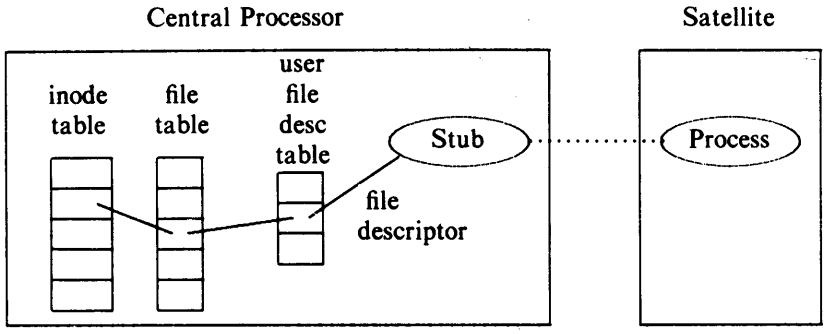

Figure 13.4. Open Call from a Satellite Process

the *open* file; the file descriptor returned by the *open* is the index into the user file descriptor table of the stub process. Figure 13.4 depicts the results of an *open* system call.

For the *write* system call, the satellite processor formulates a message, containing a *write* token, file descriptor and data count. Afterwards, it copies the data from the satellite process user space and *write*s it to the communications link. The stub process decodes the *write* message, *read*s the data from the communications link, and *write*s it to the appropriate file, following the file descriptor to the file table entry and inode, all on the central processor. When done, the stub *write*s an acknowledgment message to the satellite process, including the number of bytes successfully written. The *read* call is similar: The stub informs the satellite process if it does not return the requested number of bytes, such as when *read*ing a terminal or a pipe. Both *read* and *write* may require the transmission of multiple data messages across the network, depending on the amount of data and network packet sizes.

The only system call that needs internal modification on the central processor is the *fork* system call. When a process on the central processor executes the *fork* system call, the kernel selects a satellite to execute the process and sends a message to a special server process on the satellite, informing it that it is about to download a process. Assuming the server accepts the *fork* request, it does a *fork* to create a new satellite process, initializing a process table entry and a *u area*. The central processor downloads a copy of the *fork*ing process to the satellite processor, overwriting the address space of the process just created there, *fork*s a local stub process to communicate with the new satellite process, and sends a message to the satellite processor to initialize the program counter of the new process. The stub process (on the central processor) is the child of the *fork*ing process; the satellite process is technically a child of the server process, but it is logically a child of the process that *fork*ed. The server has no logical relationship with the child process after the *fork* completes; the only purpose of the server process is to assist in

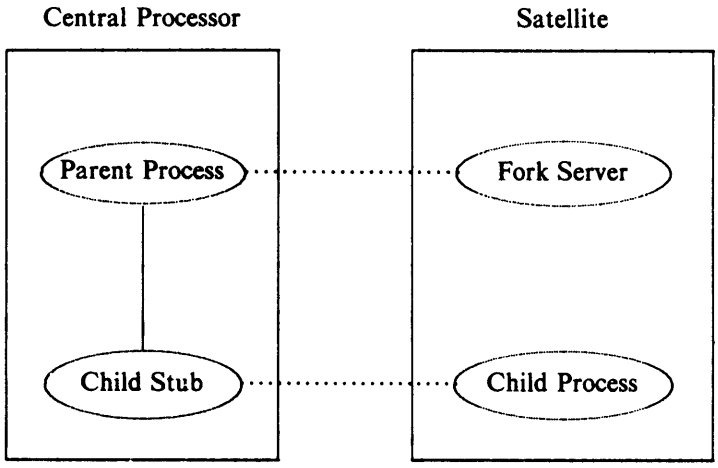

Figure 13.5. Fork on the Central Processor

downloading the child. Because of the tight coupling of the system (the satellite processors have no autonomy), the satellite and stub processes have the same process ID. Figure 13.5 illustrates the relationship between the processes: the solid line shows parent-child relationships and dotted lines depict peer-to-peer communication lines, either parent process to satellite server or child process to its stub.

When a process on a satellite processor *fork*s, it sends a message to its stub on the central processor, which then goes through a similar sequence of operations. The stub finds a new satellite processor and arranges to download the old process image: It sends a message to the parent satellite process requesting to read the process image, and the satellite responds by *writ*ing its process image to the communications link. The stub *read*s the process image and *write*s it to the child satellite. When the satellite is completely downloaded, the stub *fork*s, creating a child stub on the central processor, and *write*s the program counter to the child satellite so that it knows where to start execution. Obvious optimizations can occur if the child process is assigned to the same satellite as its parent, but this design allows processes to run on other satellite processors besides the one on which they were *fork*ed. Figure 13.6 depicts the process relationships after the *fork*. When a satellite process *exit*s, it sends an *exit* message to the stub, and the stub *exit*s. The stub cannot initiate an *exit* sequence.

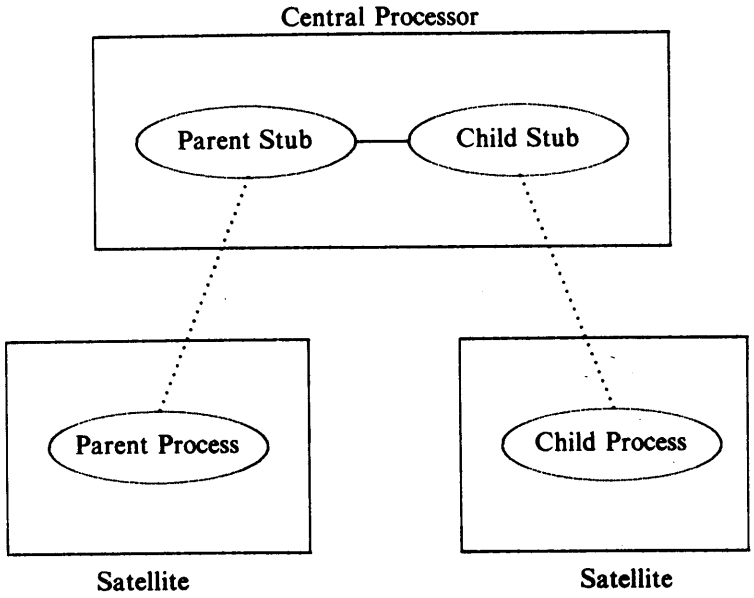

Figure 13.6. Fork on a Satellite Processor

A process must react to signals in the same way that it would react on a uniprocessor: Either it finishes the system call before it checks for the signal or it awakens immediately from its sleep and abruptly terminates the system call, depending on the priority at which it sleeps. Because a stub process handles system calls for a satellite, it must react to signals in concert with the satellite process. If a signal causes a process on a uniprocessor to finish a system call abnormally, the stub process should behave the same way. Similarly, if a signal causes a process to *exit*, the satellite *exit*s and sends an *exit* message to the stub process, which *exit*s naturally.

When a satellite process executes the *signal* system call, it stores the usual information in local tables and sends a message to the stub process, informing it whether it should ignore the particular signal or not. As will be seen, it makes no difference to the stub whether a process catches a signal or does the default operation. A process reacts to signals based on the combination of three factors (see Figure 13.7): whether the signal occurs when the process is in the middle of a system call, whether the process had called the *signal* system call to ignore the signal, or whether the signal originates on the satellite processor or on another processor. Let us consider the various possibilities.

Suppose a satellite process is asleep as the stub process executes a system call on its behalf. If a signal originates on another processor, the stub sees the signal

```
algorithm sighandle                    /* algorithm for handling signals */
input:   none
output: none
{
        if (clone process)
        {
                if (ignoring signal)
                        return;
                if (in middle of system call)
                        set signal against clone process;
                else
                        send signal message to satellite process;
        }
        else        /* satellite process */
        {
                /* whether in middle of system call or not */
                send signal to clone process;
        }
}

algorithm satellite_end_of_syscall        /* satellite end of system call */
input:        none
output:        none
{
        if (system call interrupted)
                send message to satellite telling about interrupt, signal;
        else        /* system call not interrupted */
                send system call reply:  include flag indicating arrival
                                        of signal;
}
```

Figure 13.7. Handling Signals on Satellite System

before the satellite process. There are three cases.

1. If the stub does not sleep on an event where it would wake up on occurrence of a signal, it completes the system call, sends the appropriate results in a message to the satellite process, and indicates which signal it had received.
2. If the process was ignoring the signal, the stub continues the system call algorithm without doing a *longjmp* out of an interruptible sleep — the usual behavior for ignored signals. When the stub replies to the satellite process, it does not indicate that it had received a signal.
3. If the stub process had done a *longjmp* out of the system call because of receipt of a signal, it informs the satellite process that the system call was interrupted and indicates the signal number.

The satellite process checks the response to see if signals have occurred and, if they have, handles them in the usual fashion before returning from the system call. Thus, a process behaves exactly as it would on a uniprocessor: It *exit*s without returning from the kernel, or it calls a user signal handling function, or it ignores the signal and returns from the system call.

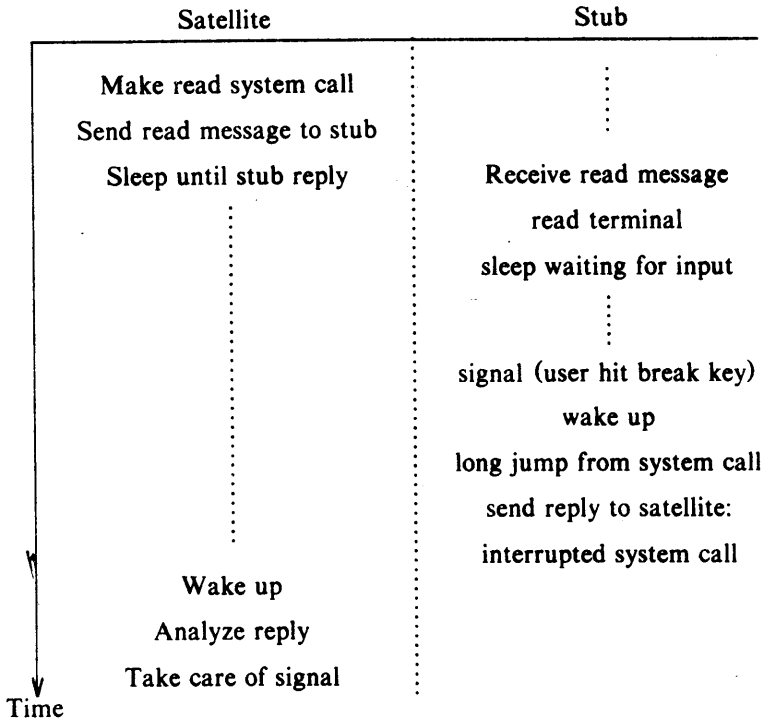

Figure 13.8. Interrupt in Middle of a System Call

For example, suppose a satellite process *read*s a terminal, which is connected to the central processor, and sleeps while the stub process executes the system call (Figure 13.8). If a user hits the break key, the stub kernel sends an interrupt signal to the stub process. If the stub was sleeping, waiting for input, it immediately wakes up and terminates the *read* call. In its response to the satellite process, the stub sets an error code (interrupted from the system call) and the signal number for interrupt. The satellite process examines the response and, because the message shows that an interrupt signal was sent, posts the signal to itself. Before returning from the *read* call, the satellite kernel checks for signals, finds the interrupt signal returned by the stub process, and handles it in the usual way. If the satellite process *exit*s as a result of the interrupt signal, the *exit* system

call takes care of killing the stub process. If it is catching interrupt signals, it calls the user signal catcher function and later returns from the *read* call, giving the user an error return. On the other hand, if the stub process was executing a *stat* system call on behalf of the satellite process, it does not terminate the system call on receipt of a signal (*stat* is guaranteed to wake up from all sleeps because it never has to wait indefinitely for a resource). The stub completes the system call and returns the signal number to the satellite process. The satellite process posts the signal to itself and discovers the signal when it returns from the system call.

If the process had been in the middle of a system call and a signal originates on the satellite processor, the satellite process has no idea whether the stub will return soon or sleep indefinitely. The satellite process sends a special message to the stub, informing it of the occurrence of the signal. The kernel on the central processor reads the message and sends the signal to the stub, which now reacts as described in the previous paragraphs: Either it interrupts the system call or it completes it. The satellite process cannot send the message to the stub directly, because the stub is in the middle of a system call and is not *read*ing the communications line. The central processor kernel recognizes the special message and posts the signal to the appropriate stub.

Repeating the *read* example explained above, the satellite process has no idea whether the stub process is waiting for input from a terminal or whether it is doing other processing. It sends the stub process a signal message: If the stub was asleep at an interruptible priority, it wakes up immediately and terminates the system call; otherwise, it completes the system call normally.

Finally, consider the cases where a signal arrives when a process is not in the middle of a system call. If the signal originates on another processor, the stub receives the signal first and sends a special signal message to the satellite process, regardless of how the satellite process wishes to dispose of the signal. The satellite kernel deciphers the message and sends the signal to the process, which reacts to it in the usual manner. If the signal had originated on the satellite processor, the satellite process does the usual processing and does not require special communication to the stub process.

When a satellite process sends a signal to other processes, it encodes a message for the *kill* system call and sends it to the stub, which executes the *kill* system call locally. If some processes that should receive the signal are on other satellite processors, their stubs receive the signal and react as described above.

13.2 THE NEWCASTLE CONNECTION

The previous section explored a tightly coupled system configuration where all file subsystem calls on a satellite processor are trapped and forwarded to a remote (central) processor. This view extends to more loosely coupled systems, where each machine wants to access files on the other machines. In a network of personal computers and work stations, for example, users may want to access files stored on a mainframe. The next two sections consider system configurations where local

systems execute all system calls but where calls to the file subsystem may access files on other machines.

These systems use one of two ways to identify remote files. Some systems insert a special character into the path name: The component name preceding the special character identifies a machine, and the remainder of the path name identifies a file on that machine. For example, the path name

"sftig!/fs1/mjb/rje"

identifies the file "/fs1/mjb/rje" on the machine "sftig". This file naming scheme follows the convention established by the *uucp* program for transferring files between UNIX systems. Other naming schemes identify remote files by prepending a special prefix such as

/../sftig/fs1/mjb/rje

where the "/.." informs the parser that the file reference is remote, and the second component name gives the remote machine name. The latter naming scheme uses the syntax of conventional file names on the UNIX system, so user software need not be converted to cope with "irregularly constructed names" as in the former scheme (see [Pike 85]).

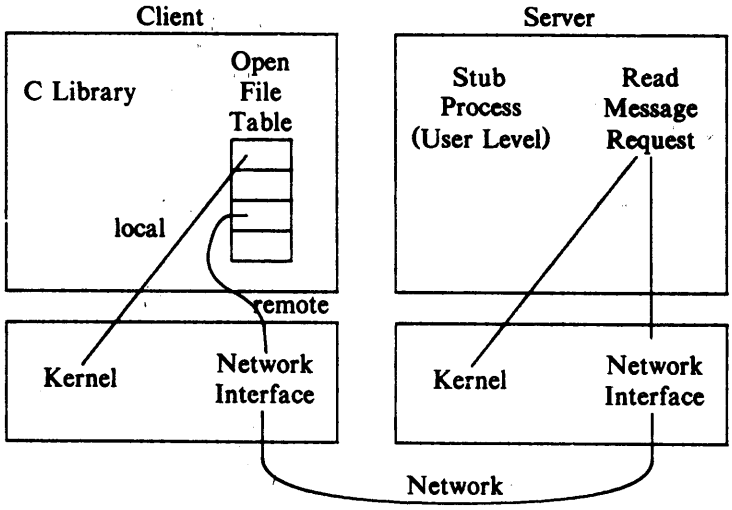

Figure 13.9. Formulation of File Service Requests

The remainder of this section describes a system modeled after the Newcastle connection, where the kernel does not participate in determining that a file is remote; instead, the C library functions that provide the kernel interface detect that

a file access is remote and take the appropriate action. For both naming conventions, the C library parses the first components of a path name to determine that a file is remote. This departs from usual implementations where the library does not parse path names. Figure 13.9 depicts how requests for file service are formulated. If a file name is local, the local kernel handles the request in the usual way. But consider execution of the system call

```
open("/../sftig/fs1/mjb/rje/file", O_RDONLY);
```

The C library routine for *open* parses the first two components of the path name and recognizes that the file should be on the remote machine "sftig". It maintains a data structure to keep track of whether the process had previously established communication to machine "sftig" and, if not, establishes a communications link to a file server process on the remote machine. When a process makes its first remote request, the remote server validates the request, mapping user and group ID fields as necessary, and creates a stub process to act as the agent for the client process.

The stub, executing requests for the client process, should have the same access rights to files that the client user would have on the remote machine. That is, user "mjb" should access remote files according to the same permissions that govern access to local files. Unfortunately, the client user ID for "mjb" may be that of a different user on the remote machine. Either the system administrators of the various machines must assign unique identifiers to all users across the network, or they must assign a transformation of user IDs at the time of request for network service. Failing the above, the stub process should execute with "other" permissions on the remote machine.

Allowing superuser access permission on remote files is a more ticklish situation. On the one hand, a client superuser should not have superuser rights on the remote system, because a user could thereby circumvent security measures on the remote system. On the other hand, various programs would not work without remote superuser capabilities. For instance, recall from Chapter 7 that the program *mkdir*, which creates a new directory, runs as a *setuid* program with superuser permissions. The remote system would not allow a client to create a new directory, because it would not recognize remote superuser permissions. The problem of creating a remote directory provides a strong rationale for implementing a *mkdir* system call, which would automatically establish all necessary directory links. Nevertheless, execution of *setuid* programs that access remote files as superuser is still a general problem that must be dealt with. Perhaps this problem could best be solved by providing files with a separate set of access permissions for remote superuser access; unfortunately, this would require changes to the structure of the disk inode to save the new permission fields and would thus cause too much turmoil in existing systems.

When an *open* call returns successfully, the local library makes an appropriate notation in a user-level library data structure, including a network address, stub process ID, stub file descriptor, and other appropriate information. The library routines for the *read* and *write* system calls examine the file descriptor to see if the

original file reference was remote and, if it was, send a message to the stub. The client process communicates with its stub for all system calls that need service on that machine. If a process accesses two files on a remote machine, it uses one stub, but if it accesses files on two remote machines, it uses two stubs: one on each machine. Similarly, if two processes access a file on a remote machine, they use two stubs. When executing a system call via a stub, the process formulates a message including the system call number, path name, and other relevant information, similar to the type of message described for satellite processors.

Manipulation of the current directory is more complicated. When a process changes directory to a remote directory, the library sends a message to the stub, which changes its current directory, and the library remembers that the current directory is remote. For all path names not beginning with a slash character, the library sends the path name to the remote machine, where the stub process resolves the path name from the current directory. If the current directory is local, the library simply passes the path name to the local kernel. Handling a *chroot* system call to a remote directory is similar, but the local kernel does not find out that the process had done a *chroot*; strictly speaking, a process can ignore a *chroot* to a remote directory, because only the library has a record of it. Exercise 13.9 considers the case of ".." over a mount point.

When a process *fork*s, the *fork* library routine sends each stub a *fork* message. The stub processes *fork* and send their child process IDs to the client parent process. The client process then invokes the (kernel) *fork* system call, and on its return to the child process, the library routine stores the appropriate address information about the child stub process; the local child process carries on its dialogue with the remote child stub. This treatment of the *fork* system call makes it easy for the stubs to keep track of open files and current directories. When a process with remote files *exit*s, the library routine sends a message to the remote stubs, which *exit* in response. The exercises explore the *exec* system call and the *exit* system call in greater detail.

The advantage of the Newcastle design is that processes can access remote files transparently, and no changes need be made to the kernel. However, there are several disadvantages with this design. System performance may be degraded. Because of the larger C library, each process takes up more memory even though it makes no remote references; the library duplicates kernel functions and takes up more space. Larger processes take longer to start up in *exec* and may cause greater contention for memory, inducing a higher degree of paging and swapping on a system. Local requests may execute more slowly because they take longer to get into the kernel, and remote requests may also be slow because they have to do more processing at user level to send requests across a network. The extra user-level processing provides more opportunities for context switches, paging, and swapping. Finally, programs must be recompiled with the new libraries to access remote files; old programs and vender supplied object modules do not work for remote files unless recompiled. The scheme described in the next section does not have these disadvantages.

13.3 TRANSPARENT DISTRIBUTED FILE SYSTEMS

The term *transparent distribution* means that users on one machine can access files on another machine without realizing that they cross a machine boundary, similar to crossing a mount point from one file system to another on one machine. Path names that access files on the remote machine look like path names that access local files: They contain no distinguishing symbols. Figure 13.10 shows a configuration where directory "/usr/src" on machine B is *mount*ed on the directory "/usr/src" on machine A. This configuration is convenient for systems that wish to share one copy of system source code, conventionally found in "/usr/src". Users on machine A can access files on machine B with the regular file name syntax, such as "/usr/src/cmd/login.c", and the kernel decides internally whether a file is remote or local. Users on machine B access local files without being aware that users on machine A can access them, too, but they cannot access files on machine A. Of course, other scenarios are possible where all remote systems are mounted at root of the local system, giving users access to all files on all systems.

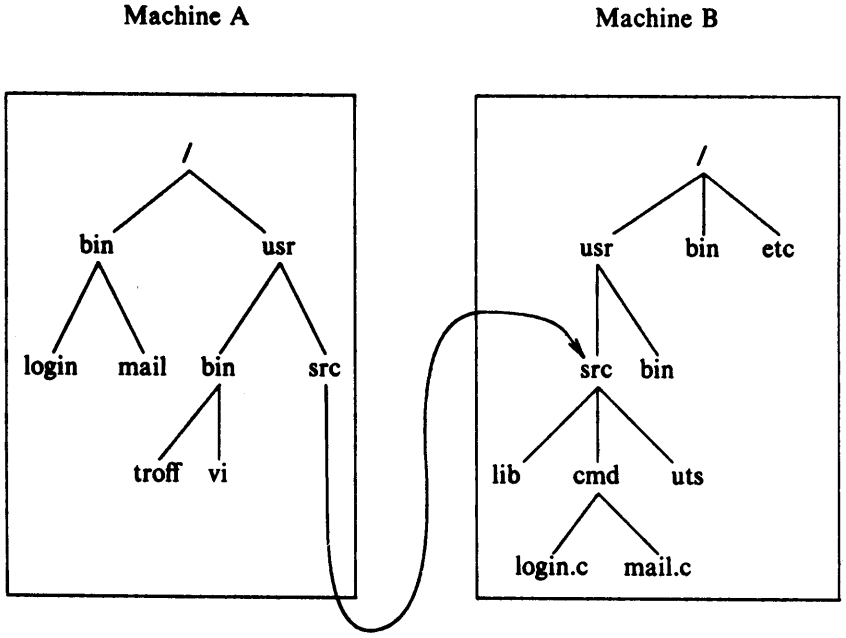

Figure 13.10. File Systems after Remote Mount

Because of the analogy between *mount*ing local file systems and providing access to remote file systems, the *mount* system call is adapted for remote file systems. The kernel contains an expanded mount table: When executing a remote

mount system call, the kernel establishes a network connection to the remote machine and stores the connection information in the mount table.

An interesting problem arises for path names that include ".." (dot-dot): If a process changes directory to a remote file system, subsequent use of ".." should return the process to the local file system rather than allow it to access files above the remotely mounted directory. Referring to Figure 13.10 again, if a process on machine A, whose current directory is in the (remote) directory "/usr/src/cmd", executes

```
cd ../../..
```

its new current directory should be root on machine A, not root on machine B. Algorithm *namei* in the remote kernel therefore checks all ".." sequences to see if the calling process is an agent for a client process, and if so, checks the current working directory to see if that client treats the directory as the root of a remotely mounted file system.

Communication with a remote machine takes on one of two forms: remote procedure call or remote system call. In a remote procedure call design, each kernel procedure that deals with inodes recognizes whether a particular inode refers to a remote file and, if it does, sends a message to the remote machine to perform a specific inode operation. This scheme fits in naturally to the abstract file system types presented at the end of Chapter 5. Thus, a system call that accesses a remote file may cause several messages across the network, depending on how many internal inode operations are involved, with correspondingly higher response time due to network latency. Carried to an extreme, the remote operations include manipulation of the inode lock, reference count, and so on. Various optimizations to the pure model have been implemented to combine several logical inode operations into a single message and to cache important data (see [Sandberg 85]).

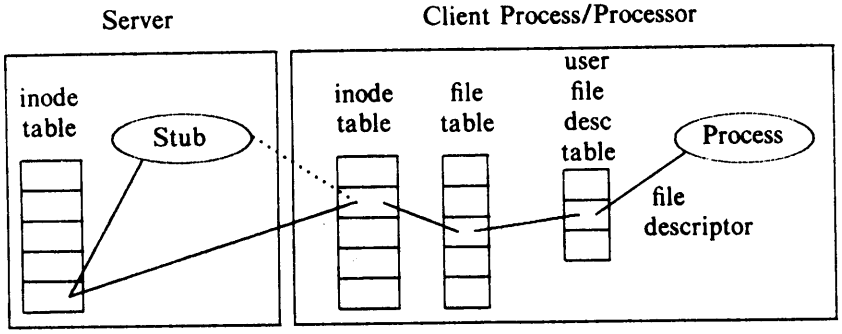

Figure 13.11. Opening a Remote File

Consider a process that *open*s the remote file "/usr/src/cmd/login.c", where "src" is the mount point. As the kernel parses the path name in *namei-iget*, it detects that the file is remote and sends a request to the remote machine to return a locked inode. On receipt of a successful response, the local kernel allocates an in-core inode that corresponds to the remote file. It then checks file modes for necessary permissions (permission to read, for instance), by sending another message to the remote machine. It continues executing the *open* algorithm as presented in Chapter 5, sending messages to the remote machine when necessary, until it completes the algorithm and unlocks the inode. Figure 13.11 illustrates the relationship of the kernel data structures at conclusion of the *open*.

For a *read* system call, the client kernel locks the local inode, sends a message to lock the remote inode, sends a message to read data, copies the data into local memory, sends a message to unlock the remote inode, and unlocks the remote inode. This scheme conforms to the semantics of existing, uniprocessor kernel code, but the frequency of network use (potentially several times per system call) hurts performance. Several operations can be combined into one message to reduce network traffic, however. In the *read* example, the client can send one "read" message to the server, which knows that it has to lock and unlock its inode while doing the read operation. Implementation of remote caches can further reduce network traffic, as mentioned above, but care must be taken to maintain the semantics of file system calls.

In a remote system call design, the local kernel recognizes that a system call refers to a remote file, as above, and sends the parameters of the system call to the remote system, which executes the *system call* and returns the results to the client. The client machine receives the results of the remote system call and *longjmp*s out of the system call. Most system calls can be executed with only one network message, resulting in reasonably good system response, but several kernel operations do not fit the model. For instance, the kernel creates a "core" file for a process on receipt of various signals (Chapter 7). Creation of a core file does not correspond to one system call but entails several inode operations, such as creation of a file, checking access permissions, and doing several write operations.

For an *open* system call, the remote system call message consists of the remainder of the path name (the path name string after the component where the remote path name was detected) and the various flags. Repeating the earlier example for a process that *open*s the file "usr/src/cmd/login.c", the kernel sends the path name "cmd/login.c" to the remote machine. The message also contains identifying information, such as user ID and group ID, needed to determine file access capabilities on the remote machine. When the remote machine responds that the *open* call succeeded, the local kernel allocates a free, local, in-core inode, marks it "remote," saves the information needed to identify the remote machine and the remote inode, and allocates a new file table entry in the usual manner. The inode on the local machine is a dummy for the real inode on the remote machine, resulting in the same configuration as the remote procedure call model (Figure 13.11). When a process issues a system call that accesses a remote file by its file

descriptor, the local kernel recognizes that the file is remote by examining its (local) inode, formulates a message encapsulating the system call, and sends the message to the remote machine. The message contains the remote inode index so that the stub can identify the remote file.

For all system calls, the local kernel may execute special code to take care of the response and may eventually *longjmp* out of the system call, because subsequent local processing, designed for a uniprocessor system, may be irrelevant. Therefore, the semantics of kernel algorithms may change to support a remote system call model. However, network traffic is kept to a minimum, allowing system response to be as fast as possible.

13.4 A TRANSPARENT DISTRIBUTED MODEL WITHOUT STUB PROCESSES

Use of stub processes in the transparent distributed system model makes it easy for the remote system to keep track of remote files, but the process table on the remote system becomes cluttered with stubs that are idle most of the time. Other schemes use special server processes on the remote machine to handle remote requests (see [Sandberg 85] and [Cole 85]). The remote system has a pool of server processes and assigns them temporarily to handle each remote request as it arrives. After handling a request, the server process reenters the pool and is available for reassignment to other requests: The server does not remember the user context (such as user ID) between system calls, because it may handle system calls for several processes. Consequently, each message from a client process must include data about its environment, such as UIDs, current directory, disposition of signals, and so on. Stub processes acquire this data at setup time or during the normal course of system call execution.

When a process *opens* a remote file, the remote kernel allocates an inode for later reference to the file. The local machine has the usual entries in the user file descriptor table, file table, and inode table, and the inode entry identifies the remote machine and inode. For system calls that use a file descriptor, like *read*, the kernel sends a message that identifies the previously allocated remote inode and passes over process-specific information, such as the user ID, the maximum allowed file size, and so on. When the remote machine dispatches a server, communication with the client process is similar to what was described previously, but the connection between the client and server exists only for the duration of the system call.

Handling flow control, signals, and remote devices is more difficult using server processes instead of stubs. If a remote machine is flooded with requests from many machines, it must queue the requests if it does not have enough server processes. This requires a higher-level protocol than the one already provided with the underlying network. In the stub model, on the other hand, a stub cannot be flooded with requests, because all transactions with a client are synchronous: A client can have at most one outstanding request.

Handling signals that interrupt a system call is also more complicated with server processes, because the remote machine must find the correct server process that is executing the system call. It is even possible that the system call request is still waiting for service if all server processes were busy. Similarly, race conditions are possible if the server returns the result of the system call to the calling process, and the response passes the signal message en route through the network. Each message must be tagged so that the remote system can locate it and interrupt server processes, if necessary. Using stub processes, the process servicing the client system call is automatically identified, and it is easy to determine if it already finished handling a system call when a signal arrives.

Finally, if a process issues a system call that causes the server process to sleep indefinitely (reading a remote terminal, for example), the server process cannot handle other requests, effectively removing it from the server process pool. If many processes access remote devices and if there is an upper bound on the number of server processes, this can be a severe bottleneck. This cannot happen when using stub processes, because the stubs are allocated per client process. Exercise 13.14 explores another problem in using server processes for remote devices.

In spite of the advantages for using process stubs, the need for process table slots is so critical in practice that most schemes use a pool of service processes to handle remote requests.

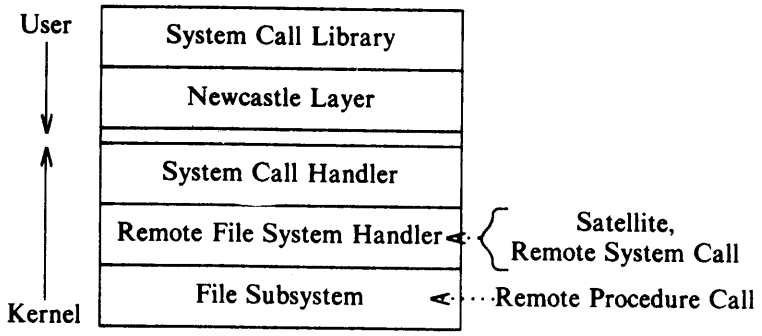

Figure 13.12. Conceptual Kernel Layer for Remote File Access

13.5 SUMMARY

This chapter has described three schemes for allowing processes to access files stored on remote machines, treating the remote file systems as an extension of the local file system. Figure 13.12 illustrates the architectural difference between them. These systems are distinguished from the multiprocessor systems described in the previous chapter, because processors do not share physical memory. The satellite

processor scheme consists of a tightly coupled set of processors that share the file resources of a central processor. The Newcastle connection gives the appearance of transparent, remote file access, but remote access is provided by a special implementation of the C library, not by the kernel. Consequently, programs must be recompiled to use the Newcastle connection, sometimes a serious drawback. Remote files are designated by special character sequences that identify the machine that stores the file, another factor that can limit portability.

A transparent distributed system uses a variation of the *mount* system call to give access to a remote file system, much as the usual *mount* system call extends the local file system to newly mounted disk units. Inodes on the local system indicate that they refer to remote files, and the local kernel sends messages to the remote kernel, describing the kernel algorithm (system call), its parameters, and the remote inode. Two designs support the remote transparent, distributed operations: a remote procedure call model, where the messages instruct the remote machine to execute inode operations, and a remote system call model, where the messages instruct the remote machine to execute system calls. Finally, the chapter examined the issues involved with serving remote requests with stub processes or with server processes from a general pool.

13.6 EXERCISES

* 1. Describe an implementation of the *exit* system call on a satellite processor system. How is this different from the case where a process *exit*s as a result of receipt of an uncaught signal? How should the kernel dump the "core" file?

2. Processes cannot ignore the *SIGKILL* signal; describe what happens on a satellite system when a process receives this signal.

* 3. Describe an implementation of the *exec* system call on a satellite processor system.

* 4. How should a central processor assign processes to satellite processors to balance the execution load?

* 5. What happens if a satellite processor does not contain enough memory for the processes downloaded to it? How should it handle swapping or paging across a network?

6. Consider a system that allows access to remote file server machines by recognizing path names by special prefaces. Suppose a process executes

```
execl("/../sftig/bin/sh", "sh", 0);
```

The executable image is on the remote machine but should execute on the local machine. Describe how the local system brings the remote executable file to the local system to do the *exec*.

7. If an administrator wishes to add new machines to a Newcastle system, what is the best way to inform the C library modules?

* 8. The kernel overwrites the address space of a process during *exec*, including the library tables used by a Newcastle-style implementation to keep track of remote file references. The process must still be able to access these files by their old file descriptors after the *exec*. Describe an implementation.

* 9. As described in Section 13.2, execution of the *exit* system call on Newcastle systems results in a message being sent to the stub process that causes it to *exit*. This is done at the library level. What happens if the local process receives a signal that causes it to *exit* from the kernel?
* 10. In a Newcastle-style system, where remote files are designated by special prefaces, how should the system allow a user to use the ".." (parent directory) component to back up over a remote mount point?
11. Recall from Chapter 7 that various signals cause a process to dump a core file in its current directory. What should happen if the current directory is in a remote file system? What happens on a Newcastle system?
* 12. If someone on a remote processor kills all stub or server processes, how should the local processes hear the good news?
* 13. In the transparent distribution system, discuss implementations of *link*, which has two possibly remote path names, and *exec*, which has several internal read operations. Consider the two designs: remote procedure call and remote system call.
* 14. When a (nonstub) server process accesses a device, it may have to sleep until the device driver wakes it up. Given a fixed number of servers, it is conceivable that a system would be unable to satisfy any more requests from a local machine, because all servers are sleeping in a device driver. Devise a scheme that is safe, in that not all servers can sleep, waiting for device I/O. A system call should not fail because all servers are currently busy.

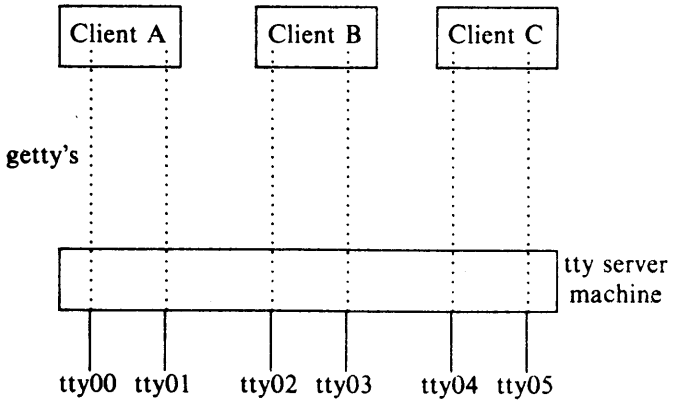

Figure 13.13. A Terminal Server Configuration

* 15. When a user logs into a system, the terminal line discipline saves information that the terminal is a control terminal, noting the process group. In this way, processes receive interrupt signals when a user hits the break key at the terminal. Consider a system configuration where all terminals are physically connected to one machine, but users log in logically on other machines (Figure 13.13). Specifically, a system spawns a *getty* process for a remote terminal. If a pool of server processes handle remote system calls, a server sleeps in the driver open procedure, waiting for a connection. When the server completes the *open* system call, it goes back into the process pool, severing its

connection to the terminal. If a user hits the break key, how is the interrupt signal sent to processes in the process group executing on the client machine?

* 16. The shared memory feature is inherently a local-machine operation. Logically, it would be possible for processes on different machines to access a common piece of physical memory, whether the memory is local or remote. Describe an implementation.

* 17. The demand paging and swapping algorithms examined in Chapter 9 assume the use of a local swap device. What modifications must be made to these algorithms to support remote swap devices?

* 18. Suppose a remote machine crashes (or the network goes down) and the local network protocol can recognize this fact. Design recovery schemes for a local system that makes requests of a remote, server system. Conversely, design recovery schemes for a server system that loses its connection with client machines.

* 19. When a process accesses a remote file, the path name may stretch across several machines until it is completely resolved. Following the path name "/usr/src/uts/3b2/os" for example, "/usr" may be on machine A, the root of machine B may be mounted on "/usr/src", and the root of machine C may be mounted on "/usr/src/uts/3b2". Moving through several machines to get to the final destination is called *multihop*. If a direct network connection exists between A and C, however, it is inefficient to transfer data between the machines via machine B. Describe a design for multi-hop in the Newcastle and transparent distribution models.

APPENDIX – SYSTEM CALLS

This appendix contains a brief synopsis of the UNIX system calls. Refer to the UNIX System V User Programmer's Manual for a complete specification of these calls. The specification here is sufficient for reference when reading the various program examples in the book.

The specified file names are null terminated character strings, whose individual components are separated by slash characters. All system calls return −1 on error, and the external variable errno indicates the specific error. Unless specified otherwise, system calls return 0 on success. Some system calls are the entry point for several functions: this means that the assembly language interface for the functions is the same. The list here follows the usual conventions for UNIX system manuals, but the programmer should not care whether a system call entry point handles one or many system calls.

access

```
access(filename, mode)
char *filename;
int mode;
```

Access checks if the calling process has read, write, or execute permission for the file, according to the value of *mode*. The value of *mode* is a combination of the bit

patterns 4 (for read), 2 (for write), and 1 (for execute). The real-user ID is checked instead of the effective user ID.

acct

```
acct(filename)
char *filename;
```

Acct enables system accounting if *filename* is non-null, and disables it otherwise.

alarm

```
unsigned alarm(seconds)
unsigned seconds;
```

Alarm schedules the occurrence of an alarm signal for the calling process in the indicated number of *seconds*. It returns the amount of time remaining until the alarm signal at the time of the call.

brk

```
int brk(end_data_seg)
char *end_data_seg;
```

Brk sets the highest address of a process's data region to *end_data_seg*. Another function, *sbrk*, uses this system call entry point and changes the highest address of a process's data region according to a specified increment.

chdir

```
chdir(filename)
char *filename;
```

Chdir changes the current directory of the calling process to *filename*.

chmod

```
chmod(filename, mode)
char *filename;
```

Chmod changes the access permissions of the indicated file to the specified *mode*, which is a combination of the following bits (in octal):

04000	setuid bit
02000	set group ID bit

01000	sticky bit
00400	read for owner
00200	write for owner
00100	execute for owner
00040	read for group
00020	write for group
00010	execute for group
00004	read for others
00002	write for others
00001	execute for others

chown

```
chown(filename, owner, group)
char *filename;
int owner, group;
```

Chown changes the owner and group of the indicated file to the specified *owner* and *group* IDs.

chroot

```
chroot(filename)
char *filename;
```

Chroot sets the private, changed-root of the calling process to *filename*.

close

```
close(fildes)
int fildes;
```

Close closes a file descriptor obtained from a prior *open*, *creat*, *dup*, *pipe*, or *fcntl* system call, or a file descriptor inherited from a *fork* call.

creat

```
creat(filename, mode)
char *filename;
int mode;
```

Creat creates a new file with the indicated file name and access permission modes. *Mode* is as specified in *access*, except that the sticky-bit is cleared and bits set via *umask* are cleared. If the file already exists, *creat* truncates the file. *Creat* returns a file descriptor for use in other system calls.

dup

```
dup(fildes)
int fildes;
```

Dup duplicates the specified file descriptor, returning the lowest available file descriptor. The old and new file descriptors use the same file pointer and share other attributes.

exec

```
execve(filename, argv, envp)
char *filename;
char *argv[];
char *envp[];
```

Execve executes the program file *filename*, overlaying the address space of the executing process. *Argv* is an array of character strings parameters to the *exec*ed program, and *envp* is an array of character strings that are the environment of the new process.

exit

```
exit(status)
int status;
```

Exit causes the calling process to terminate, reporting the 8 low-order bits of status to its waiting parent. The kernel may call exit internally, in response to certain signals.

fcntl

```
fcntl(fildes, cmd, arg)
int fildes, cmd, arg;
```

Fcntl supports a set of miscellaneous operations for open files, identified via the file descriptor *fildes*. The interpretation of *cmd* and *arg* is as follows (manifest constants are defined in file "/usr/include/fcntl.h"):

F_DUPFD	return lowest numbered file descriptor >= arg
F_SETFD	set close-on-exec flag to low order bit of arg (if 1, file is closed in exec)
F_GETFD	return value of close-on-exec flag
F_SETFL	set file status flags (O_NDELAY do not sleep for I/O and

O_APPEND append written data to end of file)
F_GETFL get file status flags

```
struct flock
        short l_type;    /* F_RDLCK for read lock, F_WRLCK for write lock,
                            F_UNLCK for unlock operations */
        short l_whence; /* lock offset is from beginning of file (0), current position of file
                            pointer (1), or end of file (2) */
        long l_start;    /* byte offset interpreted according to l_whence */
        long l_len;      /* number of bytes to lock. If 0, lock from l_start to end of file */
        long l_pid;      /* ID of process that locked file */
        long l_sysid;    /* sys ID of process that locked file */
```

F_GETLK get first lock that would prevent application of the lock specified by *arg* and overwrite *arg*. If no such lock exists, change l_type in *arg* to F_UNLCK
F_SETLK lock or unlock the file as specified by *arg*. Return -1 if unable to lock.
F_SETLKW lock or unlock data in a file as specified by *arg*. Sleep if unable to lock.

Several read locks can overlap in a file. No locks can overlap a write lock.

fork

fork()

Fork creates a new process. The child process is a logical copy of the parent process, except that the parent's return value from the *fork* is the process ID of the child, and the child's return value is 0.

getpid

getpid()

Getpid returns the process ID of the calling process. Other calls that use this entry point are *getpgrp*, which returns the process group of the calling process, and *getppid*, which returns the parent process ID of the calling process.

getuid

getuid()

Getuid returns the real user ID of the calling process. Other calls that use this system call entry point are *geteuid*, which returns the effective user ID, *getgid*, which returns the group ID, and *getegid*, which returns the effective group ID of the calling process.

ioctl

```
ioctl(fildes, cmd, arg)
int fildes, cmd;
```

Ioctl does device-specific operations on the open device whose file descriptor is *fildes*. *Cmd* specifies the command to be done on the device, and *arg* is a parameter whose type depends on the command.

kill

```
kill(pid, sig)
int pid, sig;
```

Kill sends the signal sig to the processes identified by *pid*.

pid positive	send signal to process whose PID is pid.
pid 0	send signal to processes whose process group ID is PID of sender.
pid −1	if effective UID of sender is super user, send signal to all processes otherwise, send signal to all processes whose real UID equals effective UID of sender.
pid < −1	send signal to processes whose process group ID is pid.

The effective UID of the sender must be superuser, or the sender's real or effective UID must equal the real or effective UID of the receiving processes.

link

```
link(filename1, filename2)
char *filename1, *filename2;
```

Link gives another name, *filename2*, to the file *filename1*. The file becomes accessible through either name.

lseek

```
lseek(fildes, offset, origin)
int fildes, origin;
long offset;
```

Lseek changes the position of the read-write pointer for the file descriptor *fildes* and returns the new value. The value of the pointer depends on *origin*:

0	set the pointer to offset bytes from the beginning of the file.
1	increment the current value of the pointer by offset.
2	set the pointer to the size of the file plus offset bytes.

mknod

```
mknod(filename, modes, dev)
char *filename;
int mode, dev;
```

Mknod creates a special file, directory, or FIFO according to the type of *modes*:

010000	FIFO (named pipe)
020000	character special device file
040000	directory
060000	block special device file

The 12 low order bits of *modes* have the same meaning as described above for *chmod*. If the file is block special or character special, *dev* gives the major and minor numbers of the device.

mount

```
mount(specialfile, dir, rwflag)
char *specialfile, *dir;
int rwflag;
```

Mount mounts the file system specified by *specialfile* onto the directory *dir*. If the low-order bit of *rwflag* is 1, the file system is mounted read-only.

msgctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgctl(id, cmd, buf)
int id, cmd;
struct msqid_ds *buf;
```

Msgctl allows processes to set or query the status of the message queue id, or to remove the queue, according to the value of *cmd*. The structure *msqid_ds* is defined as follows:

```
struct ipc_perm {
    ushort      uid;    /* current user id */
    ushort      gid;    /* current group id */
    ushort      cuid;   /* creator user id */
    ushort      cgid;   /* creator group id */
    ushort      mode;   /* access modes */
    short       pad1;   /* used by system */
    long        pad2;   /* used by system */
};
```

```
struct msqid_ds {
        struct ipc_perm    msg_perm;    /* permission struct */
        short              pad1[7];     /* used by system */
        ushort             msg_qnum;    /* number of messages on q */
        ushort             msg_qbytes;  /* max number of bytes on q */
        ushort             msg_lspid;   /* pid of last msgsnd operation */
        ushort             msg_lrpid;   /* pid of last msgrcv operation */
        time_t             msg_stime;   /* last msgsnd time */
        time_t             msg_rtime;   /* last msgrcv time */
        time_t             msg_ctime;   /* last change time */
};
```

The commands and their meaning are as follows:

IPC_STAT	Read the message queue header associated with id into buf.
IPC_SET	Set the values of msg_perm.uid, msg_perm.gid, msg_perm.mode (9 low-order bits), and msg_qbytes from the corresponding values in buf.
IPC_RMID	Remove the message queue for id.

msgget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgget(key, flag)
key_t key;
int flag;
```

Msgget returns an identifier to a message queue whose name is *key*. *Key* can specify that the returned queue identifier should refer to a private queue (*IPC_PRIVATE*), in which case a new message queue is created. *Flag* specifies if the queue should be created (*IPC_CREAT*), and if creation of the queue should be exclusive (*IPC_EXCL*). In the latter case, *msgget* fails if the queue already exists.

msgsnd and msgrcv

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

msgsnd(id, msgp, size, flag)
int id, size, flag;
struct msgbuf *msgp;
```

```
msgrcv(id, msgp, size, type, flag)
int id, size, type, flag;
struct msgbuf *msgmp;
```

Msgsnd sends a message of size bytes in the buffer *msgp* to the message queue *id*. *Msgbuf* is defined as

```
struct msgbuf
        long mtype;
        char mtext[];
};
```

If the *IPC_NOWAIT* bit is off in *flag*, *msgsnd* sleeps if the number of bytes on the message queue exceeds the maximum, or if the number of messages system-wide exceeds a maximum value. If *IPC_NOWAIT* is set, *msgsnd* returns immediately in these cases.

Msgrcv receives messages from the queue identified by *id*. If *type* is 0, the first message on the queue is received; if positive, the first message of that type is received; if negative, the first message of the lowest type less than or equal to type is received. *Size* indicates the maximum size of message text the user wants to receive. If *MSG_NOERROR* is set in *flag*, the kernel truncates the received message if its size is larger than *size*. Otherwise it returns an error. If *IPC_NOWAIT* is not set in *flag*, *msgrcv* sleeps until a message that satisfies *type* is sent. If *IPC_NOWAIT* is set, it returns immediately. *Msgrcv* returns the number of bytes in the message text.

nice

```
nice(increment)
int increment;
```

Nice adds *increment* to the process nice value. A higher nice value gives the process lower scheduling priorities.

open

```
#include <fcntl.h>

open(filename, flag, mode)
char *filename;
int flag, mode;
```

Open opens the specified file according to the value of *flag*. The value of *flag* is a combination of the following bits (exactly one of the first three bits must be used).

O_RDONLY	open for reading only.
O_WRONLY	open for writing only.
O_RDWR	open for reading and writing.
O_NDELAY	For special devices, open returns without waiting for carrier. if set. For named pipes, open will return immediately (with an error if O_WRONLY set), instead of waiting for another process to open the named pipe.
O_APPEND	causes all writes to append data to the end of the file.
O_CREAT	create the file if it does not exist. Mode specifies permissions as in creat system call. The flag has no meaning if the file already exists.
O_TRUNC	Truncate length of file to 0.
O_EXCL	Fail the open call if this bit and O_CREAT are set and file exists. This is a so-called exclusive open.

Open returns a file descriptor for use in other system calls.

pause

```
pause()
```

Pause suspends the execution of the calling process until it receives a signal.

pipe

```
pipe(fildes)
int fildes[2];
```

Pipe returns a read and write file descriptor (*fildes[0]* and *fildes[1]*, respectivelv). Data is transmitted through a pipe in first-in-first-out order; data cannot be read twice.

plock

```
#include <sys/lock.h>

plock(op)
int op;
```

Plock locks and unlocks process regions in memory according to the value of *op*:

PROCLOCK	lock text and data regions in memory.
TXTLOCK	lock text region in memory.
DATLOCK	lock data region in memory.
UNLOCK	remove locks for all regions.

profil

```
profil(buf, size, offset, scale)
char *buf;
int size, offset, scale;
```

Profil requests that the kernel give an execution profile of the process. *Buf* is an array in the process that accumulates frequency counts of execution in different addresses of the process. *Size* is the size of the buf array, *offset* is the starting address in the process that should be profiled, and *scale* is a scaling factor.

ptrace

```
ptrace(cmd, pid, addr, data)
int cmd, pid, addr, data;
```

Ptrace allows a process to trace the execution of another process, *pid*, according to the value of *cmd*.

0	enable child for tracing (called by child).
1,2	return word at location addr in traced process pid.
3	return word from offset addr in traced process u area.
4,5	write value of data into location addr in traced process.
6	write value of data into offset addr in u area.
7	cause traced process to resume execution.
8	cause traced process to exit.
9	machine dependent – set bit in PSW for single-stepping execution.

read

```
read(fildes, buf, size)
int fildes,
char *buf;
int size;
```

Read reads up to size bytes from the file *fildes* into the user buffer *buf*. *Read* returns the number of bytes it read. For special devices and pipes, *read* returns immediately if *O_NDELAY* was set in open and no data is available for return.

semctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
semctl(id, num, cmd, arg)
int id, num, cmd;
union semun {
        int val;
        struct semid_ds *buf;
        ushort *array;
} arg;
```

Semctl does the specified *cmd* on the semaphore queue indicated by *id*.

GETVAL	return the value of the semaphore whose index is num.
SETVAL	set the value of the semaphore whose index is num to arg.val.
GETPID	return value of last PID that did a semop on the semaphore whose index is num.
GETNCNT	return number of processes waiting for semaphore value to become positive.
GETZCNT	return number of processes waiting for semaphore value to become 0.
GETALL	return values of all semaphores into array arg.array.
SETALL	set values of all semaphores according to array arg.array.
IPC_STAT	read structure of semaphore header for id into arg.buf.
IPC_SET	set sem perm.uid, sem_per.gid, and sem_perm.mode (low-order 9 bits) according to arg.buf.
IPC_RMID	remove the semaphores associated with id.

Num gives the number of semaphores in the set to be processed. The structure *semid_ds* is defined by:

```
struct semid_ds {
    struct ipc_perm   sem_perm;    /* permission struct */
    int *             pad;         /* used by system */
    ushort            sem_nsems;   /* number of semaphores in set */
    time_t            sem_otime;   /* last semop operation time */
    time_t            sem_ctime;   /* last change time */
};
```

The structure *ipc_perm* is the same as defined in *msgctl*.

semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

semget(key, nsems, flag)
key_t key;
int nsems, flag;
```

Semget creates an array of semaphores, corresponding to *key*. *Key* and *flag* take on the same meaning as they do in *msgget*.

semop

```
semop(id, ops, num)
int id, num;
struct sembuf **ops;
```

Semop does the set of semaphore operations in the array of structures *ops*, to the set of semaphores identified by *id*. *Num* is the number of entries in *ops*. The structure of *sembuf* is:

```
struct sembuf {
        short       sem_num;    /* semaphore number */
        short       sem_op;     /* semaphore operation */
        short       sem_flg;    /* flag */
};
```

Sem_num specifies the index in the semaphore array for the particular operation, and *sem_flg* specifies flags for the operation. The operations *sem_op* for semaphores are:

negative	if sum of semaphore value and sem_op >= 0, add sem_op to to semaphore value. Otherwise, sleep, as per flag.
positive	add sem_op to semaphore value.
zero	continue, if semaphore value is 0. Otherwise, sleep as per flag.

If *IPC_NOWAIT* is set in *sem_flg* for a particular operation, *semop* returns immediately for those occasions it would have slept. If the *SEM_UNDO* flag is set, the operation is subtracted from a running sum of such values. When the process exits, this sum is added to the value of the semaphore. *Semop* returns the value of the last semaphore operation in *ops* at the time of the call.

setpgrp

```
setpgrp()
```

Setpgrp sets the process group ID of the calling process to its process ID and returns the new value.

setuid

```
setuid(uid)
int uid;
```

```
setgid(gid)
int gid;
```

Setuid sets the real and effective user ID of the calling process. If the effective user ID of the caller is superuser, *setuid* resets the real and effective user IDs. Otherwise, if its real user ID equals *uid*, *setuid* resets the effective user ID to *uid*. Finally, if its saved user ID (set by executing a setuid program in *exec*) equals *uid*, *setuid* resets the effective user ID to *uid*. *Setgid* works the same way for real and effective group IDs.

shmctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

shmctl(id, cmd, buf)
int id, cmd;
struct shmid_ds *buf;
```

Shmctl does various control operations on the shared memory region identified by *id*. The structure *shmid_ds* is defined by:

```
struct shmid_ds {
        struct ipc_perm   shm_perm;    /* permission struct */
        int               shm_segsz;   /* size of segment */
        int *             pad1;        /* used by system */
        ushort            shm_lpid;    /* pid of last operation */
        ushort            shm_cpid;    /* pid of creator */
        ushort            shm_nattch;  /* number currently attached */
        short             pad2;        /* used by system */
        time_t            shm_atime;   /* last attach time */
        time_t            shm_dtime;   /* last detach time */
        time_t            shm_ctime;   /* last change time */
};
```

The operations are:

IPC_STAT	read values of shared memory header for id into buf.
IPC_SET	set shm_perm.uid, shm_perm.gid, and shm_perm.mode (9 low-order bits) in shared memory header according to values in buf.
IPC_RMID	remove shared memory region for id.

shmget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
shmget(key, size, flag)
key_t key;
int size, flag;
```

Shmget accesses or creates a shared memory region of *size* bytes. The parameters *key* and *flag* have the same meaning as they do for *msgget*.

shmop

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

shmat(id, addr, flag)
int id, flag;
char *addr;

shmdt(addr)
char *addr;
```

Shmat attaches the shared memory region identified by *id* to the address space of a process. If *addr* is 0, the kernel chooses an appropriate address to attach the region. Otherwise, it attempts to attach the region at the specified address. If the *SHM_RND* bit is on in *flag*, the kernel rounds off the address, if necessary. *Shmat* returns the address where the region is attached.

Shmdt detaches the shared memory region previously attached at *addr*.

signal

```
#include <signal.h>

signal(sig, function)
int sig;
void (*func)();
```

Signal allows the calling process to control signal processing. The values of *sig* are:

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT	quit
SIGILL	illegal instruction
SIGTRAP	trace trap
SIGIOT	IOT instruction
SIGEMT	EMT instruction
SIGFPE	floating point exception
SIGKILL	kill

SIGBUS	bus error
SIGSEGV	segmentation violation
SIGSYS	bad argument in system call
SIGPIPE	write on a pipe with no reader
SIGALRM	alarm
SIGTERM	software termination
SIGUSR1	user-defined signal
SIGUSR2	second user-defined signal
SIGCLD	death of child
SIGPWR	power failure

The interpretation of *function* is as follows:

SIG_DFL	default operation. For all signals except SIGPWR and SIGCLD, process terminates. It creates a core image for signals SIGQUIT, SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, and SIGSYS.
SIG_IGN	ignore the occurrence of the signal.
function	an address of a procedure in the process. The kernel arranges to call the function with the signal number as argument when it returns to user mode. The kernel automatically resets the value of the signal handler to SIG_DFL for all signals except SIGILL, SIGTRAP, and SIGPWR. A process cannot catch SIGKILL signals.

stat

```
stat(filename, statbuf)
char *filename;
struct stat *statbuf;

fstat(fd, statbuf)
int fd;
struct stat *statbuf;
```

Stat returns status information about the specified file. *Fstat* does the same for the open file whose descriptor is *fd*. The structure of *statbuf* is:

```
struct stat {
        dev_t   st_dev;     /* device number for dev containing file */
        ino_t   st_ino;     /* inode number */
        ushort  st_mode;    /* file type (see mknod) and perms (see chmod) */
        short   st_nlink;   /* number of links for file */
        ushort  st_uid;     /* user ID of file's owner */
        ushort  st_gid;     /* group ID of file's group */
        dev_t   st_rdev;    /* major and minor device numbers */
        off_t   st_size;    /* size in bytes */
```

```
        time_t    st_atime;    /* time of last access */
        time_t    st_mtime;    /* time of last modification */
        time_t    st_ctime;    /* time of last status change */
};
```

stime

```
stime(tptr)
long *tptr;
```

Stime sets the system time and date, according to the value pointed to by *tptr*. Times are specified in seconds since 00:00:00 January, 1, 1970, GMT.

sync

```
sync()
```

Sync flushes file system data in system buffers onto disk.

time

```
time(tloc)
long *tloc;
```

Time returns the number of seconds since 00:00:00 January 1, 1970, GMT. If *tloc* is not 0, it will contain the return value, too.

times

```
#include <sys/types.h>
#include <sys/times.h>

times(tbuf)
struct tms *tbuf;
```

Times returns the elapsed real time in clock ticks from an arbitrary fixed time in the recent past, and fills *tbuf* with accounting information:

```
struct tms {
        time_t    tms_utime;    /* CPU time spent in user mode */
        time_t    tms_stime;    /* CPU time spent in kernel mode */
        time_t    tms_cutime;   /* Sum of tms_utime and tms_cutime of children */
        time_t    tms_sutime;   /* Sum of tms_stime and tms_sutime of children */
};
```

ulimit

```
ulimit(cmd, limit)
int cmd;
long limit;
```

Ulimit allows a process to set various limits according to the value of *cmd*:

1 return maximum file size (in 512 byte blocks) the process can write
2 set maximum file size to limit.
3 return maximum possible break value (highest possible address in data region).

umask

```
umask(mask)
int mask;
```

Set the file mode creation *mask* and return the old value. When creating a file, permissions are turned off if the corresponding bits in *mask* are set.

umount

```
umount(specialfile)
*char *specialfile;
```

Unmount the file system in the block special device *specialfile*.

uname

```
#include <sys/utsname.h>

uname(name)
struct utsname *name;
```

Uname returns system-specific information according to the following structure:

```
struct utsname {
        char    sysname[9];     /* name */
        char    nodename[9];    /* network node name */
        char    release[9];     /* system version information */
        char    version[9];     /* more version information */
        char    machine[9];     /* hardware */
};
```

unlink

```
unlink(filename)
char *filename;
```

Remove the directory entry for the indicated file.

ustat

```
#include <sys/types.h>
#include <ustat.h>

ustat(dev, ubuf)
int dev;
struct ustat *ubuf;
```

Ustat returns statistics about the file system identified by *dev* (the major and minor number). The structure *ustat* is defined by:

```
struct ustat {
        daddr_t   f_tfree;      /* number of free blocks */
        ino_t     f_tinode;     /* number of free inodes */
        char      f_fname[6];   /* filsys name */
        char      f_fpack[6];   /* filsys pack name */
};
```

utime

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

utime(filename, times)
char *filename;
struct utimbuf *times;
```

Utime sets the access and modification times of the specified file according to the value of *times*. If 0, the current time is used. Otherwise, *times* points to the following structure:

```
struct utimbuf {
        time_t      axtime;     /* access time */
        time_t      modtime;    /* modification time */
};
```

All times are measured from 00:00:00 January 1, 1970 GMT.

wait

```
wait(wait_stat)
int *wait_stat;
```

Wait causes the process to sleep until it discovers a child process that had exited or a process asleep in trace mode. If *wait_stat* is not 0, it points to an address that contains status information on return from the call. Only the 16 low order bits are written. If wait returns because it found a child process that had exited, the low order 8 bits are 0, and the high order 8 bits contain the low order 8 bits the child process had passed as a parameter to *exit*. If the child exited because of a signal, the high order 8 bits are 0, and the low order 8 bits contain the signal number. In addition, bit 0200 is set if core was dumped. If *wait* returns because it found a traced process, the high order 8 bits (of the 16 bits) contain the signal number that caused it to stop, and the low order 8 bits contain octal 0177.

write

```
write(fd, buf, count)
int fd, count;
char *buf;
```

Write writes *count* bytes of data from user address *buf* to the file whose descriptor is *fd*.

BIBLIOGRAPHY

[Babaoglu 81] Babaoglu, O., and W. Joy, "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the 8th Symposium on Operating Systems Principles, ACM Operating Systems Review*, Vol. 15(5), Dec. 1981, pp. 78-86.

[Bach 84] Bach, M. J., and S. J. Buroff, "Multiprocessor UNIX Systems," *AT&T Bell Laboratories Technical Journal*, Oct. 1984, Vol 63, No. 8, Part 2, pp. 1733-1750.

[Barak 80] Barak, A. B. and A. Shapir, "UNIX with Satellite Processors," *Software - Practice and Experience*, Vol. 10, 1980, pp. 383-392.

[Beck 85] Beck, B. and B. Kasten, "VLSI Assist in Building a Multiprocessor UNIX System," *Proceedings of the USENIX Association Summer Conference*, June 1985, pp. 255-275.

[Berkeley 83] *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, August 1983.

[Birrell 84] Birrell, A.D. and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 39-59.

[Bodenstab 84] Bodenstab, D. E., T. F. Houghton, K. A. Kelleman, G. Ronkin, and E. P. Schan, "UNIX Operating System Porting Experiences," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Oct. 1984, pp. 1769-1790.

[Bourne 78] Bourne, S. R., "The UNIX Shell," *The Bell System Technical Journal*, July-August 1978, Vol. 57, No. 6, Part 2, pp. 1971-1990.