```
#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 *exit*s, the kernel examines its active user file descriptors and internally *close*s 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 *close*s 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

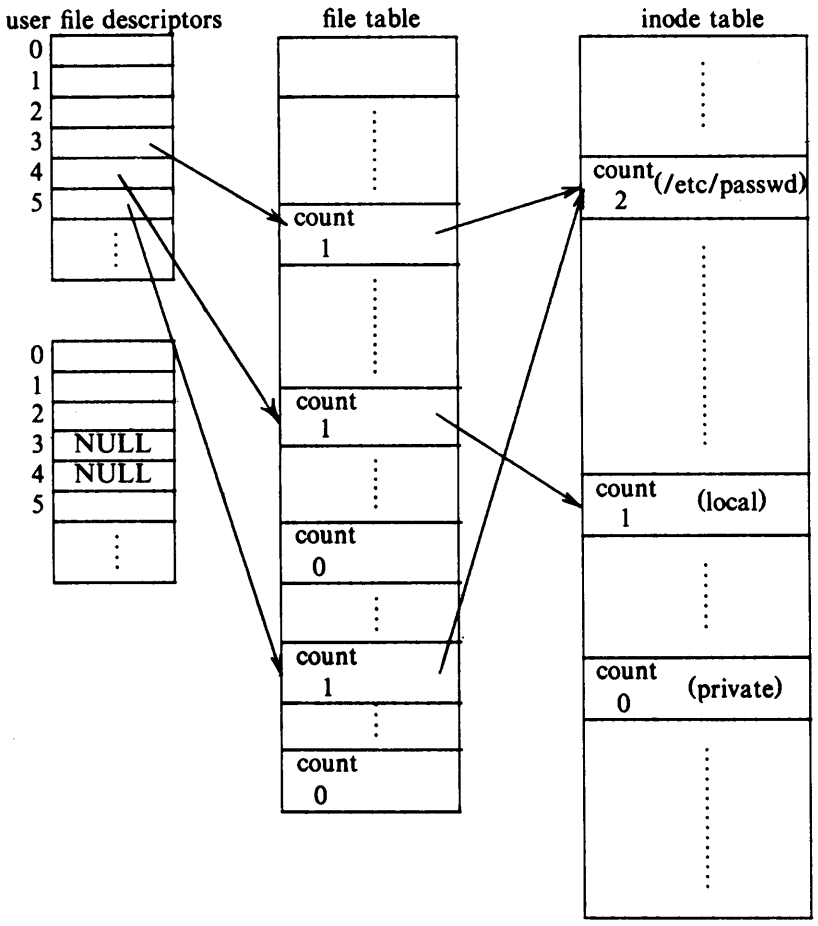

Figure 5.11. Tables after Closing a File

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 *exit*s.

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*.

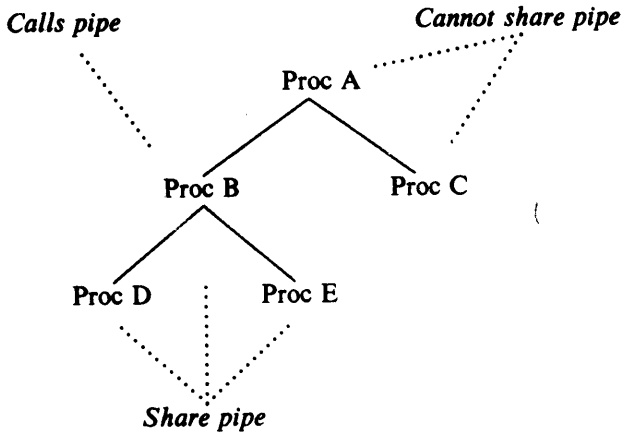

Figure 5.15. Process Tree and Sharing Pipes

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 *open*s 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 *open*s 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 *open*s a named pipe for reading and a writing process exists, the *open* call completes. Or if a process *open*s 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 *open*s the pipe. Similar rules hold for a process that *open*s 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

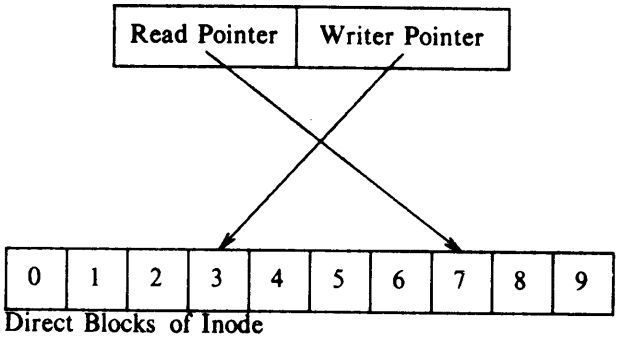

Figure 5.17. Logical View of Reading and Writing a Pipe

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 *writes*

```
#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,

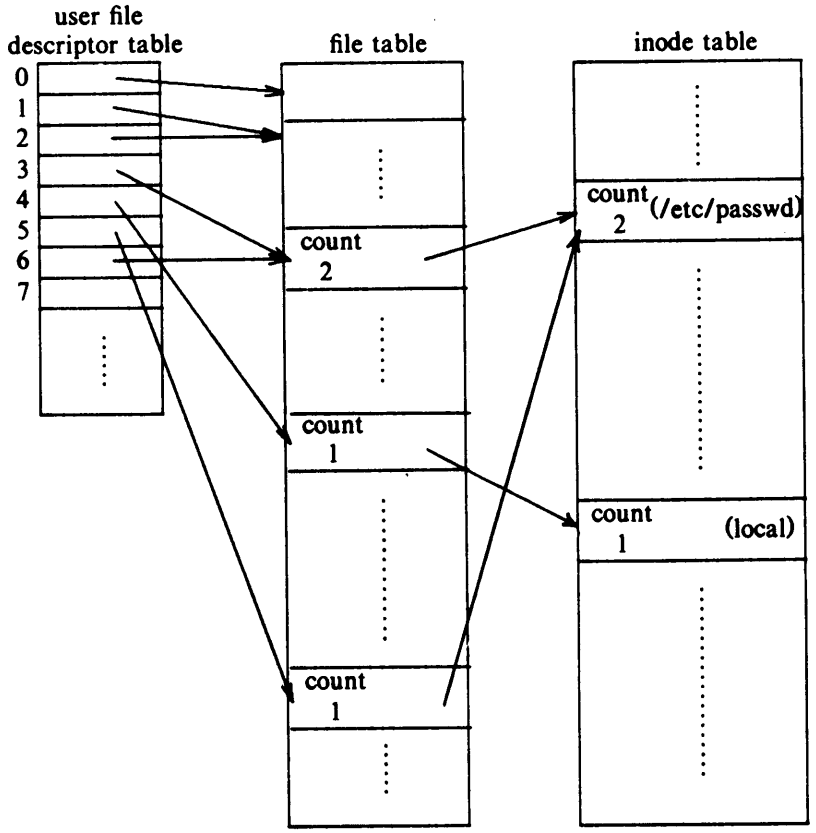

Figure 5.20. Data Structures after Dup

*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 *open*s the same file twice and *read*s 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"

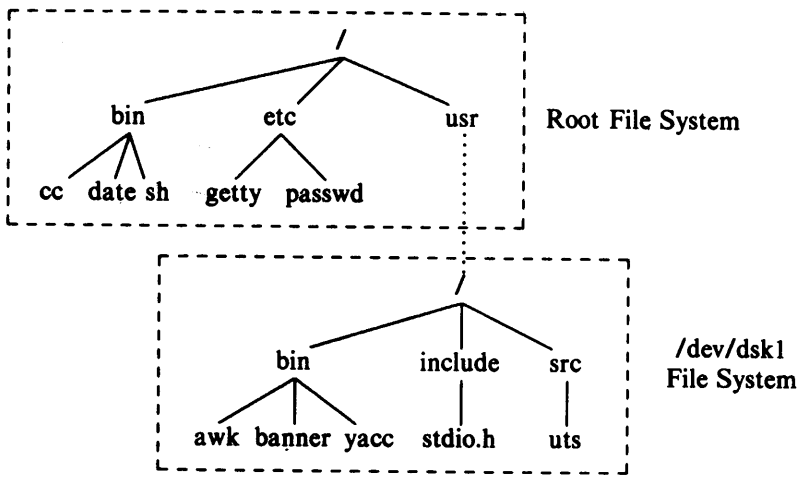

Figure 5.22. File System Tree Before and After Mount

(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.

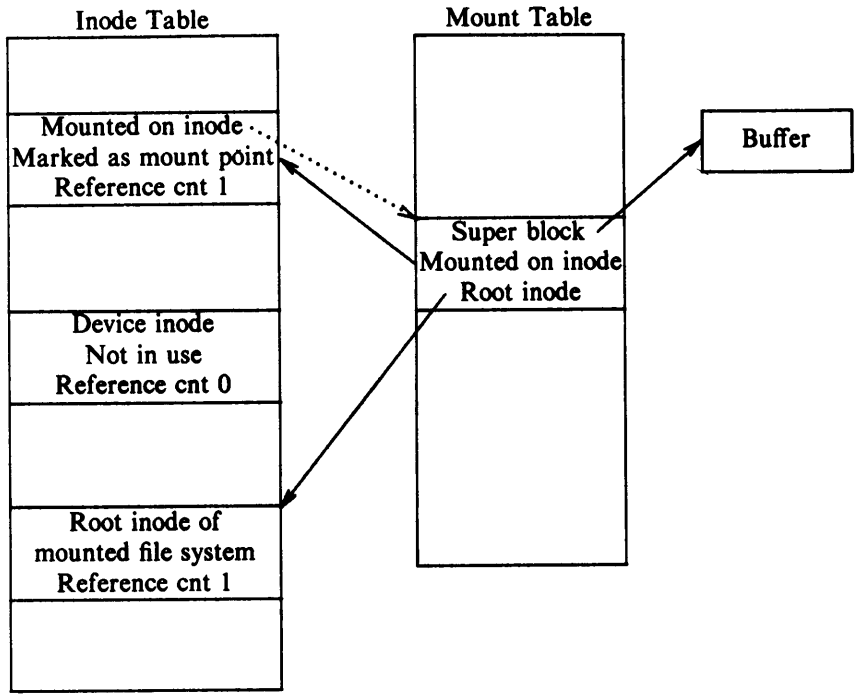

Figure 5.24. Data Structures after Mount

```
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.

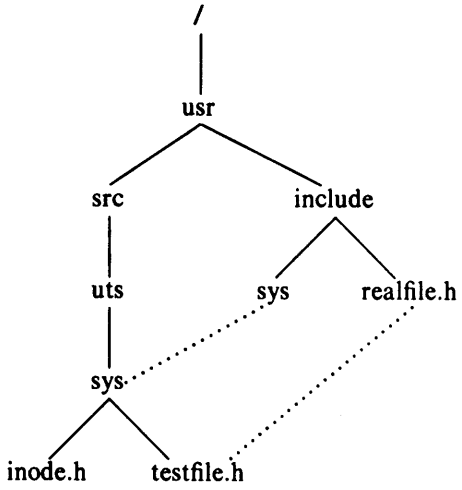

Figure 5.28. Linked Files in File System Tree

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 *link*s. 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.

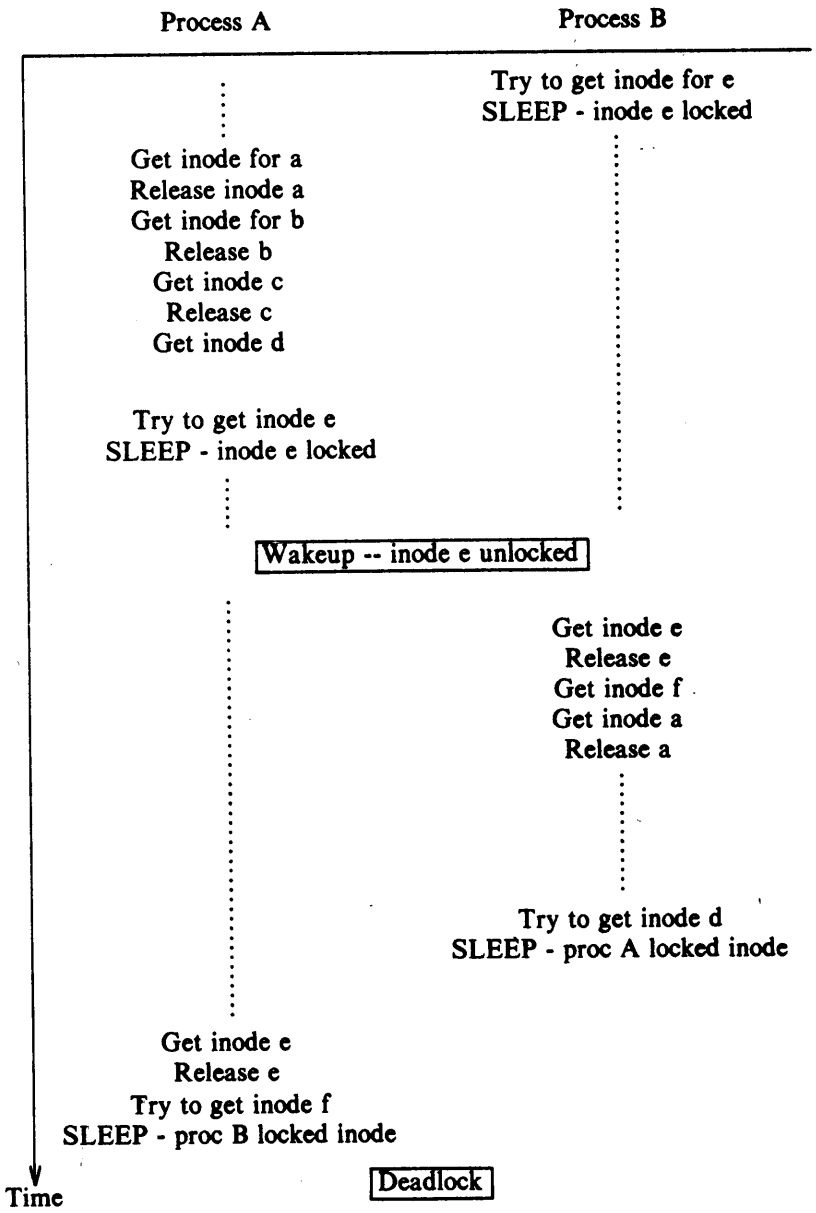

Figure 5.30. Deadlock Scenario for Link

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 link 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

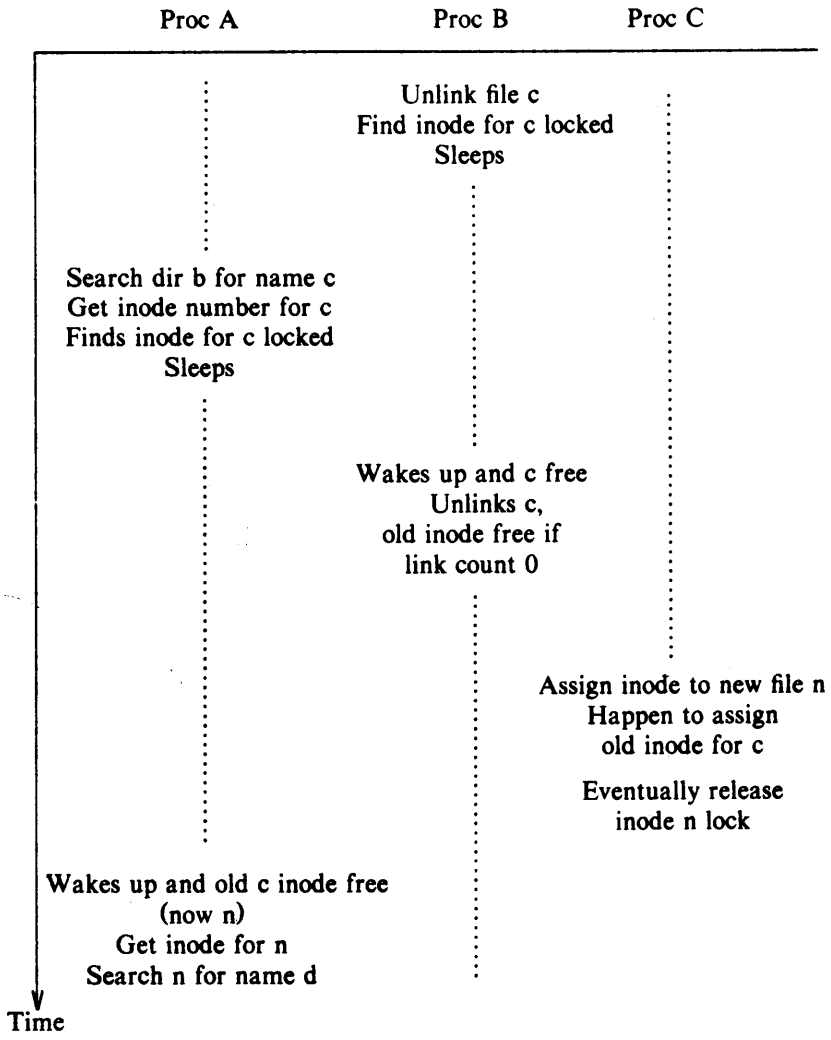

Figure 5.32. Unlink Race Condition

```
#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.

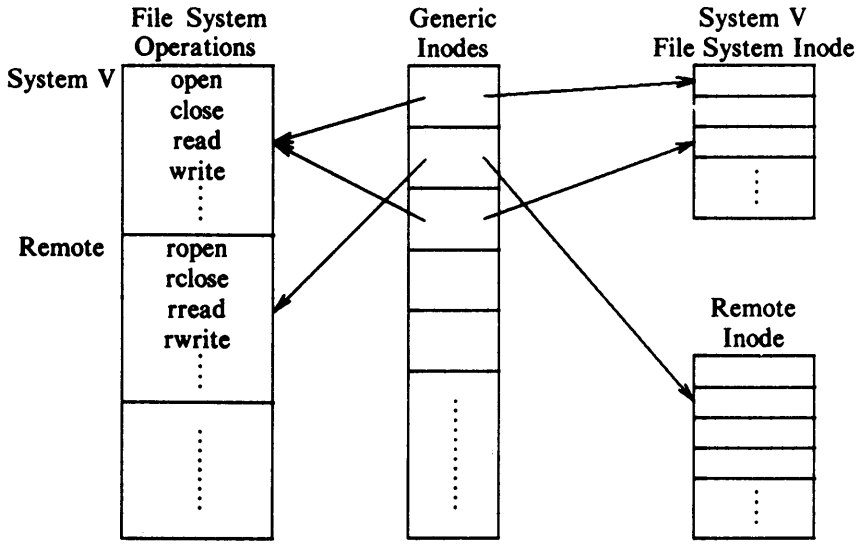

Figure 5.34. Inodes for File System Types

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 *unlink*ed before making sure that all files contained in the directory were *unlink*ed.

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 *close*s 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 *write*s the string "hello" in the pipe then *read*s 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 *open*ing a named pipe for reading, a process sleeps in the *open* until another process *open*s 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 *dup*ed 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 *mount*s 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 *mount*s 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?

6

THE STRUCTURE OF PROCESSES

Chapter 2 formulated the high-level characteristics of processes. This chapter presents the ideas more formally, defining the context of a process and showing how the kernel identifies and locates a process. Section 6.1 defines the process state model for the UNIX system and the set of state transitions. The kernel contains a process table with an entry that describes the state of every active process in the system. The *u area* contains additional information that controls the operation of a process. The process table entry and the *u area* are part of the context of a process. The aspect of the process context that most visibly distinguishes it from the context of another process is, of course, the contents of its address space. Section 6.2 describes the principles of memory management for processes and for the kernel and how the operating system and the hardware cooperate to do virtual memory address translation. Section 6.3 examines the components of the context of a process, and the rest of the chapter describes the low-level algorithms that manipulate the process context. Section 6.4 shows how the kernel saves the context of a process during an interrupt, system call, or context switch and how it later resumes execution of the suspended process. Section 6.5 gives various algorithms, used by the system calls described in the next chapter, that manipulate the process address space. Finally, Section 6.6 covers the algorithms for putting a process to sleep and for waking it up.

6.1 PROCESS STATES AND TRANSITIONS

As outlined in Chapter 2, the lifetime of a process can be conceptually divided into a set of states that describe the process. The following list contains the complete set of process states.

1. The process is executing in user mode.
2. The process is executing in kernel mode.
3. The process is not executing but is ready to run as soon as the kernel schedules it.
4. The process is sleeping and resides in main memory.
5. The process is ready to run, but the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute. Chapter 9 will reconsider this state in a paging system.
6. The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.
7. The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process. The distinction between this state and state 3 ("ready to run") will be brought out shortly.
8. The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. This state is the start state for all processes except process 0.
9. The process executed the *exit* system call and is in the *zombie* state. The process no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state is the final state of a process.

Figure 6.1 gives the complete process state transition diagram. Consider a typical process as it moves through the state transition model. The events depicted are artificial in that processes do not always experience them, but they illustrate various state transitions. The process enters the state model in the "created" state when the parent process executes the *fork* system call and eventually moves into a state where it is ready to run (3 or 5). For simplicity, assume the process enters the state "ready to run in memory." The process scheduler will eventually pick the process to execute, and the process enters the state "kernel running," where it completes its part of the *fork* system call.

When the process completes the system call, it may move to the state "user running," where it executes in user mode. After a period of time, the system clock may interrupt the processor, and the process enters state "kernel running" again. When the clock interrupt handler finishes servicing the clock interrupt, the kernel may decide to schedule another process to execute, so the first process enters state "preempted" and the other process executes. The state "preempted" is really the same as the state "ready to run in memory" (the dotted line in the figure that connects the two states emphasizes their equivalence), but they are depicted separately to stress that a process executing in kernel mode can be preempted only

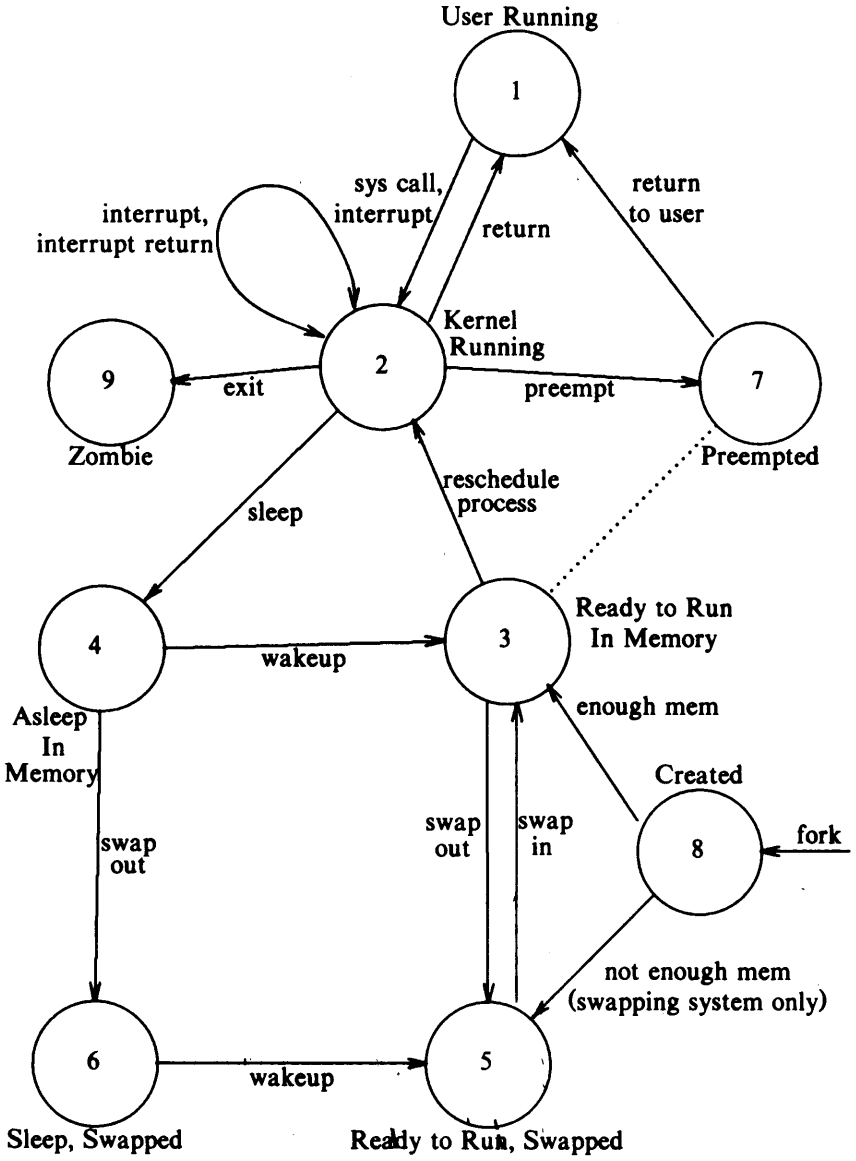

Figure 6.1. Process State Transition Diagram

when it is about to return to user mode. Consequently, the kernel could swap a process from the state "preempted" if necessary. Eventually, the scheduler will choose the process to execute, and it returns to the state "user running," executing in user mode again.

When a process executes a system call, it leaves the state "user running" and enters the state "kernel running." Suppose the system call requires I/O from the disk, and the process must wait for the I/O to complete. It enters the state "asleep in memory," putting itself to sleep until it is notified that the I/O has completed. When the I/O later completes, the hardware interrupts the CPU, and the interrupt handler awakens the process, causing it to enter the state "ready to run in memory."

Suppose the system is executing many processes that do not fit simultaneously into main memory, and the swapper (process 0) swaps out the process to make room for another process that is in the state "ready to run swapped." When evicted from main memory, the process enters the state "ready to run swapped." Eventually, the swapper chooses the process as the most suitable to swap into main memory, and the process reenters the state "ready to run in memory." The scheduler will eventually choose to run the process, and it enters the state "kernel running" and proceeds. When a process completes, it invokes the *exit* system call, thus entering the states "kernel running" and, finally, the "zombie" state.

The process has control over some state transitions at user-level. First, a process can create another process. However, the state transitions the process takes from the "created" state (that is, to the states "ready to run in memory" or "ready to run swapped") depend on the kernel: The process has no control over those state transitions. Second, a process can make system calls to move from state "user running" to state "kernel running" and enter the kernel of its own volition. However, the process has no control over when it will return from the kernel; events may dictate that it never returns but enters the zombie state (see Section 7.2 on signals). Finally, a process can *exit* of its own volition, but as indicated before, external events may dictate that it *exit*s without explicitly invoking the *exit* system call. All other state transitions follow a rigid model encoded in the kernel, reacting to events in a predictable way according to rules formulated in this and later chapters. Some rules have already been cited: No process can preempt another process executing in the kernel, for example.

Two kernel data structures describe the state of a process: the process table entry and the *u area*. The process table contains fields that must always be accessible to the kernel, but the *u area* contains fields that need to be accessible only to the running process. Therefore, the kernel allocates space for the *u area* only when creating a process: It does not need *u area*s for process table entries that do not have processes.

The fields in the process table are the following.

- The *state* field identifies the process state.
- The process table entry contains fields that allow the kernel to locate the process and its *u area* in main memory or in secondary storage. The kernel uses the

information to do a *context switch* to the process when the process moves from state "ready to run in memory" to the state "kernel running" or from the state "preempted" to the state "user running." In addition, it uses this information when swapping (or paging) processes to and from main memory (between the two "in memory" states and the two "swapped" states). The process table entry also contains a field that gives the process size, so that the kernel knows how much space to allocate for the process.

- Several user identifiers (user IDs or UIDs) determine various process privileges. For example, the user ID fields delineate the sets of processes that can send signals to each other, as will be explained in the next chapter.
- Process identifiers (process IDs or PIDs) specify the relationship of processes to each other. These ID fields are set up when the process enters the state "created" in the *fork* system call.
- The process table entry contains an event descriptor when the process is in the "sleep" state. This chapter will examine its use in the algorithms for *sleep* and *wakeup*.
- Scheduling parameters allow the kernel to determine the order in which processes move to the states "kernel running" and "user running."
- A signal field enumerates the signals sent to a process but not yet handled (Section 7.2).
- Various timers give process execution time and kernel resource utilization, used for process accounting and for the calculation of process scheduling priority. One field is a user-set timer used to send an alarm signal to a process (Section 8.3).

The *u area* contains the following fields that further characterize the process states. Previous chapters have described the last seven fields, which are briefly described again for completeness.

- A pointer to the process table identifies the entry that corresponds to the *u area*.
- The real and effective user IDs determine various privileges allowed the process, such as file access rights (see Section 7.6).
- Timer fields record the time the process (and its descendants) spent executing in user mode and in kernel mode.
- An array indicates how the process wishes to react to signals.
- The control terminal field identifies the "login terminal" associated with the process, if one exists.
- An error field records errors encountered during a system call.
- A return value field contains the result of system calls.
- I/O parameters describe the amount of data to transfer, the address of the source (or target) data array in user space, file offsets for I/O, and so on.
- The current directory and current root describe the file system environment of the process.
- The user file descriptor table records the files the process has *open*.