```
algorithm P                /* P semaphore operation */
input:  (1) semaphore
        (2) priority
output: 0 for normal return
        -1 for abnormal wakeup due to signals catching in kernel
        long jumps for signals not catching in kernel
{
      Pprim(semaphore.lock);
      decrement (semaphore.value);
      if (semaphore.value >= 0)
      {
            Vprim(semaphore.lock);
            return(0);
      }
      /* must go to sleep */
      if (checking signals)
      {
            if (there is a signal that interrupts sleep)
            {
                  increment (semaphore.value);
                  if (catching signal in kernel)
                  {
                        Vprim(semaphore.lock);
                        return(-1);
                  }
                  else
                  {
                        Vprim(semaphore.lock);
                        longjmp;
                  }
            }
      }
      enqueue process at end of sleep list of semaphore;
      Vprim(semaphore.lock);
      do context switch;
      check signals, as above;
      return(0);
}
```

Figure 12.8. Algorithm for Implementation of P

to test the semaphore and find its value equal to 0 and for process B on processor B to do a *P*, decrementing the value of the semaphore to −1 (Figure 12.10) just after the test on A. Process A would continue executing, assuming that it had awakened every sleeping process on the semaphore. Hence, the loop does not insure that every sleeping process wakes up, because it is not atomic.

```
algorithm V                /* V semaphore operation */
input:  address of semaphore
output: none
{
        Pprim(semaphore.lock);
        increment (semaphore.value);
        if (semaphore.value <= 0)
        {
                remove first process from semaphore sleep list;
                make it ready to run (wake it up);
        }
        Vprim(semaphore.lock);
}
```

Figure 12.9. Algorithm for Implementation of V

Consider another phenomenon in the use of semaphores on a uniprocessor system. Suppose two processes, A and B, contend for a semaphore: Process A finds the semaphore free and process B sleeps; the value of the semaphore is -1. When process A releases the semaphore with a *V*, it wakes up process B and increments the semaphore value to 0. Now suppose process A, still executing in kernel mode, tries to lock the semaphore again: It will sleep in the *P* function, because the value of the semaphore is 0, even though the resource is still free! The system will incur the expense of an extra context switch. On the other hand, if the lock were implemented by a sleep-lock, process A would gain immediate reuse of the resource, because no other process could lock it in the meantime. In this case, the sleep-lock would be more efficient than a semaphore.

When locking several semaphores, the locking order must be consistent to avoid deadlock. For instance, consider two semaphores, A and B, and consider two kernel algorithms that must have both semaphores simultaneously locked. If the two algorithms were to lock the semaphores in reverse order, a deadlock could arise, as shown in Figure 12.11; process A on processor A locks semaphore SA while process B on processor B locks semaphore SB. Process A attempts to lock semaphore SB, but the *P* operation causes process A to go to sleep, since the value of SB is at most 0. Similarly, process B attempts to lock semaphore SA, but its *P* puts process B to sleep. Neither process can proceed.

Deadlocks can be avoided by implementing deadlock detection algorithms that determine if a deadlock exists and, if so, break the deadlock condition. However, implementation of deadlock detection algorithms would complicate the kernel code. Since there are only a finite number of places in the kernel where a process must simultaneously lock several semaphores, it is easier to implement the kernel algorithms to avoid deadlock conditions before they occur. For instance, if particular sets of semaphores were always locked in the same order, the deadlock condition could never arise. But if it is impossible to avoid locking semaphores in

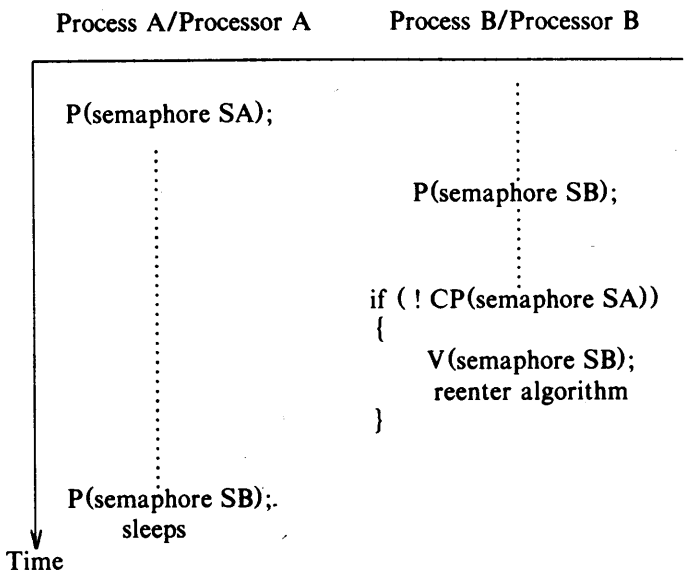

Figure 12.12. Use of Conditional P to Avoid Deadlock

hash queue, because a process would never go to sleep and leave the hash queue in an inconsistent state. In a multiprocessor system, however, two processes could manipulate the linked list of the hash queue; the semaphore for the hash queue permits only one process at a time to manipulate the linked list. Similarly, the free list requires a semaphore because several processes could otherwise corrupt it.

Figure 12.14 depicts the first part of the *getblk* algorithm as implemented with semaphores on a multiprocessor system (recall Figure 3.4). To search the buffer cache for a given block, the kernel locks the hash queue semaphore with a *P* operation. If another process had already done a *P* operation on the semaphore, the executing process sleeps until the original process does a *V*. When it gains exclusive control of the hash queue, it searches for the appropriate buffer. Assume that the buffer is on the hash queue. The kernel (process A) attempts to lock the buffer, but if it were to use a *P* operation and if the buffer was already locked, it would sleep with the hash queue locked, preventing other processes from accessing the hash queue, even though they were searching for other buffers. Instead, process A attempts to lock the buffer using the *CP* operation; if the *CP* succeeds, it can use the buffer. Process A locks the free list semaphore using *CP* in a spin loop, because the expected time the lock is held is short, and hence, it does not pay to sleep with a *P* operation. The kernel then removes the buffer from the free list, unlocks the free list, unlocks the hash queue, and returns the locked buffer.

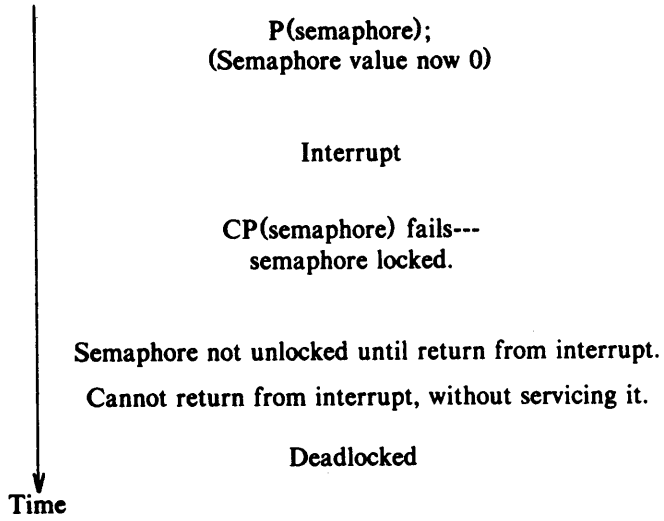

Figure 12.13. Deadlock in Interrupt Handler

Suppose the *CP* operation on the buffer fails because another process had locked the buffer semaphore. Process A releases the hash queue semaphore and then sleeps on the buffer semaphore with a *P* operation. The *P* operates on the semaphore that just caused the *CP* to fail! It does not matter whether process A sleeps on the semaphore: After completion of the *P* operation, process A controls the buffer. Because the rest of the algorithm assumes that the buffer and the hash queue are locked, process A now attempts to lock the hash queue.[1] Because the locking order here (buffer semaphore, then hash queue semaphore) is the opposite of the locking order explained above (hash queue semaphore, then buffer semaphore), the *CP* semaphore operation is used. The obvious processing happens if the lock fails. But if the lock succeeds, the kernel cannot be sure that it has the correct buffer, because another process may have found the buffer on the free list and changed the contents to those of another block before relinquishing control of the buffer semaphore. Process A, waiting for the semaphore to become free, had no idea that the buffer it was waiting for was no longer the one in which it was interested and must therefore check that the buffer is still valid; if not, it restarts the algorithm. If the buffer contains valid data, process A completes the algorithm.

1. The algorithm could avoid locking the hash queue here by setting a flag and testing it before the *V* later on, but this method illustrates the technique for locking semaphores in reversed order.

```
algorithm getblk        /* multiprocessor version */
input:   file system number
         block number
output: locked buffer that can now be used for block
{
        while (buffer not found)
        {
                P(hash queue semaphore);
                if (block in hash queue)
                {
                        if (CP(buffer semaphore) fails)        /* buffer busy */
                        {
                                V(hash queue semaphore);
                                P(buffer semaphore);        /* sleep until free */
                                if (CP(hash queue semaphore) fails)
                                {
                                        V(buffer semaphore);
                                        continue;        /* to while loop */
                                }
                                else if (dev or block num changed)
                                {
                                        V(buffer semaphore);
                                        V(hash queue semaphore);
                                }
                        }
                        while (CP(free list semaphore) fails)
                                ;        /* spin loop */
                        mark buffer busy;
                        remove buffer from free list;
                        V(free list semaphore);
                        V(hash queue semaphore);
                        return buffer;
                }
                else        /* buffer not in hash queue */
                /* remainder of algorithm continues here */
        }
}
```

Figure 12.14. Buffer Allocation with Semaphores

```
multiprocessor algorithm wait
{
        for (;;)        /* loop */
        {
                search all child processes:
                if (status of child is zombie)
                        return;
                P(zombie_semaphore);        /* initialized to 0 */
        }
}
```

Figure 12.15. Multiprocessor Algorithm for Wait/Exit

The remainder of the algorithm is left as an exercise.

12.3.3.2 Wait

Recall from Chapter 7 that a process sleeps in the *wait* system call until a child *exits*. The problem on a multiprocessor system is to make sure that a parent does not miss a zombie child as it executes the *wait* algorithm; for example, if a child *exits* on one processor as the parent executes *wait* on another processor, the parent must not sleep waiting for a second child to *exit*. Each process table entry contains a semaphore *zombie_semaphore*, initialized to 0, where a process sleeps in *wait* until a child *exits* (Figure 12.15). When a process *exits*, it does a *V* on the parent semaphore, awakening the parent if it was sleeping in *wait*. If the child process *exits* before the parent executes *wait*, the parent finds the child in the zombie state and returns. If the two processes execute *exit* and *wait* simultaneously but the child *exits* after the parent already checked its status, the child *V* will prevent the parent from sleeping. At worst, the parent will make an extra iteration through the loop.

12.3.3.3 Drivers

The multiprocessor implementation for the AT&T 3B20A computer avoided inserting semaphores into driver code by doing *P* and *V* operations at the driver entry points (see [Bach 84]). Recall from Chapter 10 that the interface to device drivers is well defined with only a few entry points (about 20, in practice). Drivers are protected by bracketing the entry points, as in:

```
P(driver_semaphore);
open(driver);
V(driver_semaphore):
```

By using the same semaphore for all entry points to a driver and using different semaphores for each driver, at most one process can execute critical code in the driver at a time. The semaphores can be configured per device unit or for classes of devices. For example, a semaphore may be associated with each physical terminal, or one semaphore may be associated with all terminals. The former case is potentially faster, because processes accessing one terminal do not lock the semaphore for other terminals, as in the latter case. However, some device drivers interact internally with other device drivers; in such cases, specifying one semaphore for a class of devices is easier to understand. Alternatively, the 3B20A implementation allows particular devices to be configured such that the driver code runs on specified processors.

Problems could occur when a device interrupts the system when its semaphore is locked: the interrupt handler cannot be invoked, because otherwise there would be danger of corruption. On the other hand, the kernel must make sure that it does not lose an interrupt. The 3B20A queues interrupts until the semaphore is unlocked and it is safe to execute the interrupt handler, and it calls the interrupt handler from the code that unlocks drivers, if necessary.

12.3.3.4 Dummy Processes

When the kernel does a context switch on a uniprocessor, it executes in the context of the process relinquishing control, as explained in Chapter 6. If no processes are ready to run, the kernel idles in the context of the process that last ran. When interrupted by the clock or by other peripherals, it handles the interrupt in the context of the process it had been idling in.

In a multiprocessor system, the kernel cannot idle in the context of the process executed most recently on the processor. For if a process goes to sleep on processor A, consider what happens when the process wakes up: It is ready to run, but it does not execute immediately even though its context is already available on processor A. If processor B now chooses the process for execution, it would do a context switch and resume execution. When processor A emerges from its idle loop as the result of another interrupt, it executes in the context of process A again until it switches context. Thus, for a short period of time, the two processors could be writing the identical address space, particularly, the kernel stack.

The solution to this problem is to create a dummy process per processor; when a processor has no work to do, the kernel does a context switch to the dummy process and the processor idles in the context of its dummy process. The dummy process consists of a kernel stack only; it cannot be scheduled. Since only one processor can idle in its dummy process, processors cannot corrupt each other.

12.4 THE TUNIS SYSTEM

The Tunis system has a user interface that is compatible to that of the UNIX system, but its *nucleus*, written in the language Concurrent Euclid, consists of kernel processes that control each part of the system. The Tunis system solves the mutual exclusion problem because only one instance of a kernel process can run at a time, and because kernel processes do not manipulate the data structures of other processes. Kernel processes are activated by queuing messages for input, and Concurrent Euclid implements *monitors* to prevent corruption of the queues. A monitor is a procedure that enforces mutual exclusion by allowing only one process at a time to execute the body of the procedure. They differ from semaphores because they force modularity (the *P* and *V* are at the entry and exit points of the monitor routine) and because the compiler generates the synchronization primitives. Holt notes that such systems are easier to construct using a language that supports the notion of concurrency and monitors (see page 190 of [Holt 83]). However, the internal structure of the Tunis system differs radically from traditional implementations of the UNIX system.

12.5 PERFORMANCE LIMITATIONS

This chapter has presented two methods that have been used to implement multiprocessor UNIX systems: the master-slave configuration, where only one processor can execute in kernel mode, and a semaphore method that allows all processors to execute in kernel mode simultaneously. The implementations of multiprocessor UNIX systems described in this chapter generalize to any number of processors, but system throughput will not increase at a linear rate with the number of processors. First, there is degradation because of increased memory contention in the hardware, meaning that memory accesses takes longer. Second, in the semaphore scheme, there is increased contention for semaphores; processes find semaphores locked more frequently, more processes queue waiting for semaphores to become free, and therefore processes have to wait a longer period of time to gain access to the semaphore. Similarly, in the master-slave scheme, the master processor becomes a system bottleneck as the number of processors in the system grows, because it is the only processor that can execute kernel code. Although careful hardware design can reduce contention and provide nearly linear increase in system throughput with additional processors for some loads (see [Beck 85], for example), all multiprocessor systems built with current technology reach a limit beyond which the addition of more processors does not increase system throughput.

12.6 EXERCISES

1. Implement a solution to the multiprocessor problem such that any processor in a multiprocessor configuration can execute the kernel but only one processor can do so at

a time. This differs from the first solution discussed in the text, where one processor is designated the master to handle all kernel services. How could such a system make sure that only one processor is in the kernel? What is a reasonable strategy for handling interrupts and still make sure that only one processor is in the kernel?

2. Use the shared memory system calls to test the C code for implementation of semaphores, shown in Figure 12.6. Several independent processes should execute *P-V* sequences on a semaphore. How would you demonstrate a bug in the code?
3. Design an algorithm for *CP* (conditional *P*) along the lines of the algorithm for P.
4. Explain why the algorithms for *P* and *V* in Figure 12.8 and 12.9 must block interrupts. At what points should they be blocked?
5. If a semaphore is used in a spin-lock, as in

```
while (! CP(semaphore));
```

why can the kernel *never* use an unconditional *P* operation on it? (Hint: If a process sleeps on the *P* operation, what happens in the spin-lock?)

6. Refer to the algorithm *getblk* in Chapter 3 and describe a multiprocessor implementation for the case that the block is not in the buffer cache.
* 7. In the buffer allocation algorithm, suppose there is too much contention for the buffer free list semaphore. Implement a scheme to cut down the contention by partitioning the free list into two free lists.
* 8. Suppose a terminal driver has a semaphore, initialized to 0, where processes sleep if they flood the terminal with output. When the terminal can accept more data, it wakes up every process sleeping on the semaphore. Design a scheme to wake up all processes using *P* and *V*. Define other flags and driver locking semaphores, as necessary. If the wakeup results from an interrupt and a processor cannot block interrupts on other processors, how safe can the scheme be?
* 9. When protecting driver entry points with semaphores, provision must be made to release the semaphore when a process sleeps in the driver. Describe an implementation. Similarly, how should the driver handle interrupts that occur when the driver semaphore is locked?
10. Recall the system calls in Chapter 8 for setting and accessing system time. A system cannot assume identical clock rates for different multiprocessors. How should the time system calls work?

13

DISTRIBUTED UNIX SYSTEMS

The previous chapter examined tightly coupled multiprocessor systems that share common memory and kernel data structures and schedule processes from a common pool. However, it is frequently desirable to pool computers to allow resource sharing such that each computer retains autonomy over its environment. For example, a user of a personal computer wants to access files that are stored on a larger machine but wants to retain control of the personal computer. Although several programs such as *uucp* allow file transfer and other applications across a network, their use is not transparent because the user is aware of the network. Furthermore, programs such as text editors do not work on remote files as they do for local files. Users would like to do the normal set of UNIX system calls and, except for a possible degradation in performance, not be aware that they cross a machine boundary. Specifically, system calls such as *open* and *read* should work for files on remote machines just as they do for files on local systems.

Figure 13.1 shows the architecture of a distributed system. Each computer, shown in a circle, is an autonomous unit, consisting of a CPU, memory and peripherals. A computer can fit the model even though it does not have local file storage: It must have peripherals to communicate with other machines, but all its regular files can be on another machine. Most critically, the physical memory available to each machine is independent of activity on other machines. This feature distinguishes distributed systems from the tightly coupled multiprocessor systems described in the last chapter. Consequently, the kernels on each machine

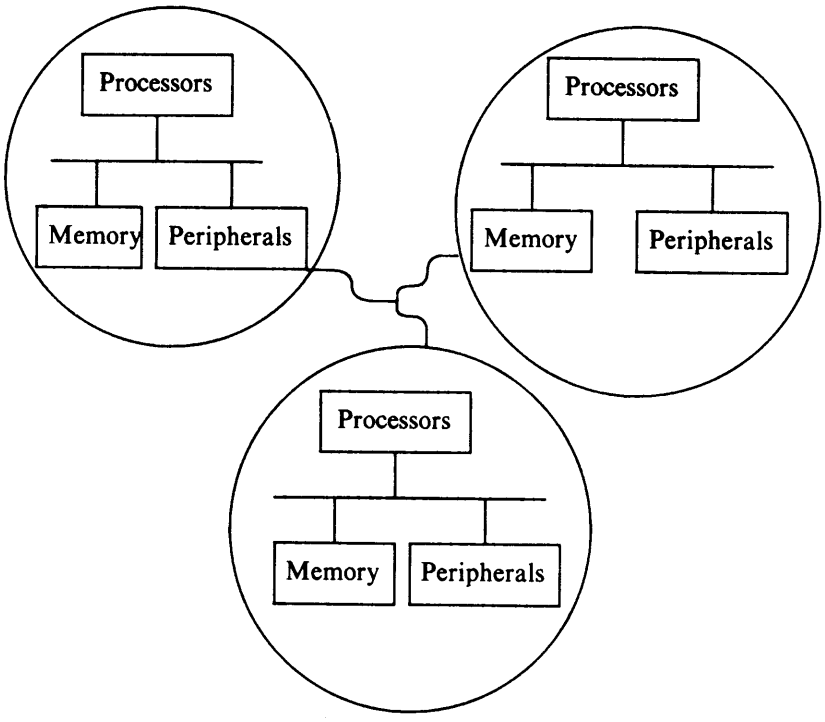

Figure 13.1. Model of Distributed Architectures

are independent, subject to the external constraints of running in a distributed environment.

Many implementations of distributed systems have been described in the literature, falling into the following categories.

- *Satellite* systems are tightly clustered groups of machines that center on one (usually larger) machine. The satellite processors share the process load with the central processor and refer all system calls to it. The purpose of a satellite system is to increase system throughput and, possibly, to allow dedicated use of a processor for one process in a UNIX system environment. The system runs as a unit; unlike other models of distributed systems, satellites do not have real autonomy except, sometimes, in process scheduling and in local memory allocation.
- "Newcastle" distributed systems allow access to remote systems by recognizing names of remote files in the C library. (The name comes from a paper entitled "The Newcastle Connection" — see [Brownbridge 82].) The remote files are designated by special characters embedded in the path name or by special path

component sequences that precede the file system root. This method can be implemented without making changes to the kernel and is therefore easier to implement than the other implementations described in this chapter, but it is less flexible.

- Fully transparent distributed systems allow standard path names to refer to files on other machines; the kernel recognizes that they are remote. Path names cross machine boundaries at mount points, much as they cross file system mount points on disks.

This chapter examines the architecture of each model; the descriptions here are *not* based on particular implementations but on information published in various technical papers. They assume that low-level protocol modules and device drivers take care of addressing, routing, flow control, and error detection and correction and, thus, assume that each model is independent of the underlying network. The system call examples given in the next section for the satellite processor systems work in similar fashion for the Newcastle and transparent models presented in later sections; hence, they will be explained in detail once, and the sections on the other models will concentrate on particular features that most distinguish them.

13.1 SATELLITE PROCESSORS

Figure 13.2 shows the architecture for a satellite processor configuration. The purpose of such a configuration is to improve system throughput by offloading processes from the central processor and executing them on the satellite processors. Each satellite processor has no local peripherals except for those it needs to communicate with the central processor: The file system and all devices are on the central processor. Without loss of generality, assume that all user processes run on a satellite processor and that processes do not migrate between satellite processors; once a process is assigned to a processor, it stays there until it *exit*s. The satellite processor contains a simplified operating system to handle local system calls, interrupts, memory management, network protocols, and a driver for the device it uses to communicate with the central processor.

When the system is initialized, the kernel on the central processor downloads a local operating system into each satellite processor, which continues to run there until the system is taken down. Each process on a satellite processor has an associated *stub* process on the central processor (see [Birrell 84]); when a process on a satellite processor makes a system call that requires services provided only by the central processor, the satellite process communicates with its stub on the central processor to satisfy the request. The stub executes the system call and sends the results back to the satellite processor. The satellite process and its stub enjoy a client-server relationship similar to those described in Chapter 11: The satellite is the client of the stub, which provides file system services. The term *stub* emphasizes that the remote server process serves only one client process. Section 13.4 considers server processes that serve several client processes. For convenience,

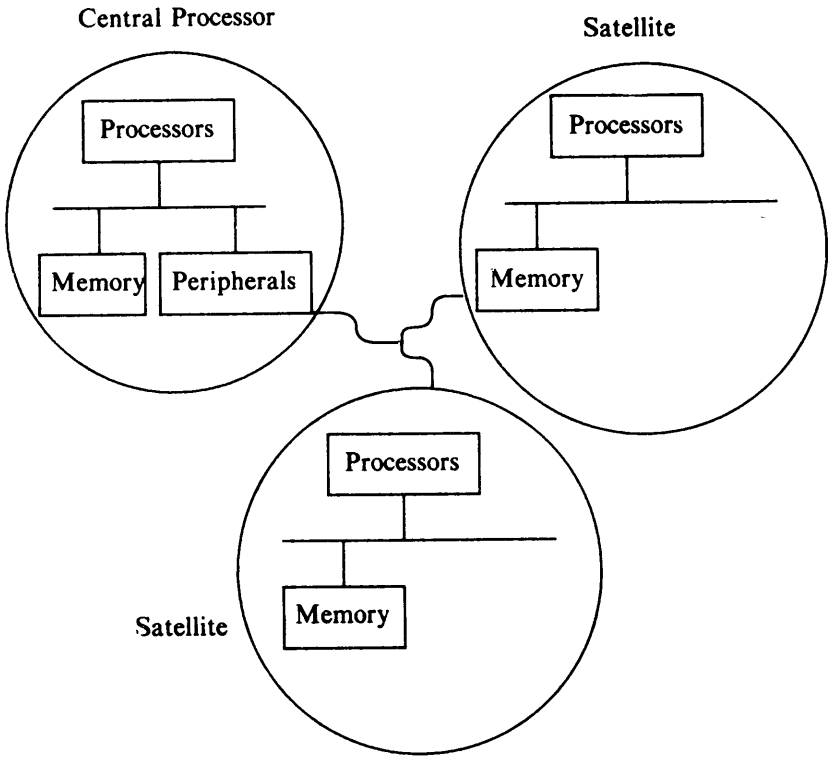

Figure 13.2. Satellite Processor Configuration

the term *satellite process* will refer to a process running on a satellite processor.

When a satellite process makes a system call that can be handled locally, the kernel does not have to send a request to the stub process. For example, it can execute the *sbrk* system call locally to obtain more memory for a process. But if it needs to obtain service from the central processor, such as when *open*ing a file, it encodes the parameters of the system call and the process environment into a message that it sends to the stub process (Figure 13.3). The message consists of a token that specifies the system call the stub should make on behalf of the client, parameters to the system call, and environmental data such as user ID and group ID, which may vary per system call. The remainder of the message contains variable length data, such as a file path name or data for a *write* system call.

The stub waits for requests from the satellite process; when it receives a request, it decodes the message, determines what system call it should invoke, executes the system call, and encodes the results of the system call into a response for the satellite process. The response contains the return values to be returned to the

Message Format

Token for Syscall	Syscall Parameters	Environment Data	Path Name ..... or ..... Data Stream

Response

Syscall Return Values	Error Code	Signal Number	..... Data Stream .....

Figure 13.3. Message Formats

calling process as the result of the system call, an error code to report errors in the stub, a signal number, and a variable length data array to contain data read from a file, for example. The satellite process sleeps in the system call until it receives the response, decodes it, and returns the results to the user. This is the general scheme for handling system calls; the remainder of this section examines particular system calls in greater detail.

To explain how the satellite system works, consider the following system calls: *getppid*, *open*, *write*, *fork*, *exit* and *signal*. The *getppid* system call is simple, because it requires a simple request and response between the satellite and central processors. The kernel on the satellite processor forms a message with a token that indicates that the system call was *getppid*, and sends the request to the central processor. The stub on the central processor *read*s the message from the satellite processor, decodes the system call type, executes the *getppid* system call, and finds its parent process ID. It then forms a response and *write*s it to the satellite process, which had been waiting, *read*ing the communication link. When the satellite receives the answer from the stub, it returns the result to the process that had originally invoked the *getppid* system call. Alternatively, if the satellite process retains data such as the parent process ID locally, it need not communicate with its stub at all.

For the *open* system call, the satellite process sends an *open* message to the stub process, including the file name and other parameters. Assuming the stub does the *open* call successfully, it allocates an inode and file table entry on the central processor, assigns an entry in the user file descriptor table in its *u area*, and returns the file descriptor to the satellite process. Meanwhile, the satellite process had been *read*ing the communications link, waiting for the response from the stub process. The satellite process has no kernel data structures that record information about

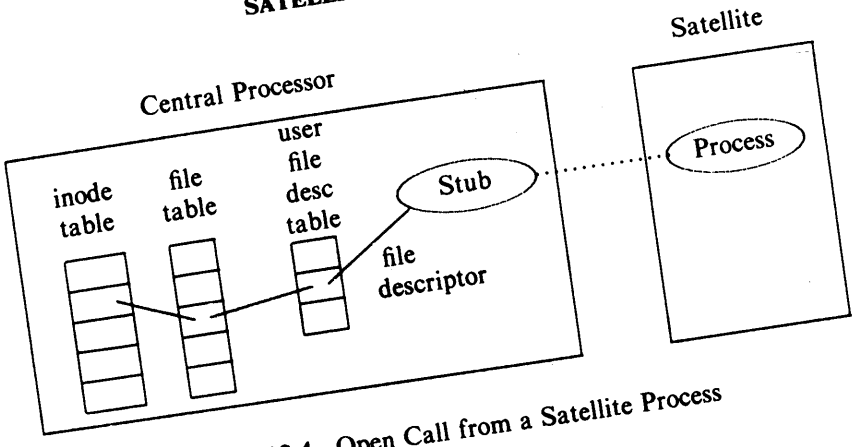

Figure 13.4. Open Call from a Satellite Process

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