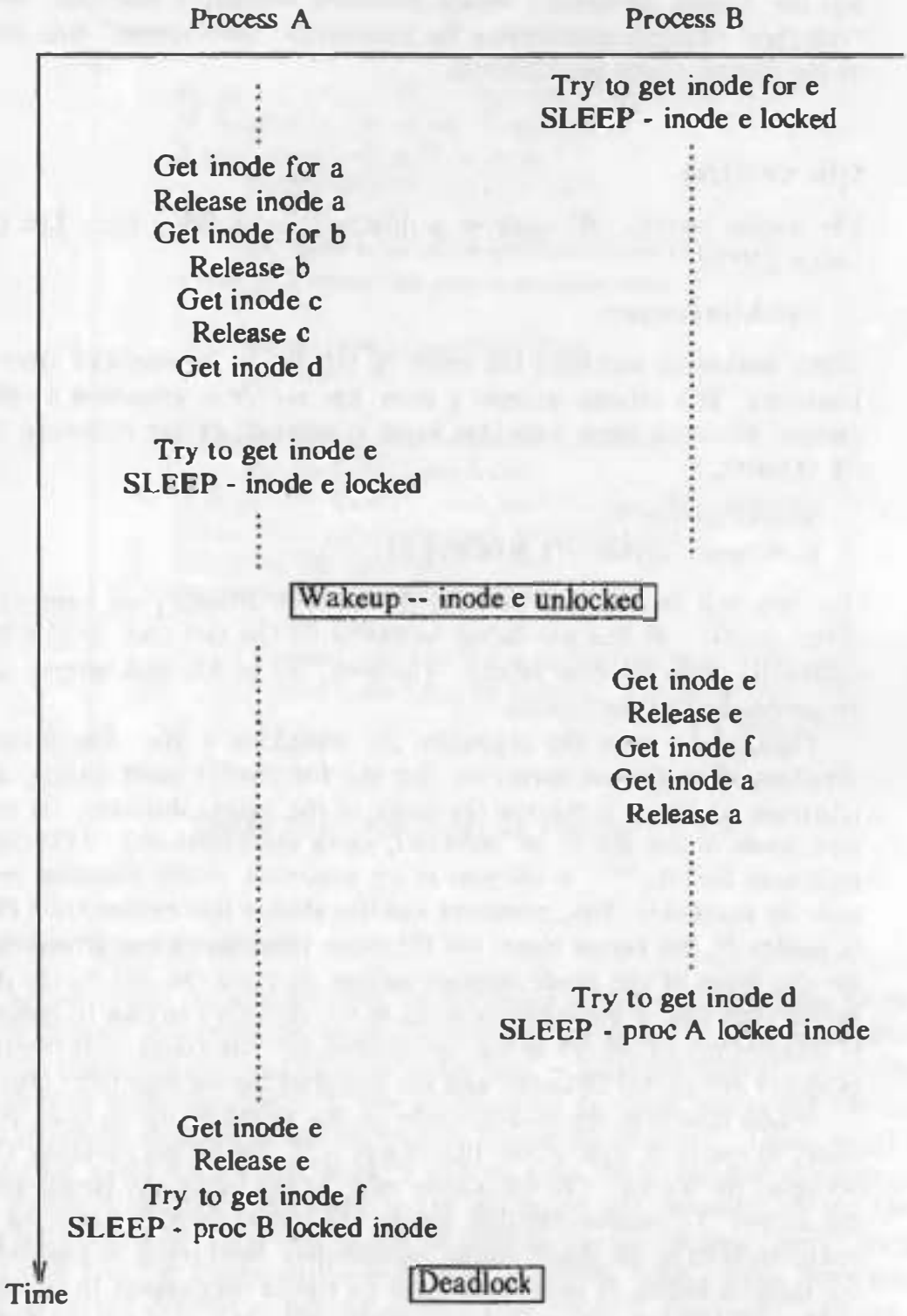



**Figure 5.30.** Deadlock Scenario for Link

However, any processes that accessed those files (or attempted to access other files via the locked directory) would deadlock. Thus, if the file were "/bin" or "/usr/bin" (typical depositories for commands) or "/bin/sh" (the shell) the effect on the system would be disastrous.

## 5.16 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 *unlink*ed from the directory hierarchy. If a process *unlink*s 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 *unlink*ed 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.

Figure 5.31 gives the algorithm for *unlink*ing a file. The kernel first uses a variation of algorithm *namei* to find the file that it must *unlink*, but instead of returning its inode, it returns the inode of the parent directory. It accesses the in-core inode of the file to be *unlink*ed, using algorithm *iget*. (The special case for unlinking the file "." is covered in an exercise.) After checking error conditions and, for executable files, removing inactive shared text entries from the region table (Chapter 7), the kernel clears the file name from the parent directory: Writing a 0 for the value of the inode number suffices to clear the slot in the directory. The kernel then does a synchronous write of the directory to disk to ensure that the file is inaccessible by its old name, decrements the link count, and releases the in-core inodes of the parent directory and the unlinked file via algorithm *iput*.

When releasing the in-core inode of the unlinked file in *iput*, if the reference count drops to 0, and if the link count is 0, the kernel reclaims the disk blocks occupied by the file. No file names refer to the inode any longer and the inode is not active. To reclaim the disk blocks, the kernel loops through the inode table of contents, freeing all direct blocks immediately (according to algorithm *free*). For the indirect blocks, it recursively frees all blocks that appear in the various levels of indirection, freeing the more direct blocks first. It zeroes out the block numbers in the inode table of contents and sets the file size in the inode to 0. It then clears the inode file type field to indicate that the inode is free and frees the inode with algorithm *ifree*. It updates the disk since the disk copy of the inode still indicated that the inode was in use; the inode is now free for assignment to other files.

```
algorithm unlink
input:  file name
output: none
{
        get parent inode of file to be unlinked (algorithm namei);
        /* if unlinking the current directory... */
        if (last component of file name is ".")
                increment inode reference count;
        else
                get inode of file to be unlinked (algorithm iget);
        if (file is directory but user is not super user)
        {
                release inodes (algorithm iput);
                return(error);
        }
        if (shared text file and ilk  count currently 1)
                remove from region table;
        write parent directory: zero inode number of unlinked file;
        release inode parent directory (algorithm iput);
        decrement file link count;
        release file inode (algorithm iput);
                /* iput checks if link count is 0:  if so,
                 * releases file blocks (algorithm free) and
                 * frees inode (algorithm ifree);
                 */
}
```

**Figure 5.31.** Algorithm for Unlinking a File

### 5.16.1 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: There would be an inode that would have a link count 1 greater than the number of directory entries that access it, but all other paths to the file would still be legal. If the directory write were not synchronous, it would be possible for the directory entry on disk to point to a free (or reallocated!) inode after a system crash. Thus there would be more directory entries in the file system that refer to the inode than the inode would have link counts. In particular, if the file name was that of the last link to the file, it would refer to an unallocated inode. System damage is clearly less severe and easier to correct in the first case (see Section 5.18).

For example, suppose a file has two links with path names "a" and "b", and suppose a process *unlink*s "a". If the kernel orders the disk write operations, then it zeros the directory entry for "a" and writes it to disk. If the system crashes after the write to disk completes, file "b" has link count of 2, but file "a" does not exist because its old entry had been zeroed before the system crash. File "b" has an extra link count, but the system functions properly when rebooted.

Now suppose the kernel ordered the disk write operations in the reverse order and the system crashes: That is, it decrements the link count for the file "b" to 1, writes the inode to disk, and crashes before it could zero the directory entry for file "a". When the system is rebooted, entries for files "a" and "b" exist in their respective directories, but the link count for the file they reference is 1. If a process then *unlink*s file "a", the file link count drops to 0 even though file "b" still references the inode. If the kernel were later to reassign the inode as the result of a *creat* system call, the new file would have link count 1 but two path names that reference it. The system cannot rectify the situation except via maintenance programs (*fsck*, described in Section 5.18) that access the file system through the block or raw interface.

The kernel also frees inodes and disk blocks in a specific order to minimize corruption in event of system failure. When removing the contents of a file and clearing its inode, it is possible to free the blocks containing the file data first, or it is possible to free and write out the inode first. The result is usually identical for both cases, but it differs if the system crashes in the middle. Suppose the kernel first frees the disk blocks of a file and crashes. When the system is rebooted, the inode still contains references to the old disk blocks, which may no longer contain data relevant to the file. The kernel would see an apparently good file, but a user accessing the file would notice corruption. It is also possible that other files were assigned those disk blocks. The effort to clean the file system with the *fsck* program would be great. However, if the system first writes the inode to disk and the system crashes, a user would not notice anything wrong with the file system when the system is rebooted. The data blocks that previously belonged to the file would be inaccessible to the system, but users would notice no apparent corruption. The *fsck* program also finds the task of reclaiming unlinked disk blocks easier than the clean-up it would have to do for the first sequence of events.

### 5.16.2 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 *read*s 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. Once a process begins execution of the *unlink* call, however, no other process can access the file being unlinked since the inodes of the parent directory and the file are locked.

Recall the algorithm for the *link* system call and how the kernel unlocks the inode before completion of the call. If another process should *unlink* the file while the inode lock is free, it would only decrement the link count; since the link count had been incremented before unlinking the inode, the count would still be greater than 0. Hence, the file cannot be removed, and the system is safe. The condition is equivalent to the case where the *unlink* happens immediately after the *link* call completes.

Another race condition exists in the case where one process is converting a file path name to an inode using algorithm *namei* and another process is removing a directory in that path. Suppose process A is parsing the path name "a/b/c/d" and goes to sleep while allocating the in-core inode for "c". It could go to sleep while trying to lock the inode or while trying to access the disk block in which the inode resides (see algorithms *iget* and *bread*). If process B wants to *unlink* the directory "c", it may go to sleep, possibly for the same reasons that process A is sleeping. Suppose the kernel later schedules process B to run before process A. Process B would run to completion, unlinking directory "c" and removing it and its contents (for the last link) before process A runs again. Later, process A would try to access an illegal in-core inode that had been removed. Algorithm *namei* therefore checks that the link count is not 0 before proceeding, reporting an error otherwise.

The check is not sufficient, however, because another process could conceivably create a new directory somewhere in the file system and allocate the inode that had previously been used for "c". Process A is tricked into thinking that it accessed the correct inode (see Figure 5.32). Nevertheless, the system maintains its integrity; the worst that could happen is that the wrong file is accessed — a possible security breach — but the race condition is rare in practice.

A process can *unlink* a file while another process has the file open. (The *unlink*ing process could even be the process that did the *open*). Since the kernel unlocks the inode at the end of the *open* call, the *unlink* call will succeed. The kernel will follow the *unlink* algorithm as if the file were not open, and it will remove the directory entry for the file. No other processes will be able to access the now *unlink*ed file. However, since the *open* system call had incremented the inode reference count, the kernel does not clear the file contents when executing the *iput* algorithm at the conclusion of the *unlink* call. So the opening process can do all the normal file operations with its file descriptor, including *read*ing and *writ*ing the file. But when it *close*s the file, the inode reference count drops to 0 in *iput*, and the kernel clears the contents of the file. In short, the process that had *open*ed the file proceeds as if the *unlink* did not occur, and the *unlink* happens as if the file were not open. Other system calls will continue to work for the opening process, too.

In Figure 5.33 for example, a process *open*s a file supplied as a parameter and then *unlink*s the file it just *open*ed. The *stat* call fails because the original path

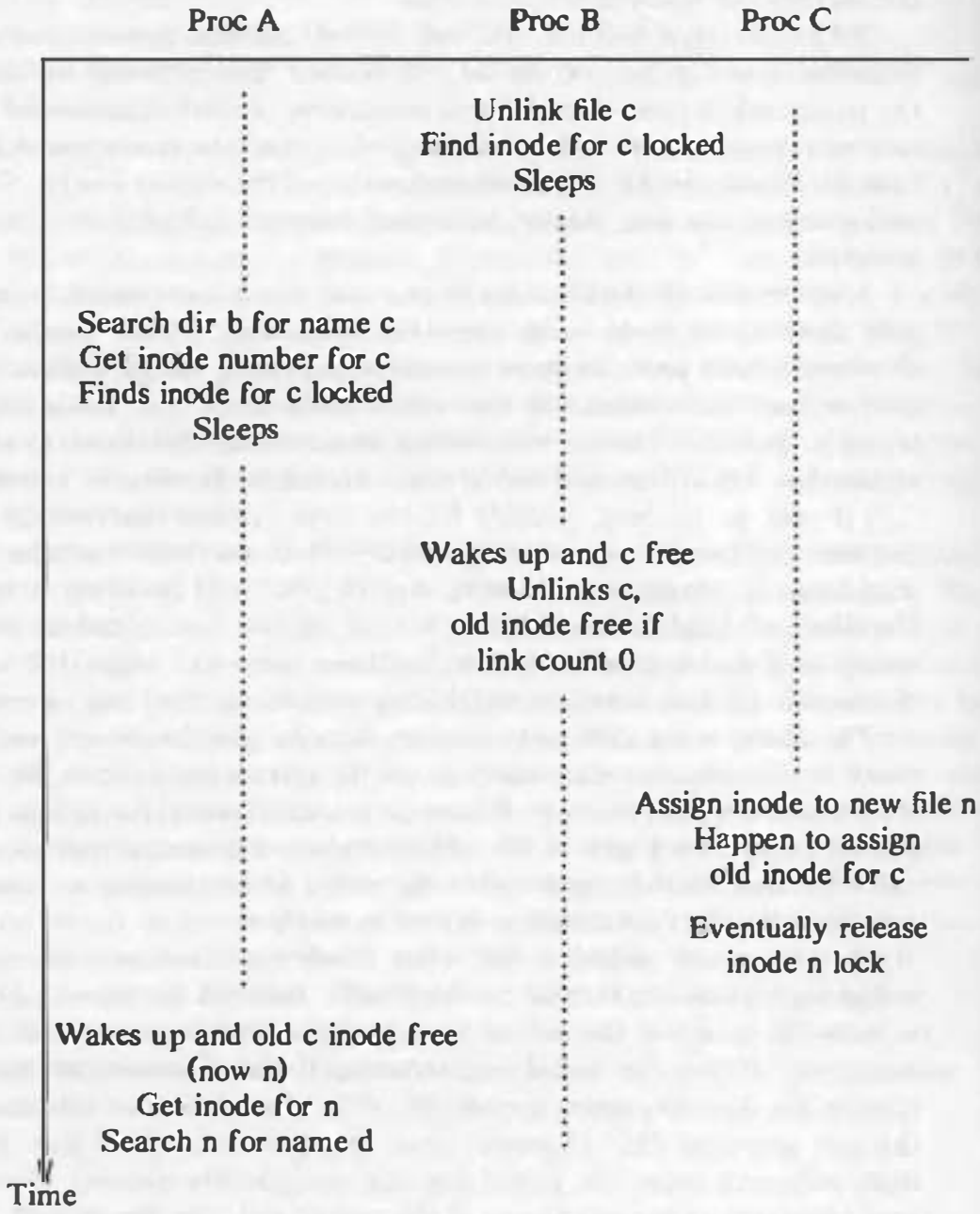



**Figure 5.32.** Unlink Race Condition

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <fcntl.h>

main(argc, argv)
        int argc;
        char *argv[];
{
        int fd;
        char buf[1024];
        struct stat statbuf;

        if (argc != 2)                  /* need a parameter */
                exit();
        fd = open(argv[1], O_RDONLY);
        if (fd == -1)                   /* open fails */
                exit();
        if (unlink(argv[1]) == -1)      /* unlink file just opened */
                exit();
        if (stat(argv[1], &statbuf) == -1)      /* stat the file by name*/
                printf("stat %s fails as it should\n", argv[1]);
        else
                printf("stat %s succeeded!!!!\n", argv[1]);
        if (fstat(fd, &statbuf) == -1)          /* stat the file by fd */
                printf("fstat %s fails!!!\n", argv[1]);
        else
                printf("fstat %s succeeds as it should\n", argv[1]);
        while (read(fd, buf, sizeof(buf)) > 0)  /* read open/unlinked file */
                printf("%1024s", buf);          /* prints 1K byte field */
}
```

**Figure 5.33.** Unlinking an Opened File

name no longer refers to a file after the *unlink* (assuming no other process created a file by that name in the meantime), but the *fstat* call succeeds because it gets to the inode via the file descriptor. The process loops, *read*ing the file 1024 bytes at a time and printing the file to the standard output. When the *read* encounters the end of the file, the process *exit*s: After the close in *exit*, the file no longer exists. Processes commonly create temporary files and immediately unlink them; they can continue to read and write them, but the file name no longer appears in the directory hierarchy. If the process should fail for some reason, it leaves no trail of temporary files behind it.

## 5.17 FILE SYSTEM ABSTRACTIONS

Weinberger introduced *file system types* to support his network file system (see [Killian 84] for a brief description of this mechanism), and the latest release of System V supports a derivation of his scheme. File system types allow the kernel to support multiple file systems simultaneously, such as network file systems (Chapter 13) 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.

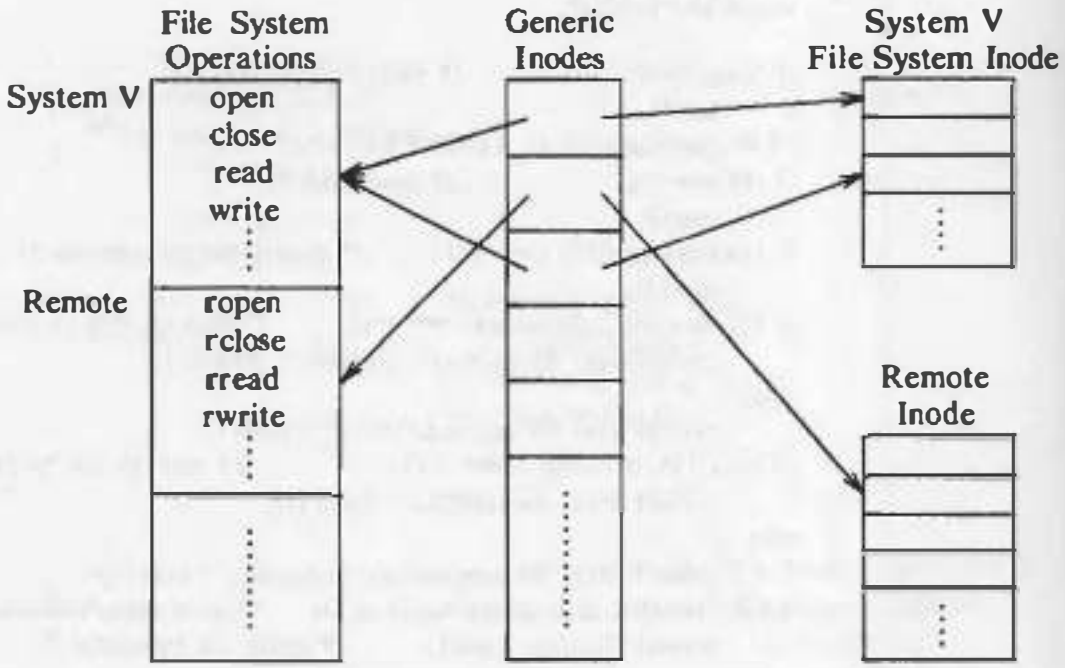



**Figure 5.34.** Inodes for File System Types

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, file type, size, owner, and reference count. Other data that is file-system-specific includes the super block and directory structures. Figure 5.34 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. 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 (see Figure 5.34). 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. Chapter 13 will illustrate the use of file system abstractions in the description of a distributed file system.

## 5.18 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 (Chapter 10) and bypasses the regular file access methods. This section describes several inconsistencies checked by *fsck*.

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.

If a block number is not on the free list of blocks nor contained in a file, the file system is inconsistent because, as mentioned above, all blocks must appear somewhere. This situation could happen if a block was removed from a file and placed on the super block free list. If the old file was written to disk and the system crashed before the super block was written to disk, the block would not appear on any lists stored on disk.

An inode may have a non-0 link count, but its inode number may not exist in any directories in the file system. All files except (unnamed) pipes must exist in the file system tree. If the system crashes after creating a pipe or after creating a file but before creating its directory entry, the inode will have its link field set even though it does not appear to be in the file system. The problem could also arise if a directory were *unlinked* before making sure that all files contained in the directory were *unlinked*.

If the format of an inode is incorrect (for instance, if the file type field has an undefined value), something is wrong. This could happen if an administrator mounted an improperly formatted file system. The kernel accesses disk blocks that it thinks contain inodes but in reality contain data.

If an inode number appears in a directory entry but the inode is free, the file system is inconsistent because an inode number that appears in a directory entry should be that of an allocated inode. This could happen if the kernel was creating a new file and wrote the directory entry to disk but did not write the inode to disk before the crash. It could also occur if a process *unlink*ed a file and wrote the freed inode to disk, but did not write the directory element to disk before it crashed. These situations are avoided by ordering the write operations properly.

If the number of free blocks or free inodes recorded in the super block does not conform to the number that exist on disk, the file system is inconsistent. The summary information in the super block must always be consistent with the state of the file system.

## 5.19 SUMMARY

This chapter concludes the first part of the book, the explanation of the file system. It introduced three kernel tables: the user file descriptor table, the system file table, and the mount table. It described the algorithms for many system calls relating to the file system and their interaction. It introduced file system abstractions, which allow the UNIX system to support varied file system types. Finally, it described how *fsck* checks the consistency of the file system.

## 5.20 EXERCISES

1. Consider the program in Figure 5.35. What is the return value for all the *read*s and what is the contents of the buffer? Describe what is happening in the kernel during each *read*.
2. Reconsider the program in Figure 5.35 but suppose the statement

   ```
   lseek(fd, 9000L, 0);
   ```

   is placed before the first *read*. What does the process see and what happens inside the kernel?
3. A process can *open* a file in write-append mode, meaning that every write operations starts at the byte offset marking the current end of file. Therefore, two processes can *open* a file in write-append mode and write the file without overwriting data. What happens if a process *open*s a file in write-append mode and seeks to the beginning of the file?
4. The standard I/O library makes user reading and writing more efficient by buffering the data in the library and thus potentially saving the number of system calls a user has to make. How would you implement the library functions *fread* and *fwrite*? What should the library functions *fopen* and *fclose* do?

```
#include <fcntl.h>
main()
{
        int fd;
        char buf[1024];
        fd = creat("junk", 0666);
        lseek(fd, 2000L, 2);            /* seek to byte 2000 */
        write(fd, "hello", 5);
        close(fd);

        fd = open("junk", O_RDONLY);
        read(fd, buf, 1024);            /* read zero's */
        read(fd, buf, 1024);            /* catch something */
        read(fd, buf, 1024);
}
```

**Figure 5.35.** Reading 0s and End of File

5. If a process is reading data consecutively from a file, the kernel notes the value of the read-ahead block in the in-core inode. What happens if several processes simultaneously read data consecutively from the same file?

```
#include <fcntl.h>
main()
{
        int fd;
        char buf[256];

        fd = open("/etc/passwd", O_RDONLY);
        if (read(fd, buf, 1024) < 0)
                printf("read fails\n");
}
```

**Figure 5.36.** A Big Read in a Little Buffer

6. Consider the program in Figure 5.36. What happens when the program is executed? Why? What would happen if the declaration of *buf* were sandwiched between the declaration of two other arrays of size 1024? How does the kernel recognize that the *read* is too big for the buffer?

* 7. The BSD file system allows fragmentation of the last block of a file as needed, according to the following rules:

   - Structures similar to the super block keep track of free fragments;
   - The kernel does not keep a preallocated pool of free fragments but breaks a free block into fragments when necessary;

- The kernel can assign block fragments only for the last block of a file;
- If a block is partitioned into several fragments, the kernel can assign them to different files;
- The number of fragments in a block is fixed per file system;
- The kernel allocates fragments during the *write* system call.

Design an algorithm that allocates block fragments to a file. What changes must be made to the inode to allow for fragments? How advantageous is it from a performance standpoint to use fragments for files that use indirect blocks? Would it be more advantageous to allocate fragments during a *close* call instead of during a *write* call?

* 8. Recall the discussion in Chapter 4 for placing data in a file's inode. If the size of the inode is that of a disk block, design an algorithm such that the last data of a file is written in the inode block if it fits. Compare this method with that described in the previous problem.

* 9. System V uses the *fcntl* system call to implement file and record locking:

```
fcntl(fd, cmd, arg);
```

where *fd* is the file descriptor, *cmd* specifies the type of locking operation, and *arg* specifies various parameters, such as lock type (read or write) and byte offsets (see the appendix). The locking operations include

- Test for locks belonging to other processes and return immediately, indicating whether other locks were found,
- Set a lock and sleep until successful,
- Set a lock but return immediately if unsuccessful.

The kernel automatically releases locks set by a process when it *closes* the file. Describe an algorithm that implements file and record locking. If the locks are *mandatory*, other processes should be prevented from accessing the file. What changes must be made to *read* and *write*?

* 10. If a process goes to sleep while waiting for a file lock to become free, the possibility for deadlock exists: process A may lock file "one" and attempt to lock file "two," and process B may lock file "two" and attempt to lock file "one." Both processes are in a state where they cannot continue. Extend the algorithm of the previous problem so that the kernel detects the deadlock situation as it is about to occur and fails the system call. Is the kernel the right place to check for deadlocks?

11. Before the existence of a file locking system call, users could get cooperating processes to implement a locking mechanism by executing system calls that exhibited atomic features. What system calls described in this chapter could be used? What are the dangers inherent in using such methods?

12. Ritchie claims (see [Ritchie 81]) that file locking is not sufficient to prevent the confusion caused by programs such as editors that make a copy of a file while editing and then write the original file when done. Explain what he meant and comment.

13. Consider another method for locking files to prevent destructive update: Suppose the inode contains a new permission setting such that it allows only one process at a time to *open* the file for writing, but many processes can *open* the file for reading. Describe an implementation.

* 14. Consider the program in Figure 5.37 that creates a directory node in the wrong format (there are no directory entries for "." and ".."). Try a few commands on the new directory such as *ls −l*, *ls −ld*, or *cd*. What is happening?

```
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 2)
        {
                printf("try: command directory name\n");
                exit();
        }

        /* modes indicate: directory (04) rwx permission for all */
        /* only super user can do this */
        if (mknod(argv[1], 040777, 0) == -1)
                printf("mknod fails\n");
}
```

**Figure 5.37.** A Half-Baked Directory

15. Write a program that prints the owner, file type, access permissions, and access times of files supplied as parameters. If a file (parameter) is a directory, the program should *read* the directory and print the above information for all files in the directory.
16. Suppose a directory has read permission for a user but not execute permission. What happens when the directory is used as a parameter to *ls* with the "−i" option? What about the "−l" option? Explain the answers. Repeat the problem for the case that the directory has execute permission but not read permission.
17. Compare the permissions a process must have for the following operations and comment.
    - Creating a new file requires write permission in a directory.
    - Creating an existing file requires write permission on the file.
    - Unlinking a file requires write permission in the directory, not on the file.

* 18. Write a program that visits every directory, starting with the current directory. How should it handle loops in the directory hierarchy?

19. Execute the program in Figure 5.38 and describe what happens in the kernel. (Hint: Execute *pwd* when the program completes.)
20. Write a program that changes its root to a particular directory, and investigate the directory tree accessible to that program.
21. Why can't a process undo a previous *chroot* system call? Change the implementation so that it can change its root back to a previous root. What are the advantages and disadvantages of such a feature?
22. Consider the simple pipe example in Figure 5.19, where a process *writes* the string "hello" in the pipe then *reads* the string. What would happen if the count of data written to the pipe were 1024 instead of 6 (but the count of read data stays at 6)? What would happen if the order of the *read* and *write* system calls were reversed?
23. In the program illustrating the use of named pipes (Figure 5.19), what happens if *mknod* discovers that the named pipe already exists? How does the kernel implement this? What would happen if many reader and writer processes all attempted to

```
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 2)
        {
                printf("need 1 dir arg\n");
                exit();
        }

        if (chdir(argv[1]) == -1)
                printf("%s not a directory\n", argv[1]);
}
```

**Figure 5.38.** Sample Program with Chdir System Call

communicate through the named pipe instead of the one reader and one writer implicit in the text? How could the processes ensure that only one reader and one writer process were communicating?

24. When *opening* a named pipe for reading, a process sleeps in the *open* until another process *opens* the pipe for writing. Why? Couldn't the process return successfully from the *open*, continue processing until it tried to *read* from the pipe, and sleep in the *read*?

25. How would you implement the *dup2* (from Version 7) system call with syntax

    dup2(oldfd, newfd);

    where *oldfd* is the file descriptor to be *duped* to file descriptor number *newfd*? What should happen if *newfd* already refers to an open file?

* 26. What strange things could happen if the kernel would allow two processes to mount the same file system simultaneously at two mount points?

27. Suppose a process changes its current directory to "/mnt/a/b/c" and a second process then *mounts* a file system onto "/mnt". Should the *mount* succeed? What happens if the first process executes *pwd*? The kernel does not allow the *mount* to succeed if the inode reference count of "/mnt" is greater than 1. Comment.

28. In the algorithm for crossing a mount point on recognition of ".." in the file path name, the kernel checks three conditions to see if it is at a mount point: that the found inode has the root inode number, that the working inode is root of the file system, and that the path name component is "..". Why must it check all three conditions? Show that checking any two conditions is insufficient to allow the process to cross the mount point.

29. If a user *mounts* a file system "read-only," the kernel sets a flag in the super block. How should it prevent write operations during the *write*, *creat*, *link*, *unlink*, *chown*, and *chmod* system calls? What write operations do all the above system calls do to the file system?

* 30. Suppose a process attempts to *umount* a file system and another process is simultaneously attempting to *creat* a new file on that file system. Only one system call can succeed. Explore the race condition.

* 31. When the *umount* system call checks that no more files are active on a file system, it has a problem with the file system root inode, allocated via *iget* during the *mount* system call and hence having reference count greater than 0. How can *umount* be sure there are no active files and take account for the file system root? Consider two cases:
    - *umount* releases the root inode with the *iput* algorithm before checking for active inodes. (How does it recover if there were active files after all?)
    - *umount* checks for active files before releasing the root inode but permits the root inode to remain active. (How active can the root inode get?)

32. When executing the command *ls −ld* on a directory, note that the number of links to the directory is never 1. Why?

33. How does the command *mkdir* (make a new directory) work? (Hint: When *mkdir* completes, what are the inode numbers for "." and ".."?)

* 34. Symbolic links refer to the capability to *link* files that exist on different file systems. A new type indicator specifies a symbolic link file; the data of the file is the path name of the file to which it is linked. Describe an implementation of symbolic links.

* 35. What happens when a process executes

```
unlink(".");
```

What is the current directory of the process? Assume superuser permissions.

36. Design a system call that truncates an existing file to arbitrary sizes, supplied as an argument, and describe an implementation. Implement a system call that allows a user to remove a file segment between specified byte offsets, compressing the file size. Without such system calls, encode a program that provides this functionality.

37. Describe all conditions where the reference count of an inode can be greater than 1.

38. In file system abstractions, should each file system type support a private lock operation to be called from the generic code, or does a generic lock operation suffice?