cannot go to sleep, because they would be putting an arbitrary process to sleep (the one that was interrupted). Modules must save their state information internally, making their code more cumbersome than it would be if sleeping were allowed.

Several anomalies exist in the implementation of streams.

- Process accounting is difficult under streams, because modules do not necessarily run in the context of the process that is using the stream. It is false to assume that all processes uniformly share execution of streams modules, because some processes may require use of complicated network protocols, whereas others may use simple terminal line disciplines.
- Users can put a terminal driver into raw mode, such that *read* calls return after a short time if no data is available (for example, if *newtty.c_cc[VMIN] = 0;* in Figure 10.17). It is difficult to implement this feature with streams, unless special-case code is introduced at the stream-head level.
- Streams are linear connections and do not easily allow multiplexing in the kernel. For example, the window example in the previous section does the multiplexing in a user-level process.

In spite of these anomalies, streams holds great promise for improving the design of driver modules.

10.5 SUMMARY

This chapter presented an overview of device drivers on the UNIX system. Devices are either block devices or character devices; the interface between them and the rest of the kernel depends on the device type. The block device interface is the block device switch table, which consists of entry points for device open, close, and strategy procedures. The strategy procedure controls data transfer to and from the block device. The character device interface is the character device switch table, which consists of entry points for device open, close, read, write, and ioctl procedures. The *ioctl* system call uses the *ioctl* interface to character devices, which permits control information to be sent between processes and devices. The kernel calls device interrupt handlers on receipt of a device interrupt, based on information stored in the interrupt vector table and on parameters supplied by the interrupting hardware.

Disk drivers convert logical block numbers used by the file system to locations on the physical disk. The block interface allows the kernel to buffer data. The raw interface allows faster I/O to and from the disk but bypasses the buffer cache, allowing more chances for file system corruption.

Terminal drivers support the primary interface to users. The kernel associates three clists with each terminal, one for raw input from the keyboard, one for processed input to account for erase and kill characters and carriage returns, and one for output. The *ioctl* system call allows processes to control how the kernel treats input data, placing the terminal in canonical mode or setting various parameters for raw mode. The *getty* process *open*s terminal lines and waits for a

connection: It sets its process group so that the login shell is eventually a process group leader, initializes terminal parameters via *ioctl*, and prompts the user through a login sequence. The control terminal thus set up sends signals to processes in the process group, in response to events such as when the user hangs up or presses the break key.

Streams are a scheme for improving the modularity of device drivers and protocols. A stream is a full-duplex connection between processes and device drivers, which may contain line disciplines and protocols to process data en route. Streams modules are characterized by well-defined interfaces and by their flexibility for use in combination with other modules. The flexibility they offer has strong benefits for network protocols and drivers.

10.6 EXERCISES

* 1. Suppose a system contains two device files that have the same major and minor number and are both character devices. If two processes wish to *open* the physical device simultaneously, show that it makes no difference whether they *open* the same device file or different device files. What happens when they *close* the device?
* 2. Recall from Chapter 5 that the *mknod* system call requires superuser permission to create a device special file. Given that device access is governed by the permission modes of a file, why must *mknod* require superuser permission?
3. Write a program that verifies that the file systems on a disk do not overlap. The program should take two arguments: a device file that represents a disk volume and a descriptor file that gives section numbers and section lengths for the disk type. The program should read the super blocks to make sure that file systems do not overlap. Will such a program always be correct?
4. The program *mkfs* initializes a file system on a disk by creating the super block, leaving space for the inode list, putting all the data blocks on a linked list, and making the root inode directory. How would you program *mkfs*? How does the program change if there is a volume table of contents? How should it initialize the volume table of contents?
5. The programs *mkfs* and *fsck* (Chapter 5) are user-level programs instead of part of the kernel. Comment.
6. Suppose a programmer wants to write a data base system to run on the UNIX system. The data base programs run at user level, not as part of the kernel. How should the system interact with the disk? Consider the following issues:
   - Use of the regular file system interface versus the raw disk,
   - Need for speed,
   - Need to know when data actually resides on disk,
   - Size of the data base: Does it fit into one file system, an entire disk volume, or several disk volumes?
7. The UNIX kernel tacitly assumes that the file system is contained on perfect disks. However, disks could contain faults that incapacitate certain sectors although the remainder of the disk is still "good." How could a disk driver (or intelligent disk controller) make allowances for small numbers of bad sectors. How would this affect performance?

8. When *mount*ing a file system, the kernel invokes the driver open procedure but later releases the inode for the device special file at the end of the *mount* call. When *umount*ing a file system, the kernel accesses the inode of the device special file, invokes the driver close procedure, and releases the inode. Compare the sequence of inode operations and driver open and close calls to the sequence when *open*ing and *clos*ing a block device. Comment.
9. Run the program in Figure 10.14 but direct the output to a file. Compare the contents of the file to the output when output goes to the terminal. You will have to interrupt the processes to stop them; let them run long enough to get a sufficient amount of output. What happens if the *write* call in the program is replaced with

   ```
   printf(output);
   ```

10. What happens when a user attempts to do text editing in the background:

    ```
    ed file &
    ```

    Why?
11. Terminal files typically have access permissions set as in

    ```
    crw--w--w-  2 mjb     lus     33, 11 Oct 25 20:27   tty61
    ```

    when a user is logged on. That is, read/write permission is permitted for user "mjb," but only write permission is allowed other users. Why?
12. Assuming you know the terminal device file name of a friend, write a program that allows you to write messages to your friend's terminal. What other information do you need to encode a reasonable facsimile of the usual *write* command?
13. Implement the *stty* command: with no parameters, it retrieves the values of terminal settings and reports them to the user. Otherwise, the user can set various settings interactively.
14. Encode a line discipline that writes the machine name at the beginning of each line of output.
15. In canonical mode, a user can temporarily stop output to a terminal by typing "control s" at the terminal and resume output by typing "control q." How should the standard line discipline implement this feature?
* 16. The *init* process spawns a *getty* process for each terminal line in the system. What would happen if two *getty* processes were to exist simultaneously for one terminal, waiting for a user to log in? Can the kernel prevent this?
17. Suppose the shell were coded so that it "ignored" the end of file and continued to *read* its standard input. What would happen when a user (in the login shell) hits end of file and continues typing?
* 18. Suppose a process *read*s its control terminal but ignores or catches hangup signals. What happens when the process continues to *read* the control terminal after a hangup?
19. The *getty* program is responsible for opening a terminal line, and *login* is responsible for checking login and password information. What advantages are there for doing the two functions in separate programs?
20. Consider the two methods for implementing the indirect terminal driver ("/dev/tty"), described in Section 10.3.6. What differences would a user perceive? (Hint: Think about the system calls *stat* and *fstat*.)
21. Design a method for scheduling streams modules, where the kernel contains a special process that executes module service procedures when they are scheduled to execute.

* 22. Design a scheme for virtual terminals (windows) using conventional (nonstreams) drivers.
* 23. Design a method for implementing virtual terminals using streams such that a kernel module, rather than a user process, multiplexes I/O between the virtual and physical terminals. Describe a mechanism for connecting the streams to allow fan-in and fan-out. Is it better to put a multiplexing module inside the kernel or construct it as a user process?
24. The command *ps* reports interesting information on process activity in a running system. In traditional implementations, *ps* reads the information in the process table directly from kernel memory. Such a method is unstable in a development environment where the size of process table entries changes and *ps* cannot easily find the correct fields in the process table. Encode a driver that is impervious to a changing environment.

11

INTERPROCESS COMMUNICATION

Interprocess communication mechanisms allow arbitrary processes to exchange data and synchronize execution. We have already considered several forms of interprocess communication, such as pipes, named pipes, and signals. Pipes (unnamed) suffer from the drawback that they are known only to processes which are descendants of the process that invoked the *pipe* system call: Unrelated processes cannot communicate via pipes. Although named pipes allow unrelated processes to communicate, they cannot generally be used across a network (see Chapter 13), nor do they readily lend themselves to setting up multiple communications paths for different sets of communicating processes: it is impossible to multiplex a named pipe to provide private channels for pairs of communicating processes. Arbitrary processes can also communicate by sending signals via the *kill* system call, but the "message" consists only of the signal number.

This chapter describes other forms of interprocess communication. It starts off by examining process tracing, whereby one process traces and controls the execution of another process and then explains the the System V IPC package: messages, shared memory, and semaphores. It reviews the traditional methods by which processes communicate with processes on other machines over a network and, finally, gives a user-level overview of BSD sockets. It does not discuss network-specific issues such as protocols, addressing, and name service, which are beyond the scope of this book.

11.1 PROCESS TRACING

The UNIX system provides a primitive form of interprocess communication for tracing processes, useful for debugging. A debugger process, such as *sdb*, spawns a process to be traced and controls its execution with the *ptrace* system call, setting and clearing break points, and reading and writing data in its virtual address space. Process tracing thus consists of synchronization of the debugger process and the traced process and controlling the execution of the traced process.

```
if ((pid = fork()) == 0)
{
        /* child - traced process */
        ptrace(0, 0, 0, 0);
        exec("name of traced process here");
}
/* debugger process continues here */
for (;;)
{
        wait((int *) 0);
        read(input for tracing instructions)
        ptrace(cmd, pid, ...);
        if (quitting trace)
                break;
}
```

Figure 11.1. Structure of Debugging Process

The pseudo-code in Figure 11.1 shows the typical structure of a debugger program. The debugger spawns a child process, which invokes the *ptrace* system call and, as a result, the kernel sets a trace bit in the child process table entry. The child now *exec*s the program being traced. For example, if a user is debugging the program *a.out*, the child would *exec a.out*. The kernel executes the *exec* call as usual, but at the end notes that the trace bit is set and sends the child a "trap" signal. The kernel checks for signals when returning from the *exec* system call, just as it checks for signals after any system call, finds the "trap" signal it had just sent itself, and executes code for process tracing as a special case for handling signals. Noting that the trace bit is set in its process table entry, the child awakens the parent from its sleep in the *wait* system call (as will be seen), enters a special trace state similar to the sleep state (not shown in the process state diagram in Figure 6.1), and does a context switch.

Typically, the parent (debugger) process would have meanwhile entered a user-level loop, *wait*ing to be awakened by the traced process. When the traced process awakens the debugger, the debugger returns from *wait*, *read*s user input commands, and converts them to a series of *ptrace* calls to control the child (traced) process. The syntax of the *ptrace* system call is

ptrace(cmd, pid, addr, data);

where *cmd* specifies various commands such as reading data, writing data, resuming execution and so on, *pid* is the process ID of the traced process, *addr* is the virtual address to be read or written in the child process, and *data* is an integer value to be written. When executing the *ptrace* system call, the kernel verifies that the debugger has a child whose ID is *pid* and that the child is in the traced state and then uses a global trace data structure to transfer data between the two processes. It locks the trace data structure to prevent other tracing processes from overwriting it, copies *cmd*, *addr*, and *data* into the data structure, wakes up the child process and puts it into the "ready-to-run" state, then sleeps until the child responds. When the child resumes execution (in kernel mode), it does the appropriate trace command, writes its reply into the trace data structure, then awakens the debugger. Depending on the command type, the child may reenter the trace state and wait for a new command or return from handling signals and resume execution. When the debugger resumes execution, the kernel saves the "return value" supplied by the traced process, unlocks the trace data structure, and returns to the user.

If the debugger process is not sleeping in the *wait* system call when the child enters the trace state, it will not discover its traced child until it calls *wait*, at which time it returns immediately and proceeds as just described.

```
int data[32];
main()
{
        int i;
        for (i = 0;  i < 32;  i++)
                printf("data[%d] = %d\n", i, data[i]);
        printf("ptrace data addr 0x%x\n", data);
}
```

Figure 11.2. Trace — A Traced Process

Consider the two programs in Figures 11.2 and 11.3, called *trace* and *debug*, respectively. Running *trace* at the terminal, the array values for *data* will be 0; the process prints the address of *data* and exits. Now, running *debug* with a parameter equal to the value printed out by *trace*, *debug* saves the parameter in *addr*, creates a child process that invokes *ptrace* to make itself eligible for tracing, and execs *trace*. The kernel sends the child process (call it *trace*) a *SIGTRAP* signal at the end of *exec*, and *trace* enters the trace state, waiting for a command from *debug*. If *debug* had been sleeping in *wait*, it wakes up, finds the traced child process, and returns from *wait*. *Debug* then calls *ptrace*, writes the value of the loop variable *i* into the data space of *trace* at address *addr*, and increments *addr*; in *trace*, *addr* is an address of an entry in the array *data*. *Debug*'s last call to *ptrace* causes *trace* to run, and this time, the array *data* contains the values 0 to

```
#define TR_SETUP 0
#define TR_WRITE 5
#define TR_RESUME 7
int addr;

main(argc, argv)
        int argc;
        char *argv[];
{
        int i, pid;

        sscanf(argv[1], "%x", &addr);

        if ((pid = fork()) == 0)
        {
                ptrace(TR_SETUP, 0, 0, 0);
                execl("trace", "trace", 0);
                exit();
        }
        for (i = 0;  i < 32;  i++)
        {
                wait((int *) 0);
                /* write value of i into address addr in proc pid */
                if (ptrace(TR_WRITE, pid, addr, i) == -1)
                        exit();
                addr += sizeof(int);
        }
        /* traced process should resume execution */
        ptrace(TR_RESUME, pid, 1, 0);
}
```

Figure 11.3. Debug — A Tracing Process

31. A debugger such as *sdb* has access to the traced process's symbol table, from which it determines the addresses it uses as parameters to *ptrace* calls.

The use of *ptrace* for process tracing is primitive and suffers several drawbacks.

- The kernel must do four context switches to transfer a word of data between a debugger and a traced process: The kernel switches context in the debugger in the *ptrace* call until the traced process replies to a query, switches context to and from the traced process, and switches context back to the debugger process with the answer to the *ptrace* call. The overhead is necessary, because a debugger has no other way to gain access to the virtual address space of a traced process, but process tracing is consequently slow.

- A debugger process can trace several child processes simultaneously, although this feature is rarely used in practice. More critically, a debugger can only trace child processes: If a traced child *fork*s, the debugger has no control over the grandchild, a severe handicap when debugging sophisticated programs. If a traced process *exec*s, the later *exec*ed images are still being traced because of the original *ptrace*, but the debugger may not know the name of the *exec*ed image, making symbolic debugging difficult.
- A debugger cannot trace a process that is already executing if the debugged process had not called *ptrace* to let the kernel know that it consents to be traced. This is inconvenient, because a process that needs debugging must be killed and restarted in trace mode.
- It is impossible to trace *setuid* programs, because users could violate security by writing their address space via *ptrace* and doing illegal operations. For example, suppose a *setuid* program calls *exec* with file name "privatefile". A clever user could use *ptrace* to overwrite the file name with "/bin/sh", executing the shell (and all programs executed by the shell) with unauthorized permission. *Exec* ignores the *setuid* bit if the process is traced to prevent a user from overwriting the address space of a *setuid* program.

Killian [Killian 84] describes a different scheme for process tracing, based on the file system switch described in Chapter 5. An administrator mounts a file system, "/proc"; users identify processes by their PID and treat them as files in "/proc". The kernel gives permission to *open* the files according to the process user ID and group ID. Users can examine the process address space by *read*ing the file, and they can set breakpoints by *writ*ing the file. *Stat* returns various statistics about the process. This method removes three disadvantages of *ptrace*. First, it is faster, because a debugger process can transfer more data per system call than it can with *ptrace*. Second, a debugger can trace arbitrary processes, not necessarily a child process. Finally, the traced process does not have to make prior arrangement to allow tracing; a debugger can trace existing processes. As part of the regular file protection mechanism, only a superuser can debug processes that are *setuid* to root.

11.2 SYSTEM V IPC

The UNIX System V IPC package consists of three mechanisms. Messages allow processes to send formatted data streams to arbitrary processes, shared memory allows processes to share parts of their virtual address space, and semaphores allow processes to synchronize execution. Implemented as a unit, they share common properties.

- Each mechanism contains a table whose entries describe all instances of the mechanism.
- Each entry contains a numeric *key*, which is its user-chosen name.

- Each mechanism contains a "get" system call to create a new entry or to retrieve an existing one, and the parameters to the calls include a key and flags. The kernel searches the proper table for an entry named by the key. Processes can call the "get" system calls with the key *IPC_PRIVATE* to assure the return of an unused entry. They can set the *IPC_CREAT* bit in the flag field to create a new entry if one by the given key does not already exist, and they can force an error notification by setting the *IPC_EXCL* and *IPC_CREAT* flags, if an entry already exists for the key. The "get" system calls return a kernel-chosen descriptor for use in the other system calls and are thus analogous to the file system *creat* and *open* calls.
- For each IPC mechanism, the kernel uses the following formula to find the index into the table of data structures from the descriptor:

  index = descriptor modulo (number of entries in table)

  For example, if the table of message structures contains 100 entries, the descriptors for entry 1 are 1, 101, 201, and so on. When a process removes an entry, the kernel increments the descriptor associated with it by the number of entries in the table: The incremented value becomes the new descriptor for the entry when it is next allocated by a "get" call. Processes that attempt to access the entry by its old descriptor fail on their access. Referring to the previous example, if the descriptor associated with message entry 1 is 201 when it is removed, the kernel assigns a new descriptor, 301, to the entry. Processes that attempt to access descriptor 201 receive an error, because it is no longer valid. The kernel eventually recycles descriptor numbers, presumably after a long time lapse.
- Each IPC entry has a permissions structure that includes the user ID and group ID of the process that created the entry, a user and group ID set by the "control" system call (below), and a set of read-write-execute permissions for user, group, and others, similar to the file permission modes.
- Each entry contains other status information, such as the process ID of the last process to update the entry (send a message, receive a message, attach shared memory, and so on), and the time of last access or update.
- Each mechanism contains a "control" system call to query status of an entry, to set status information, or to remove the entry from the system. When a process queries the status of an entry, the kernel verifies that the process has read permission and then copies data from the table entry to the user address. Similarly, to set parameters on an entry, the kernel verifies that the user ID of the process matches the user ID or the creator user ID of the entry or that the process is run by a superuser; write permission is not sufficient to set parameters. The kernel copies the user data into the table entry, setting the user ID, group ID, permission modes, and other fields dependent on the type of mechanism. The kernel does not change the creator user and group ID fields, so the user who created an entry retains control rights to it. Finally, a user can remove an entry if it is the superuser or if its process ID matches either ID field

in the entry structure. The kernel increments the descriptor number so that the next instance of assigning the entry will return a different descriptor. Hence, system calls will fail if a process attempts to access an entry by an old descriptor. as explained earlier.

11.2.1 Messages

There are four system calls for messages: *msgget* returns (and possibly creates) a message descriptor that designates a message queue for use in other system calls, *msgctl* has options to set and return parameters associated with a message descriptor and an option to remove descriptors, *msgsnd* sends a message, and *msgrcv* receives a message.

The syntax of the *msgget* system call is

```
msgqid = msgget(key, flag);
```

where *msgqid* is the descriptor returned by the call, and *key* and *flag* have the semantics described above for the general "get" calls. The kernel stores messages on a linked list (queue) per descriptor, and it uses *msgqid* as an index into an array of message queue headers. In addition to the general IPC permissions field mentioned above, the queue structure contains the following fields:

- Pointers to the first and last messages on a linked list;
- The number of messages and the total number of data bytes on the linked list;
- The maximum number of bytes of data that can be on the linked list;
- The process IDs of the last processes to send and receive messages;
- Time stamps of the last *msgsnd*, *msgrcv*, and *msgctl* operations.

When a user calls *msgget* to create a new descriptor, the kernel searches the array of message queues to see if one exists with the given key. If there is no entry for the specified key, the kernel allocates a new queue structure, initializes it, and returns an identifier to the user. Otherwise, it checks permissions and returns.

A process uses the *msgsnd* system call to send a message:

```
msgsnd(msgqid, msg, count, flag);
```

where *msgqid* is the descriptor of a message queue typically returned by a *msgget* call, *msg* is a pointer to a structure consisting of a user-chosen integer type and a character array, *count* gives the size of the data array, and *flag* specifies the action the kernel should take if it runs out of internal buffer space.

The kernel checks (Figure 11.4) that the sending process has write permission for the message descriptor, that the message length does not exceed the system limit, that the message queue does not contain too many bytes, and that the message type is a positive integer. If all tests succeed, the kernel allocates space for the message from a message *map* (recall Section 9.1) and copies the data from user space. The kernel allocates a message header and puts it on the end of the linked list of message headers for the message queue. It records the message type and

```
algorithm msgsnd                /* send a message */
input:  (1) message queue descriptor
        (2) address of message structure
        (3) size of message
        (4) flags
output: number of bytes sent
{
      check legality of descriptor, permissions;
      while (not enough space to store message)
      {
            if (flags specify not to wait)
                  return;
            sleep(until event enough space is available);
      }
      get message header;
      read message text from user space to kernel;
      adjust data structures: enqueue message header,
                  message header points to data,
                  counts, time stamps, process ID;
      wakeup all processes waiting to read message from queue;
}
```

Figure 11.4. Algorithm for Msgsnd

size in the message header, sets the message header to point to the message data, and updates various statistics fields (number of messages and bytes on queue, time stamps and process ID of sender) in the queue header. The kernel then awakens processes that were asleep, waiting for messages to arrive on the queue. If the number of bytes on the queue exceeds the queue's limit, the process sleeps until other messages are removed from the queue. If the process specified not to wait (flag IPC_NOWAIT), however, it returns immediately with an error indication. Figure 11.5 depicts messages on a queue, showing queue headers, linked lists of message headers, and pointers from the message headers to a data area.

Consider the program in Figure 11.6: A process calls *msgget* to get a descriptor for *MSGKEY*. It sets up a message of length 256 bytes, although it uses only the first integer, copies its process ID into the message text, assigns the message type value 1, then calls *msgsnd* to send the message. We will return to this example later.

A process receives messages by

```
count = msgrcv(id, msg, maxcount, type, flag);
```

where *id* is the message descriptor, *msg* is the address of a user structure to contain the received message, *maxcount* is the size of the data array in *msg*, *type* specifies the message type the user wants to read, and *flag* specifies what the kernel should

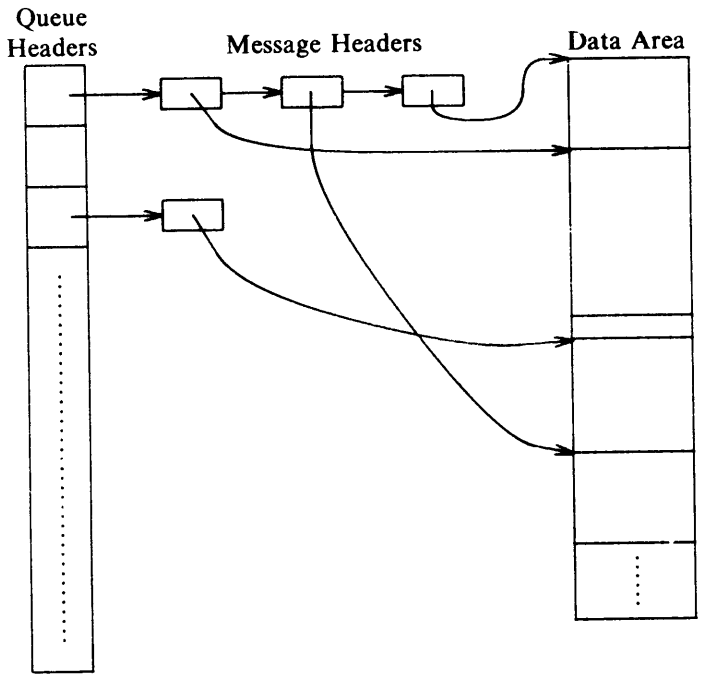

Figure 11.5. Data Structures for Messages

do if no messages are on the queue. The return value, *count*, is the number of bytes returned to the user.

The kernel checks (Figure 11.7) that the user has the necessary access rights to the message queue, as above. If the requested message *type* is 0, the kernel finds the first message on the linked list. If its size is less than or equal to the size requested by the user, the kernel copies the message data to the user data structure and adjusts its internal structures appropriately: It decrements the count of messages on the queue and the number of data bytes on the queue, sets the receive time and receiving process ID, adjusts the linked list, and frees the kernel space that had stored the message data. If processes were waiting to send messages because there was no room on the list, the kernel awakens them. If the message is bigger than *maxcount* specified by the user, the kernel returns an error for the system call and leaves the message on the queue. If the process ignores size constraints, however (bit *MSG_NOERROR* is set in *flag*), the kernel truncates the message, returns the requested number of bytes, and removes the entire message from the list.

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

#define MSGKEY      75

struct msgform {
        long    mtype;
        char    mtext[256];
};

main()
{
        struct msgform msg;
        int msgid, pid, *pint;

        msgid = msgget(MSGKEY, 0777);

        pid = getpid();
        pint = (int *) msg.mtext;
        *pint = pid;        /* copy pid into message text */
        msg.mtype = 1;

        msgsnd(msgid, &msg, sizeof(int), 0);
        msgrcv(msgid, &msg, 256, pid, 0);       /* pid is used as the msg type */
        printf("client: receive from pid %d\n", *pint);
}
```

Figure 11.6. A Client Process

A process can receive messages of a particular type by setting the *type* parameter appropriately. If it is a positive integer, the kernel returns the first message of the given type. If it is negative, the kernel finds the lowest type of all messages on the queue, provided it is less than or equal to the absolute value of *type*, and returns the first message of that type. For example, if a queue contains three messages whose types are 3, 1, and 2, respectively, and a user requests a message with type −2, the kernel returns the message of type 1. In all cases, if no messages on the queue satisfy the receive request, the kernel puts the process to sleep, unless the process had specified to return immediately by setting the *IPC_NOWAIT* bit in *flag*.

Consider the programs in Figures 11.6 and 11.8. The program in Figure 11.8 shows the structure of a *server* that provides generic service to *client* processes. For instance, it may receive requests from client processes to provide information from a database; the server process is a single point of access to the database, making consistency and security easier. The server creates a message structure by setting

```
algorithm msgrcv                /* receive message */
input:  (1) message descriptor
        (2) address of data array for incoming message
        (3) size of data array
        (4) requested message type
        (5) flags
output: number of bytes in returned message
{
      check permissions;
   loop:
      check legality of message descriptor;
      /* find message to return to user */
      if (requested message type == 0)
            consider first message on queue;
      else if (requested message type > 0)
            consider first message on queue with given type;
      else    /* requested message type < 0 */
            consider first of the lowest typed messages on queue,
                  such that its type is <= absolute value of
                  requested type;
      if (there is a message)
      {
            adjust message size or return error if user size too small;
            copy message type, text from kernel space to user space;
            unlink message from queue;
            return;
      }
      /* no message */
      if (flags specify not to sleep)
            return with error;
      sleep (event message arrives on queue);
      goto loop;
}
```

Figure 11.7. Algorithm for Receiving a Message

the *IPC_CREAT* flag in the *msgget* call and receives all messages of type 1 — requests from client processes. It reads the message text, finds the process ID of the client process, and sets the return message type to the client process ID. In this example, it sends its process ID back to the client process in the message text, and the client process receives messages whose message type equals its process ID. Thus, the server process receives only messages sent to it by client processes, and client processes receive only messages sent to them by the server. The processes cooperate to set up multiple channels on one message queue.

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

#define MSGKEY      75
struct msgform
{
      long mtype;
      char mtext[256];
} msg;
int msgid;

main()
{
      int i, pid, *pint;
      extern cleanup();

      for (i = 0;  i < 20;  i++)
            signal(i, cleanup);
      msgid = msgget(MSGKEY, 0777 | IPC_CREAT);

      for (;;)
      {
            msgrcv(msgid, &msg, 256, 1, 0);
            pint = (int *) msg.mtext;
            pid = *pint;
            printf("server: receive from pid %d\n", pid);
            msg.mtype = pid;
            *pint = getpid();
            msgsnd(msgid, &msg, sizeof(int), 0);
      }
}

cleanup()
{
      msgctl(msgid, IPC_RMID, 0);
      exit();
}
```

Figure 11.8. A Server Process

Messages are formatted as type-data pairs, whereas file data is a byte stream. The *type* prefix allows processes to select messages of a particular type, if desired, a feature not readily available in the file system. Processes can thus extract messages of particular types from the message queue in the order that they arrive, and the kernel maintains the proper order. Although it is possible to implement a message

passing scheme at user level with the file system, messages provide applications with a more efficient way to transfer data between processes.

A process can query the status of a message descriptor, set its status, and remove a message descriptor with the *msgctl* system call. The syntax of the call is

```
msgctl(id, cmd, mstatbuf)
```

where *id* identifies the message descriptor, *cmd* specifies the type of command, and *mstatbuf* is the address of a user data structure that will contain control parameters or the results of a query. The implementation of the system call is straightforward; the appendix specifies the parameters in detail.

Returning to the server example in Figure 11.8, the process catches signals and calls the function *cleanup* to remove the message queue from the system. If it did not catch signals or if it receives a *SIGKILL* signal (which cannot be caught), the message queue would remain in the system even though no processes refer to it. Subsequent attempts to create (exclusively) a new message queue for the given key would fail until it was removed.

11.2.2 Shared Memory

Processes can communicate directly with each other by sharing parts of their virtual address space and then reading and writing the data stored in the *shared memory*. The system calls for manipulating shared memory are similar to the system calls for messages. The *shmget* system call creates a new region of shared memory or returns an existing one, the *shmat* system call logically attaches a region to the virtual address space of a process, the *shmdt* system call detaches a region from the virtual address space of a process, and the *shmctl* system call manipulates various parameters associated with the shared memory. Processes read and write shared memory using the same machine instructions they use to read and write regular memory. After attaching shared memory, it becomes part of the virtual address space of a process, accessible in the same way other virtual addresses are; no system calls are needed to access data in shared memory.

The syntax of the *shmget* system call is

```
shmid = shmget(key, size, flag);
```

where *size* is the number of bytes in the region. The kernel searches the shared memory table for the given *key*: if it finds an entry and the permission modes are acceptable, it returns the descriptor for the entry. If it does not find an entry and the user had set the *IPC_CREAT* flag to create a new region, the kernel verifies that the size is between system-wide minimum and maximum values and then allocates a region data structure using algorithm *allocreg* (Section 6.5.2). The kernel saves the permission modes, size, and a pointer to the region table entry in the shared memory table (Figure 11.9) and sets a flag there to indicate that no memory is associated with the region. It allocates memory (page tables and so on) for the region only when a process attaches the region to its address space. The

kernel also sets a flag on the region table entry to indicate that the region should not be freed when the last process attached to it *exits*. Thus, data in shared memory remains intact even though no processes include it as part of their virtual address space.

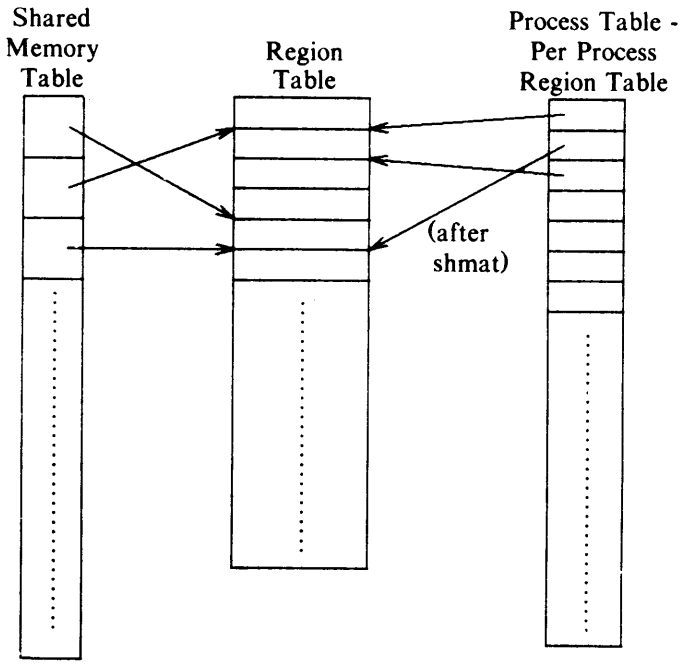

Figure 11.9. Data Structures for Shared Memory

A process attaches a shared memory region to its virtual address space with the *shmat* system call:

```
virtaddr = shmat(id, addr, flags);
```

Id, returned by a previous *shmget* system call, identifies the shared memory region, *addr* is the virtual address where the user wants to attach the shared memory, and *flags* specify whether the region is read-only and whether the kernel should round off the user-specified address. The return value, *virtaddr*, is the virtual address where the kernel attached the region, not necessarily the value requested by the process.

When executing the *shmat* system call, the kernel verifies that the process has the necessary permissions to access the region (Figure 11.10). It examines the address the user specifies: If 0, the kernel chooses a convenient virtual address.

```
algorithm shmat              /* attach shared memory */
input:  (1) shared memory descriptor
        (2) virtual address to attach memory
        (3) flags
output: virtual address where memory was attached
{
      check validity of descriptor, permissions;
      if (user specified virtual address)
      {
            round off virtual address, as specified by flags;
            check legality of virtual address, size of region;
      }
      else      /* user wants kernel to find good address */
            kernel picks virtual address: error if none available;
      attach region to process address space (algorithm attachreg);
      if (region being attached for first time)
            allocate page tables, memory for region
                                    (algorithm growreg);
      return(virtual address where attached);
}
```

Figure 11.10. Algorithm for Attaching Shared Memory

The shared memory must not overlap other regions in the process virtual address space; hence it must be chosen judiciously so that other regions do not grow into the shared memory. For instance, a process can increase the size of its data region with the *brk* system call, and the new data region is virtually contiguous with the previous data region; therefore, the kernel should not attach a shared memory region close to the data region. Similarly, it should not place shared memory close to the top of the stack so that the stack will not grow into it. For example, if the stack grows towards higher addresses, the best place for shared memory is immediately before the start of the stack region.

The kernel checks that the shared memory region fits into the process address space and attaches the region, using algorithm *attachreg*. If the calling process is the first to attach the region, the kernel allocates the necessary tables, using algorithm *growreg*, adjusts the shared memory table entry field for "last time attached," and returns the virtual address at which it attached the region.

A process detaches a shared memory region from its virtual address space by

```
shmdt(addr)
```

where *addr* is the virtual address returned by a prior *shmat* call. Although it would seem more logical to pass an identifier, the virtual address of the shared memory is used so that a process can distinguish between several instances of a shared memory region that are attached to its address space, and because the

identifier may have been removed. The kernel searches for the process region attached at the indicated virtual address and detaches it from the process address space, using algorithm *detachreg* (Section 6.5.7). Because the region tables have no back pointers to the shared memory table, the kernel searches the shared memory table for the entry that points to the region and adjusts the field for the time the region was last detached.

Consider the program in Figure 11.11: A process creates a 128K-byte shared memory region and attaches it twice to its address space at different virtual addresses. It writes data in the "first" shared memory and reads it from the "second" shared memory. Figure 11.12 shows another process attaching the same region (it gets only 64K bytes, to show that each process can attach different amounts of a shared memory region); it waits until the first process writes a nonzero value in the first word of the shared memory region and then reads the shared memory. The first process *pause*s to give the second process a chance to execute; when the first process catches a signal, it removes the shared memory region.

A process uses the *shmctl* system call to query status and set parameters for the shared memory region:

```
shmctl(id, cmd, shmstatbuf);
```

Id identifies the shared memory table entry, *cmd* specifies the type of operation, and *shmstatbuf* is the address of a user-level data structure that contains the status information of the shared memory table entry when querying or setting its status. The kernel treats the commands for querying status and changing owner and permissions similar to the implementation for messages. When removing a shared memory region, the kernel frees the entry and looks at the region table entry: If no process has the region attached to its virtual address space, it frees the region table entry and all its resources, using algorithm *freereg* (Section 6.5.6). If the region is still attached to some processes (its reference count is greater than 0), the kernel just clears the flag that indicates the region should not be freed when the last process detaches the region. Processes that are using the shared memory may continue doing so, but no new processes can attach it. When all processes detach the region, the kernel frees the region. This is analogous to the case in the file system where a process can *open* a file and continue to access it after it is *unlink*ed.

11.2.3 Semaphores

The semaphore system calls allow processes to synchronize execution by doing a set of operations atomically on a set of semaphores. Before the implementation of semaphores, a process would create a lock file with the *creat* system call if it wanted to lock a resource: The *creat* fails if the file already exists, and the process would assume that another process had the resource locked. The major disadvantages of this approach are that the process does not know when to try again, and lock files may inadvertently be left behind when the system crashes or is

```
#include      <sys/types.h>
#include      <sys/ipc.h>
#include      <sys/shm.h>
#define SHMKEY      75
#define      K      1024
int shmid;

main()
{
      int i, *pint;
      char *addr1, *addr2;
      extern char *shmat();
      extern cleanup();

      for (i = 0;  i < 20;  i++)
            signal(i, cleanup);
      shmid = shmget(SHMKEY, 128 * K, 0777 | IPC_CREAT);
      addr1 = shmat(shmid, 0, 0);
      addr2 = shmat(shmid, 0, 0);
      printf("addr1 0x%x  addr2 0x%x\n", addr1, addr2);
      pint = (int *) addr1;

      for (i = 0;  i < 256;  i++)
            *pint++ = i;
      pint = (int *) addr1;
      *pint = 256;

      pint = (int *) addr2;
      for (i = 0;  i < 256;  i++)
            printf("index %d\tvalue %d\n", i, *pint++);

      pause();
}

cleanup()
{
      shmctl(shmid, IPC_RMID, 0);
      exit();
}
```

Figure 11.11. Attaching Shared Memory Twice to a Process

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

#define SHMKEY 75
#define K 1024
int shmid;

main()
{
      int i, *pint;
      char *addr;
      extern char *shmat();

      shmid = shmget(SHMKEY, 64 * K, 0777);

      addr = shmat(shmid, 0, 0);
      pint = (int *) addr;

      while (*pint == 0)
            ;
      for (i = 0;  i < 256;  i++)
            printf("%d\n", *pint++);
}
```

Figure 11.12. Sharing Memory Between Processes

rebooted.

Dijkstra published the Dekker algorithm that describes an implementation of *semaphores*, integer-valued objects that have two atomic operations defined for them: *P* and *V* (see [Dijkstra 68]). The *P* operation decrements the value of a semaphore if its value is greater than 0, and the *V* operation increments its value. Because the operations are atomic, at most one *P* or *V* operation can succeed on a semaphore at any time. The semaphore system calls in System V are a generalization of Dijkstra's *P* and *V* operations, in that several operations can be done simultaneously and the increment and decrement operations can be by values greater than 1. The kernel does all the operations atomically; no other processes adjust the semaphore values until all operations are done. If the kernel cannot do *all* the operations, it does not do *any*; the process sleeps until it can do all the operations, as will be explained.

A semaphore in UNIX System V consists of the following elements:

- The value of the semaphore,
- The process ID of the last process to manipulate the semaphore,

- The number of processes waiting for the semaphore value to increase,
- The number of processes waiting for the semaphore value to equal 0.

The semaphore system calls are *semget* to create and gain access to a set of semaphores, *semctl* to do various control operations on the set, and *semop* to manipulate the values of semaphores.

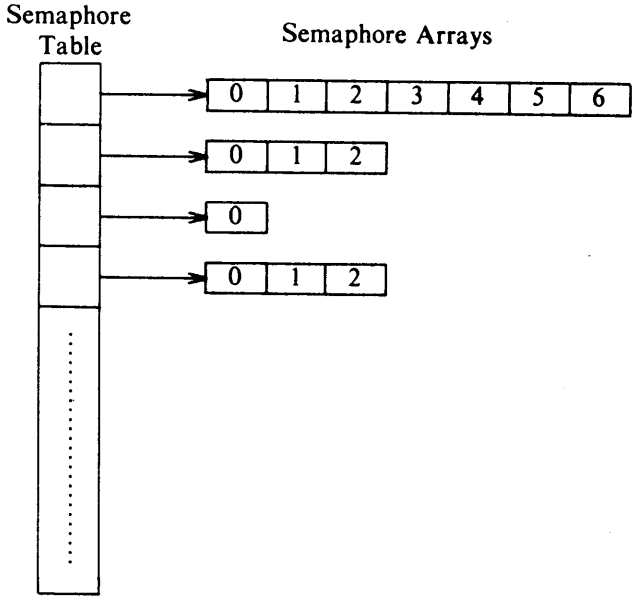

Figure 11.13. Data Structures for Semaphores

The *semget* system call creates an array of semaphores:

```
id = semget(key, count, flag);
```

where *key*, *flag* and *id* are similar to those parameters for messages and shared memory. The kernel allocates an entry that points to an array of semaphore structures with *count* elements (Figure 11.13). The entry also specifies the number of semaphores in the array, the time of the last *semop* call, and the time of the last *semctl* call. For example, the *semget* system call in Figure 11.14 creates a semaphore with two elements.

Processes manipulate semaphores with the *semop* system call:

```
oldval = semop(id, oplist, count);
```

Id is the descriptor returned by *semget*, *oplist* is a pointer to an array of semaphore operations, and *count* is the size of the array. The return value, *oldval*, is the value

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

#define SEMKEY      75
int semid;
unsigned int count;
/* definition of sembuf in file sys/sem.h
 * struct sembuf {
 *      unsigned shortsem_num;
 *      short sem_op;
 *      short sem_flg;
}; */
struct sembuf psembuf, vsembuf;              /* ops for P and V */

main(argc, argv)
      int argc;
      char *argv[];
{
      int i, first, second;
      short initarray[2], outarray[2];
      extern cleanup();

      if (argc == 1)
      {
            for (i = 0;  i < 20;  i++)
                  signal(i, cleanup);
            semid = semget(SEMKEY, 2, 0777 | IPC_CREAT);
            initarray[0] = initarray[1] = 1;
            semctl(semid, 2, SETALL, initarray);
            semctl(semid, 2, GETALL, outarray);
            printf("sem init vals %d %d\n", outarray[0], outarray[1]);
            pause();      /* sleep until awakened by a signal */
      }

      /* continued next page */
```

Figure 11.14. Locking and Unlocking Operations

of the last semaphore operated on in the set before the operation was done. The format of each element of *oplist* is

- the semaphore number identifying the semaphore array entry being operated on,
- the operation,
- flags.

```
        else if (argv[1][0] == 'a')
        {
                first = 0;
                second = 1;
        }
        else
        {
                first = 1;
                second = 0;
        }

        semid = semget(SEMKEY, 2, 0777);
        psembuf.sem_op = -1;
        psembuf.sem_flg = SEM_UNDO;
        vsembuf.sem_op = 1;
        vsembuf.sem_flg = SEM_UNDO;

        for (count = 0;  ;  count++)
        {
                psembuf.sem_num = first;
                semop(semid, &psembuf, 1);
                psembuf.sem_num = second;
                semop(semid, &psembuf, 1);
                printf("proc %d count %d\n", getpid(), count);
                vsembuf.sem_num = second;
                semop(semid, &vsembuf, 1);
                vsembuf.sem_num = first;
                semop(semid, &vsembuf, 1);
        }
}

cleanup()
{
        semctl(semid, 2, IPC_RMID, 0);
        exit();
}
```

Figure 11.14. Locking and Unlocking Operations (continued)

The kernel reads the array of semaphore operations, *oplist*, from the user address space and verifies that the semaphore numbers are legal and that the process has the necessary permissions to read or change the semaphores (Figure 11.15). If permission is not allowed, the system call fails. If the kernel must sleep as it does the list of operations, it restores the semaphores it has already operated on to their values at the start of the system call; it sleeps until the event for which

```
algorithm semop              /* semaphore operations */
inputs:  (1) semaphore descriptor
         (2) array of semaphore operations
         (3) number of elements in array
output: start value of last semaphore operated on
{
      check legality of semaphore descriptor;
 start: read array of semaphore operations from user to kernel space;
      check permissions for all semaphore operations;

      for (each semaphore operation in array)
      {
            if (semaphore operation is positive)
            {
                  add "operation" to semaphore value;
                  if (UNDO flag set on semaphore operation)
                        update process undo structure;
                  wakeup all processes sleeping (event semaphore value increases);
            }
            else if (semaphore operation is negative )
            {
                  if ("operation" + semaphore value >= 0)
                  {
                        add "operation" to semaphore value;
                        if (UNDO flag set)
                              update process undo structure;
                        if (semaphore value 0)
                              /* continued next page */
```

Figure 11.15. Algorithm for Semaphore Operation

it is waiting occurs and then restarts the system call. Because the kernel saves the semaphore operations in a global array, it reads the array from user space again if it must restart the system call. Thus, operations are done atomically — either all at once or not at all.

The kernel changes the value of a semaphore according to the value of the operation. If positive, it increments the value of the semaphore and awakens all processes that are waiting for the value of the semaphore to increase. If the semaphore operation is 0, the kernel checks the semaphore value: If 0, it continues with the other operations in the array; otherwise, it increments the number of processes asleep, waiting for the semaphore value to be 0, and goes to sleep. If the semaphore operation is negative and its absolute value is less than or equal to the value of the semaphore, the kernel adds the operation value (a negative number) to the semaphore value. If the result is 0, the kernel awakens all processes asleep, waiting for the semaphore value to be 0. If the value of the semaphore is less than

```
                        wakeup all processes sleeping (event
                            semaphore value becomes 0);
                    continue;
                }
                reverse all semaphore operations already done
                    this system call (previous iterations);
                if (flags specify not to sleep)
                    return with error;
                sleep (event semaphore value increases);
                goto start;     /* start loop from beginning */
            }
            else    /* semaphore operation is zero */
            {
                if (semaphore value non 0)
                {
                    reverse all semaphore operations done
                        this system call;
                    if (flags specify not to sleep)
                        return with error;
                    sleep (event semaphore value == 0);
                    goto start;     /* restart loop */
                }
            }
        } /* for loop ends here */
        /* semaphore operations all succeeded */
        update time stamps, process ID's;
        return value of last semaphore operated on before call succeeded;
}
```

Figure 11.15. Algorithm for Semaphore Operation (continued)

the absolute value of the semaphore operation, the kernel puts the process to sleep on the event that the value of the semaphore increases. Whenever a process sleeps in the middle of a semaphore operation, it sleeps at an interruptible priority; hence, it wakes up on receipt of a signal.

Consider the program in Figure 11.14, and suppose a user executes it (*a.out*) three times in the following sequence:

```
a.out &
a.out a &
a.out b &
```

When run without any parameters, the process creates a semaphore set with two elements and initializes their values to 1. Then, it *pauses* and sleeps until awakened by a signal, when it removes the semaphore in *cleanup*. When executing the program with parameter 'a', the process (A) does four separate semaphore

operations in the loop: It decrements the value of semaphore 0, decrements the value of semaphore 1, executes the print statement, and then increments the values of semaphores 1 and 0. A process goes to sleep if it attempts to decrement the value of a semaphore that is 0, and hence the semaphore is considered *lock*ed. Because the semaphores were initialized to 1 and no other processes are using the semaphores, process A will never sleep, and the semaphore values will oscillate between 1 and 0. When executing the program with parameter 'b', the process (B) decrements semaphores 0 and 1 in the opposite order from process A. When processes A and B run simultaneously, a situation could arise whereby process A has locked semaphore 0 and wants to lock semaphore 1, but process B has locked semaphore 1 and wants to lock semaphore 0. Both processes sleep, unable to continue. They are deadlocked and exit only on receipt of a signal.

To avoid such problems, processes can do multiple semaphore operations simultaneously. Using the following structures and code in the last example would give the desired effect.

```
struct sembuf psembuf[2];

psembuf[0].sem_num = 0;
psembuf[1].sem_num = 1;
psembuf[0].sem_op = -1;
psembuf[1].sem_op = -1;
semop(semid, psembuf, 2);
```

Psembuf is an array of semaphore operations that decrements semaphores 0 and 1 simultaneously. If either operation cannot succeed, the process sleeps until they both succeed. For instance, if the value of semaphore 0 is 1 and the value of semaphore 1 is 0, the kernel would leave the values intact until it can decrement both values.

A process can set the *IPC_NOWAIT* flag in the *semop* system call; if the kernel arrives at a situation where the process would sleep because it must wait for the semaphore value to exceed a particular value or for it to have value 0, the kernel returns from the system call with an error condition. Thus, it is possible to implement a conditional semaphore, whereby a process does not sleep if it cannot do the atomic action.

Dangerous situations could occur if a process does a semaphore operation, presumably locking some resource, and then *exit*s without resetting the semaphore value. Such situations can occur as the result of a programmer error or because of receipt of a signal that causes sudden termination of a process. If, in Figure 11.14 again, the process receives a kill signal after decrementing the semaphore values, it has no chance to reincrement them, because kill signals cannot be caught. Hence, other processes would find the semaphore locked even though the process that had locked it no longer exists. To avoid such problems, a process can set the *SEM_UNDO* flag in the *semop* call; when it *exit*s, the kernel reverses the effect of every semaphore operation the process had done. To implement this feature, the kernel maintains a table with one entry for every process in the system. Each entry

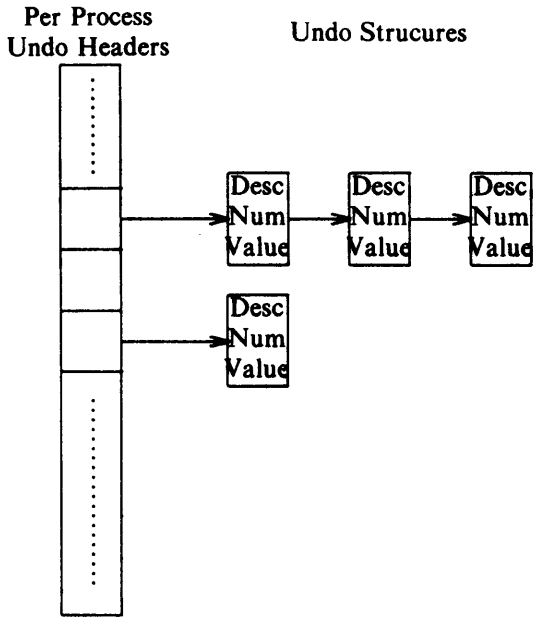

Figure 11.16. Undo Structures for Semaphores

points to a set of *undo* structures, one for each semaphore used by the process (Figure 11.16). Each undo structure is an array of triples consisting of a semaphore ID, a semaphore number in the set identified by ID, and an adjustment value.

The kernel allocates undo structures dynamically when a process executes its first *semop* system call with the *SEM_UNDO* flag set. On subsequent *semop* system calls with the *SEM_UNDO* flag set, the kernel searches the process undo structures for one with the same semaphore ID and number as the *semop* operation: If it finds one, it subtracts the value of the semaphore operation from the adjustment value. Thus, the undo structure contains a negated summation of all semaphore operations the process had done on the semaphore for which the *SEM_UNDO* flag was set. If no undo structure for the semaphore exists, the kernel creates one, sorting a list of structures by semaphore ID and number. If an adjustment value drops to 0, the kernel removes the undo structure. When a process *exit*s, the kernel calls a special routine that goes through the undo structures associated with the process and does the specified action on the indicated semaphore.

Referring back to Figure 11.14, the kernel creates an undo structure every time the process decrements the semaphore value and removes the structure every time

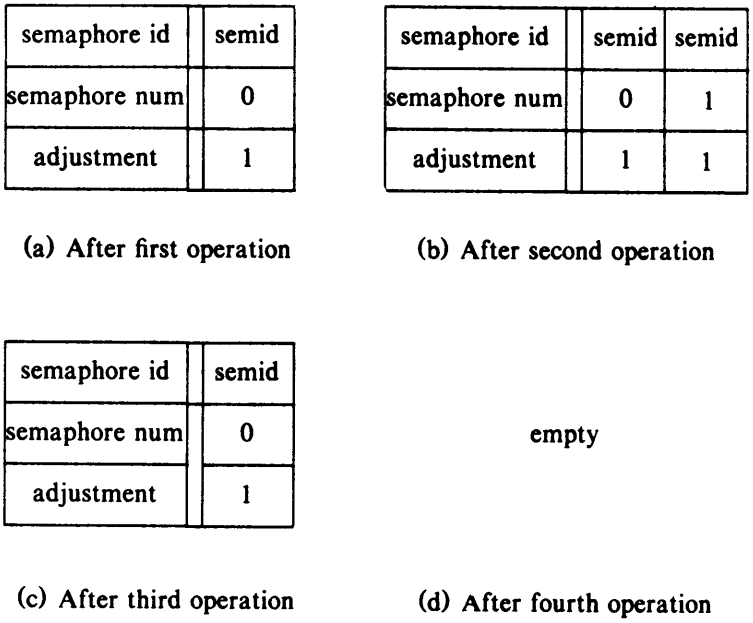

Figure 11.17. Sequence of Undo Structures

the process increments a semaphore value, because the adjustment value of the undo structure is 0. Figure 11.17 shows the sequence when invoking the program with parameter 'a'. After the first operation, the process has one triple for *semid* with semaphore number 0 and adjustment value 1, and after the second operation, it has a second triple with semaphore number 1 and adjustment value 1. If the process were to exit suddenly now, the kernel would go through the triples and add the value 1 to each semaphore, restoring their values to 0. In the regular case, the kernel decrements the adjustment value of semaphore 1 during the third operation, corresponding to the increment of the semaphore value, and it removes the triple, because its adjustment value is 0. After the fourth operation, the process has no more triples, because the adjustment values would all be 0.

The array operations on semaphores allow processes to avoid deadlock problems, as illustrated above, but they are complicated, and most applications do not need their full power. Applications that require use of multiple semaphores should deal with deadlock conditions at user level, and the kernel should not contain such complicated system calls.

The *semctl* system call contains a myriad of control operations for semaphores:

```
semctl(id, number, cmd, arg);
```

Arg is declared as a union:

```
union semunion {
        int val;
        struct semid_ds *semstat;       /* see appendix for definition */
        unsigned short *array;
} arg;
```

The kernel interprets *arg* based on the value of *cmd*, similar to the way it interprets *ioctl* commands (Chapter 10). The expected actions take place for the *cmd*s that retrieve or set control parameters (permissions and others), set one or all semaphore values in a set, or read the semaphore values. The appendix gives the details for each command. For the remove command, *IPC_RMID*, the kernel finds all processes that have undo structures for the semaphore and removes the appropriate triples. Then, it reinitializes the semaphore data structure and wakes up all processes sleeping until the occurence of some semaphore event: When the processes resume execution, they find that the semaphore ID is no longer valid and return an error to the caller.

11.2.4 General Comments

There are several similarities between the file system and the IPC mechanisms. The "get" system calls are similar to the *creat* and *open* system calls, and the "control" system calls contain an option to remove descriptors from the system, similar to the *unlink* system call. But no operations are analogous to the file system *close* system call. Thus, the kernel has no record of which processes can access an IPC mechanism, and, indeed, processes can access an IPC mechanism if they guess the correct ID and if access permissions are suitable, even though they never did a "get" call. The kernel cannot clean up unused IPC structures automatically, because it never knows when they are no longer needed. Errant processes can thus leave unneeded and unused structures cluttering the system. Although the kernel can save state information and data in the IPC structures after the death of a process, it is better to use files for such purposes.

The IPC mechanisms introduce a new name space, keys, instead of the traditional, all-pervasive files. It is difficult to extend the semantics of keys across a network, because they may describe different objects on different machines: In short, they were designed for a single-machine environment. File names are more amenable to a distributed environment as will be seen in Chapter 13. Use of keys instead of file names also means that the IPC facilities are an entity unto themselves, useful for special-purpose applications, but lacking the tool-building capabilities inherent in pipes and files, for example. Much of their functionality can be duplicated using other system facilities, so, esthetically, they should not be in the kernel. However, they provide better performance for closely cooperating application packages than standard file system facilities (see the exercises).

11.3 NETWORK COMMUNICATIONS

Programs such as mail, remote file transfer, and remote login that wish to communicate with other machines have historically used ad hoc methods to establish connections and to exchange data. For example, standard mail programs save the text of a user's mail messages in a particular file, such as "/usr/mail/mjb" for user "mjb". When a person sends mail to another user on the same machine, the *mail* program appends the mail to the addressee's file, using lock files and temporary files to preserve consistency. When a person reads mail, the *mail* program *open*s the person's mail file and *read*s the messages. To send mail to a user on another machine, the *mail* program must ultimately find the appropriate mail file on the other machine. Since it cannot manipulate files there directly, a process on the other machine must act as an agent for the local mail process; hence the local process needs a way to communicate with its remote agent across machine boundaries. The local process is called the *client* of the remote *server* process.

Because the UNIX system creates new processes via the *fork* system call, the server process must exist before the client process attempts to establish a connection. It would be inconsistent with the design of the system if the remote kernel were to create a new process when a connection request comes across the network. Instead, some process, usually *init*, creates a *server* process that *read*s a communications channel until it receives a request for service and then follows some protocol to complete the setup of the connection. Client and server programs typically choose the network media and protocols according to information in application data bases, or the data may be hard-coded into the programs.

For example, the *uucp* program allows file transfer across a network and remote execution of commands (see [Nowitz 80]). A client process queries a data base for address and routing information (such as a telephone number), *open*s an auto-dialer device, *write*s or *ioctl*s the information on the open file descriptor, and calls up the remote machine. The remote machine may have special lines dedicated for use by *uucp*; its *init* process spawns *getty* processes — the servers — to monitor the lines and wait for connection notification. After the hardware connection is established, the client process logs in, following the usual login protocol: *getty exec*s a special command interpreter, *uucico*, specified in the "/etc/passwd" file, and the client process *write*s command sequences to the remote machine, causing the remote machine to execute processes on behalf of the local machine.

Network communications have posed a problem for UNIX systems, because messages must frequently include data and control portions. The control portion may contain addressing information to specify the destination of a message. Addressing information is structured according to the type of network and protocol being used. Hence, processes need to know what type of network they are talking to, going against the principle that users do not have to be aware of a file type, because all devices look like files. Traditional methods for implementing network communications consequently rely heavily on the *ioctl* system call to specify control information, but usage is not uniform across network types. This has the unfortunate side effect that programs designed for one network may not be able to

work for other networks.

There has been considerable effort to improve network interfaces ... UNIX systems. The streams implementation in the latest releases of System V provides an elegant mechanism for network support, because protocol modules can be combined flexibly by pushing them onto open streams and their use is consistent at user level. The next section briefly describes sockets, the BSD solution to the problem.

11.4 SOCKETS

The previous section showed how processes on different machines can communicate, but the methods by which they establish communications are likely to differ, depending on protocols and media. Furthermore, the methods may not allow processes to communicate with other processes on the same machine, because they assume the existence of a server process that sleeps in a driver *open* or *read* system call. To provide common methods for interprocess communication and to allow use of sophisticated network protocols, the BSD system provides a mechanism known as *sockets* (see [Berkeley 83]). This section briefly describes some user-level aspects of sockets.

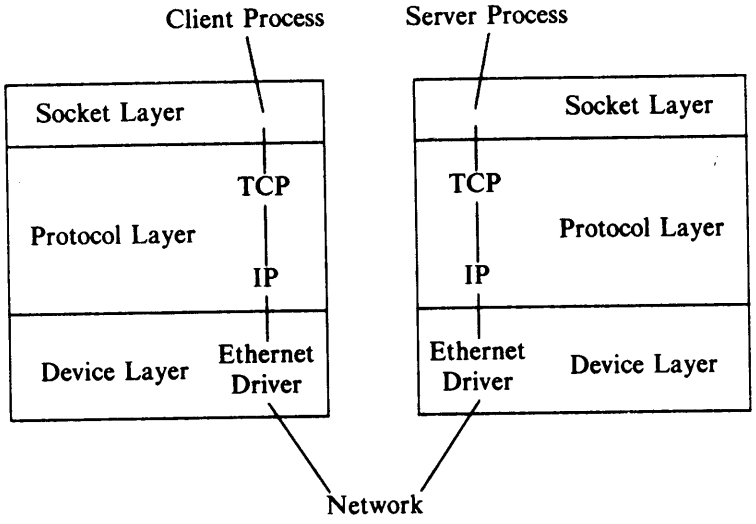

Figure 11.18. Sockets Model

The kernel structure consists of three parts: the socket layer, the protocol layer, and the device layer (Figure 11.18). The socket layer provides the interface between the system calls and the lower layers, the protocol layer contains the protocol modules used for communication (TCP and IP in the figure), and the device layer contains the device drivers that control the network devices. Legal combinations of protocols and drivers are specified when configuring the system, a method that is not as flexible as pushing streams modules. Processes communicate using the client-server model: a server process listens to a *socket*, one end point of a two-way communications path, and client processes communicate to the server process over another socket, the other end point of the communications path, which may be on another machine. The kernel maintains internal connections and routes data from client to server.

Sockets that share common communications properties, such as naming conventions and protocol address formats, are grouped into *domains*. The 4.2 BSD system supports the "UNIX system domain" for processes communicating on one machine and the "Internet domain" for processes communicating across a network using the DARPA (Defense Advanced Research Project Agency) communications protocols (see [Postel 80] and [Postel 81]). Each socket has a type — a *virtual circuit* (*stream* socket in the Berkeley terminology) or *datagram*. A virtual circuit allows sequenced, reliable delivery of data. Datagrams do not guarantee sequenced, reliable, or unduplicated delivery, but they are less expensive than virtual circuits, because they do not require expensive setup operations; hence, they are useful for some types of communication. The system contains a default protocol for every legal domain-socket type combination. For example, the Transport Connect Protocol (TCP) provides virtual circuit service and the User Datagram Protocol (UDP) provides datagram service in the Internet domain.

The socket mechanism contains several system calls. The *socket* system call establishes the end point of a communications link.

```
sd = socket(format, type, protocol);
```

Format specifies the communications domain (the UNIX system domain or the Internet domain), *type* indicates the type of communication over the socket (virtual circuit or datagram), and *protocol* indicates a particular protocol to control the communication. Processes use the socket descriptor *sd* in other system calls. The *close* system call closes sockets.

The *bind* system call associates a name with the socket descriptor:

```
bind(sd, address, length);
```

Sd is the socket descriptor, and *address* points to a structure that specifies an identifier specific to the communications domain and protocol specified in the *socket* system call. *Length* is the length of the *address* structure; without this parameter, the kernel would not know how long the address is because it can vary across domains and protocols. For example, an address in the UNIX system domain is a file name. Server processes *bind* addresses to sockets and "advertise" their names

to identify themselves to client processes.

The *connect* system call requests that the kernel make a connection to an existing socket:

```
connect(sd, address, length);
```

The semantics of the parameters are the same as for *bind*, but *address* is the address of the target socket that will form the other end of the communications line. Both sockets must use the same communications domain and protocol, and the kernel arranges that the communications links are set up correctly. If the *type* of the socket is a datagram, the *connect* call informs the kernel of the address to be used on subsequent *send* calls over the socket; no connections are made at the time of the call.

When a server process arranges to accept connections over a virtual circuit, the kernel must queue incoming requests until it can service them. The *listen* system call specifies the maximum queue length:

```
listen(sd, qlength)
```

where *sd* is the socket descriptor and *qlength* is the maximum number of outstanding requests.

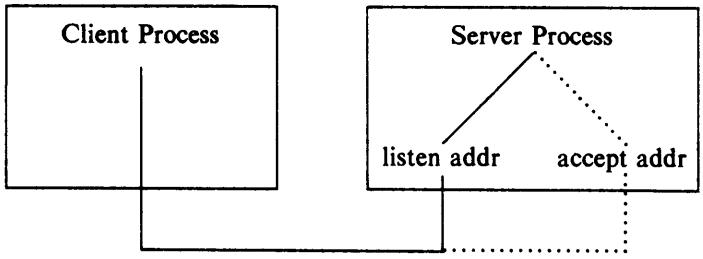

Figure 11.19. A Server Accepting a Call

The *accept* call receives incoming requests for a connection to a server process:

```
nsd = accept(sd, address, addrlen);
```

where *sd* is the socket descriptor, *address* points to a user data array that the kernel fills with the return address of the connecting client, and *addrlen* indicates the size of the user array. When *accept* returns, the kernel overwrites the contents of *addrlen* with a number that indicates the amount of space taken up by the address. *Accept* returns a new socket descriptor *nsd*, different from the socket descriptor *sd*. A server can continue listening to the advertised socket while communicating with a client process over a separate communications channel (Figure 11.19).

The *send* and *recv* system calls transmit data over a connected socket:

```
count = send(sd, msg, length, flags);
```

where *sd* is the socket descriptor, *msg* is a pointer to the data being sent, *length* is its length, and *count* is the number of bytes actually sent. The *flags* parameter may be set to the value *SOF_OOB* to send data "out-of-band," meaning that data being sent is not considered part of the regular sequence of data exchange between the communicating processes. A "remote login" program, for instance, may send an "out of band" message to simulate a user hitting the delete key at a terminal. The syntax of the *recv* system calls is

```
count = recv(sd, buf, length, flags);
```

where *buf* is the data array for incoming data, *length* is the expected length, and *count* is the number of bytes copied to the user program. *Flags* can be set to "peek" at an incoming message and examine its contents without removing it from the queue, or to receive "out of band" data. The datagram versions of these system calls, *sendto* and *recvfrom*, have additional parameters for addresses. Processes can use *read* and *write* system calls on stream sockets instead of *send* and *recv* after the connection is set up. Thus, servers can take care of network-specific protocol negotiation and spawn processes that use *read* and *write* calls only, as if they are using regular files.

The *shutdown* system call closes a socket connection:

```
shutdown(sd, mode)
```

where *mode* indicates whether the sending side, the receiving side, or both sides no longer allow data transmission. It informs the underlying protocols to close down the network communications, but the socket descriptors are still intact. The *close* system call frees the socket descriptor.

The *getsockname* system call gets the name of a socket bound by a previous *bind* call:

```
getsockname(sd, name, length);
```

The *getsockopt* and *setsockopt* calls retrieve and set various options associated with the socket, according to the communications domain and protocol of the socket.

Consider the server program in Figure 11.20. The process creates a stream socket in the "UNIX system domain" and *bind*s the name *sockname* to it. Then it invokes the *listen* system call to specify the internal queue length for incoming messages and enters a loop, waiting for incoming requests. The *accept* call sleeps until the underlying protocol notices that a connection request is directed toward the socket with the bound name; then, *accept* returns a new descriptor for the incoming request. The server process *fork*s a process to communicate with the client process: parent and child processes *close* their respective descriptors so that they do not interfere with communications traffic of the other process. The child process carries on its conversation with the client process, terminating, in this

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

main()
{
        int sd, ns;
        char buf[256];
        struct sockaddr sockaddr;
        int fromlen;

        sd = socket(AF_UNIX, SOCK_STREAM, 0);

        /* bind name - don't include null char in the name */
        bind(sd, "sockname", sizeof("sockname") - 1);
        listen(sd, 1);

        for (;;)
        {

                ns = accept(sd, &sockaddr, &fromlen);
                if (fork() == 0)
                {
                        /* child */
                        close(sd);
                        read(ns, buf, sizeof(buf));
                        printf("server read '%s'\n", buf);
                        exit();
                }
                close(ns);
        }
}
```

Figure 11.20. A Server Process in the UNIX System Domain

example, after return from the *read* system call. The server process loops and waits for another connection request in the *accept* call.

Figure 11.21 shows the client process that corresponds to the server process. The client creates a socket in the same domain as the server and issues a *connect* request for the name *sockname*, bound to some socket by a server process. When the *connect* returns, the client process has a virtual circuit to a server process. In this example, it *write*s a single message and *exit*s.

If the server process were to serve processes on a network, its system calls may specify that the socket is in the "Internet domain" by

```
socket(AF_INET, SOCK_STREAM, 0);
```

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

main()
{
        int sd, ns;
        char buf[256];
        struct sockaddr sockaddr;
        int fromlen;

        sd = socket(AF_UNIX, SOCK_STREAM, 0);

        /* connect to name - null char is not part of name */
        if (connect(sd, "sockname", sizeof("sockname") - 1) == -1)
                exit();

        write(sd, "hi guy", 6);
}
```

Figure 11.21. A Client Process in the UNIX System Domain

and *bind* a network address obtained from a name server. The BSD system has library calls that do these functions. Similarly, the second parameter to the client's *connect* would contain the addressing information needed to identify the machine on the network (or routing addresses to send messages to the destination machine via intermediate machines) and additional information to identify the particular socket on the destination machine. If the server wanted to listen to network and local processes, it would use two sockets and the *select* call to determine which client is making a connection.

11.5 SUMMARY

This chapter has presented several forms of interprocess communication. It considered process tracing, where two processes cooperate to provide a useful facility for program debugging. However, process tracing via *ptrace* is expensive and primitive, because a limited amount of data can be transferred during each call, many context switches occur, communication is restricted to parent-child processes, and processes must agree to be traced before execution. UNIX System V provides an IPC package that includes messages, semaphores, and shared memory. Unfortunately, they are special purpose, do not mesh well with other operating system primitives, and are not extensible over a network. However, they are useful to many applications and afford better performance compared to other schemes.

UNIX systems support a wide variety of networks. Traditional methods for implementing protocol negotiation rely heavily on the *ioctl* system call but their usage is not uniform across network types. The BSD system has introduced the socket system calls to provide a more general framework for network communications. In the future, System V will use the streams mechanism described in Chapter 10 to handle network configurations uniformly.

11.6 EXERCISES

1. What happens if the *wait* call is omitted by *debug* (Figure 11.3)? (Hint: There are two possibilities.)
2. A debugger using *ptrace* reads one word of data from a traced process per call. What modifications should be made in the kernel to read many words with one call? What modifications would be necessary for *ptrace*?
3. Extend the *ptrace* call such that *pid* need not be the child process of the caller. Consider the security issues: Under what circumstances should a process be allowed to read the address space of another, arbitrary process? Under what circumstances should it be able to write the address space of another process?
4. Implement the set of message system calls as a user-level library, using regular files, named pipes, and locking primitives. When creating a message queue, create a control file that records status of the queue; the file should be protected with file locks or other convenient mechanisms. When sending a message of a given type, create a named pipe for all messages of that type if such a file does not already exist, and write the data (with a prepended byte count) to the named pipe. The control file should correlate the type number with the name of the named pipe. When reading messages, the control file directs the process to the correct named pipe. Compare this scheme to the implementation described in the chapter for performance, code complexity, functionality.
5. What is the program in Figure 11.22 trying to do?
6. * Write a program that attaches shared memory too close to the end of its stack, and let the stack grow into the shared memory region. When does it incur a memory fault?
7. Rewrite the program in Figure 11.14 and use the *IPC_NOWAIT* flag, so that the semaphore operations are conditional. Demonstrate how this avoids deadlocks.
8. Show how Dijkstra's *P* and *V* semaphore operations could be implemented with named pipes. How would you implement a conditional *P* operation?
9. Write programs that lock resources, using (a) named pipes, (b) the *creat* and *unlink* system calls, and (c) the message system calls. Compare their performance.
10. Write programs to compare the performance of the message system calls to *read* and *write* on named pipes.
11. Write programs to compare the data-transfer speed using shared memory and messages. The programs for shared memory should include semaphores to synchronize completion of reads and writes.

```
#include      <sys/types.h>
#include      <sys/ipc.h>
#include      <sys/msg.h>
#define ALLTYPES 0

main()
{
      struct msgform
      {
            long mtype;
            char mtext[1024];
      } msg;
      register unsigned int id;

      for (id = 0; ; id++)
            while (msgrcv(id, &msg, 1024, ALLTYPES, IPC_NOWAIT) > 0)
                  ;
}
```

Figure 11.22. An Eavesdropping Program

12

MULTIPROCESSOR SYSTEMS

The classic design of the UNIX system assumes the use of a uniprocessor architecture, consisting of one CPU, memory, and peripherals. A multiprocessor architecture contains two or more CPUs that share common memory and peripherals (Figure 12.1), potentially providing greater system throughput, because processes can run concurrently on different processors. Each CPU executes independently, but all of them execute one copy of the kernel. Processes behave exactly as they would on a uniprocessor system — the semantics of each system call remain the same — but they can migrate between processors transparently. Unfortunately, a process does not consume less CPU time. Some multiprocessor systems are called attached processor systems, because the peripherals may not be accessible to all processors. This chapter will not distinguish between attached processor systems and general multiprocessor systems, unless explicitly stated.

Allowing several processors to execute simultaneously in kernel mode on behalf of different processes causes integrity problems unless protection mechanisms are used. This chapter explains why the original design of the UNIX system cannot run unchanged on multiprocessor systems and considers two designs for running on a multiprocessor.

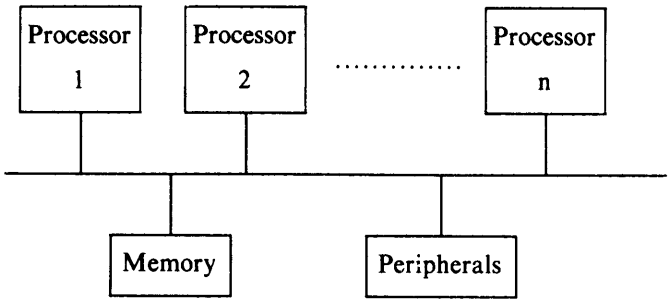

Figure 12.1. Multiprocessor Configuration

12.1 PROBLEM OF MULTIPROCESSOR SYSTEMS

Recall from Chapter 2 that the design of the UNIX system protects the integrity of kernel data structures by two policies: The kernel cannot preempt a process and switch context to another process while executing in kernel mode, and it masks out interrupts when executing a critical region of code if an interrupt handler could corrupt kernel data structures. On a multiprocessor, however, if two or more processes execute simultaneously in the kernel on separate processors, the kernel could become corrupt in spite of the protective measures that suffice for uniprocessor systems.

```
struct queue {

} *bp, *bp1;
bp1->forp = bp->forp;
bp1->backp = bp;
bp->forp = bp1;
/* consider possible context switch here */
bp1->forp->backp = bp1;
```

Figure 12.2. Placing a Buffer on a Doubly Linked List

For example, reconsider the fragment of code from Chapter 2 (Figure 12.2) that places a data structure (pointer *bp1*) after an existing structure (pointer *bp*). Suppose two processes execute the code simultaneously on different processors, such that processor A wants to place structure *bpA* after *bp* and processor B wants to place structure *bpB* after *bp*. No assumptions can be made about the relative processor execution speed: the worst case is possible, where processor B could execute the four C statements before processor A can execute another statement.

For example, handling an interrupt can delay execution of a code sequence on processor A. Corruption could occur as was illustrated in Chapter 2, even though interrupts were blocked.

The kernel must make sure that such corruption can never occur. If it were to leave a window open in which a corrupt situation could arise, no matter how rare, the kernel would be unsafe and its behavior unpredictable. There are three methods for preventing such corruption (see [Holley 79]):

1. Execute all critical activity on one processor, relying on standard uniprocessor methods for preventing corruption;
2. Serialize access to critical regions of code with locking primitives;
3. Redesign algorithms to avoid contention for data structures.

This chapter describes the first two methods to protect the kernel from corruption, and an exercise explores the third.

12.2 SOLUTION WITH MASTER AND SLAVE PROCESSORS

Goble implemented a system on a pair of modified VAX 11/780 machines where one processor, called the *master*, can execute in kernel mode and the other processor, called the *slave*, executes only in user mode (see [Goble 81]). Although Goble's implementation contained two machines, the technique extends to systems with one master and several slaves. The master processor is responsible for handling all system calls and interrupts. Slave processors execute processes in user mode and inform the master processor when a process makes a system call.

The scheduler algorithm decides which processor should execute a process (Figure 12.3). A new field in the process table designates the processor ID that a process must run on; for simplicity, assume it indicates either master or slave. When a process on a slave processor executes a system call, the slave kernel sets the processor ID field in the process table, indicating that the process should run only on the master processor, and does a context switch to schedule other processes (Figure 12.4). The master kernel schedules the process of highest priority that must run on the master processor and executes it. When it finishes the system call, it sets the processor ID field of the process to slave, allowing the process to run on slave processors again.

If processes must run on the master processor, it is preferable that the master processor run them right away and not keep them waiting. This is similar to the rationale for allowing process preemption on a uniprocessor system when returning from a system call, so that more urgent processing gets done sooner. If the master processor were executing a process in user mode when a slave processor requested service for a system call, the master process would continue executing until the next context switch according to this scheme. The master processor could respond more quickly if the slave processor set a global flag that the master processor checked in the clock interrupt handler; the master processor would do a context switch in at most one clock tick. Alternatively, the slave processor could interrupt the master

```
algorithm schedule process (modified)
input:  none
output: none
{
      while (no process picked to execute)
      {
            if (running on master processor)
                  for (every process on run queue)
                        pick highest priority process
                        that is loaded in memory;
            else      /* running on a slave processor */
                  for (every process on run queue that need not run on master)
                        pick highest priority process that is loaded in memory;
            if (no process eligible to execute)
                  idle the machine;
                  /* interrupt takes machine out of idle state */
      }
      remove chosen process from run queue;
      switch context to that of chosen process, resume its execution;
}
```

Figure 12.3. Scheduler Algorithm

```
algorithm syscall      /* revised algorithm for invocation of system call */
input:  system call number
output: result of system call
{
      if (executing on slave processor)
      {
            set processor ID field in process table entry;
            do context switch;
      }
      do regular algorithm for system call here;
      reset processor ID field to "any" (slave);
      if (other processes must run on master processor)
            do context switch;
}
```

Figure 12.4. Algorithm for System Call Handler

processor and force it to do a context switch immediately, but this assumes special hardware capability.

The clock interrupt handler on a slave processor makes sure that processes are periodically rescheduled so that no one process monopolizes the processor. Aside from that, the clock handler "wakes up" a slave processor from an idle state once a second. The slave processor schedules the highest priority process that need not run on the master processor.

The only chance for corruption of kernel data structures comes in the scheduler algorithm, because it does not protect against having a process selected for execution on two processors. For instance, if a configuration consists of a master processor and two slaves, it is possible that the two slave processors find one process in user mode ready for execution. If both processors were to schedule the process simultaneously, they would read, write and corrupt its address space.

The system can avoid this problem in two ways. First, the master can specify the slave processor on which the process should execute, permitting more than one process to be assigned to a processor. Issues of load balancing then arise: One processor may have lots of processes assigned to it, whereas others are idle. The master kernel would have to distribute the process load between the processors. Second, the kernel can allow only one processor to execute the scheduling loop at a time, using mechanisms such as *semaphores*, described in the next section.

12.3 SOLUTION WITH SEMAPHORES

Another method for supporting UNIX systems on multiprocessor configurations is to partition the kernel into critical regions such that at most one processor can execute code in a critical region at a time. Such multiprocessor systems were designed for use on the AT&T 3B20A computer and IBM 370, using semaphores to partition the kernel into critical regions (see [Bach 84]). The description here will follow those implementations. There are two issues: How to implement semaphores and where to define critical regions.

As pointed out in Chapter 2, various algorithms in uniprocessor UNIX systems use a sleep-lock to keep other processes out of a critical region in case the first process later goes to sleep inside the critical region. The mechanism for setting the lock is

```
while (lock is set)        /* test operation */
        sleep(condition until lock is free);
set lock;
```

and the mechanism for unlocking the lock is

```
free lock;
wake up all processes sleeping on condition lock set;
```

Sleep-locks delineate some critical regions, but they do not work on multiprocessor systems, as illustrated in Figure 12.5. Suppose a lock is free and two processes on

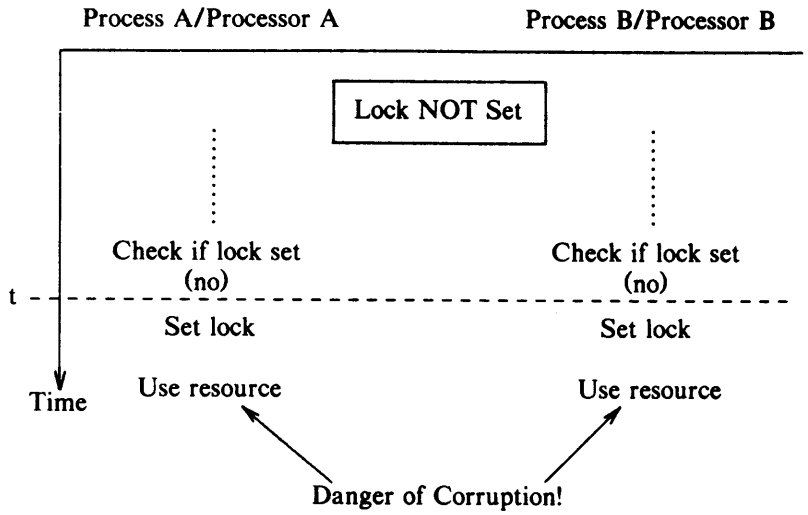

Figure 12.5. Race Conditions in Sleep-Locks on Multiprocessors

two processors simultaneously attempt to test and set it. They find that the lock is free at time t, set it, enter the critical region, and may corrupt kernel data structures. There is leeway in the requirement for simultaneity: the sleep-lock fails if neither process executes the lock operation before the other process executes the test operation. For example, if processor A handles an interrupt after finding that the lock is free and, while handling the interrupt, processor B checks the lock and sets it, processor A will return from the interrupt and set the lock. To prevent this situation, the locking primitive must be atomic: The actions of testing the status of the lock and setting the lock must be done as a single, indivisible operation, such that only one process can manipulate the lock at a time.

12.3.1 Definition of Semaphores

A semaphore is an integer-valued object manipulated by the kernel that has the following atomic operations defined for it:

- Initialization of the semaphore to a nonnegative value;
- A *P* operation that decrements the value of the semaphore. If the value of the semaphore is less than 0 after decrementing its value, the process that did the *P* goes to sleep;
- A *V* operation that increments the value of the semaphore. If the value of the semaphore becomes greater than or equal to 0 as a result, *one* process that had been sleeping as the result of a *P* operation wakes up;

- A *conditional P* operation, abbreviated *CP*, that decrements the value of the semaphore and returns an indication of true, if its value is greater than 0. If the value of the semaphore is less than or equal to 0, the value of the semaphore is unchanged and the return value is false.

The semaphores defined here are, of course, independent from the user-level semaphores described in Chapter 11.

12.3.2 Implementation of Semaphores

Dijkstra [Dijkstra 65] shows that it is possible to implement semaphores without special machine instructions. Figure 12.6 presents C functions to implement semaphores. The function *Pprim* locks the semaphore by checking the values of the array *val*; each processor in the system controls one entry in the array. When a processor locks a semaphore, it checks to see if other processors already locked the semaphore (their entry in *val* would be 2), or if processors with a lower ID are currently trying to lock it (their entry in *val* would be 1). If either condition is true, the processor resets its entry in *val* to 1 and tries again. *Pprim* starts the outer loop with the loop variable equal to the processor ID one greater than the one that most recently used the resource, insuring that no one processor can monopolize the resource (refer to [Dijkstra 65] or [Coffman 73] for a proof). The function *Vprim* frees the semaphore and allows other processors to gain exclusive access to the resource by clearing the entry of the executing processor in *val* and resetting *lastid*. The following code sequence would protect a resource.

```
Pprim(semaphore);
use resource here;
Vprim(semaphore);
```

Most machines have a set of indivisible instructions that do the equivalent locking operation more cheaply, because the loops in *Pprim* are slow and would drain performance. For instance, the IBM 370 series supports an atomic *compare and swap* instruction, and the AT&T 3B20 computer supports an atomic *read and clear* instruction. When executing the *read and clear* instruction, for example, the machine reads the value of a memory location, clears its value (sets it to 0), and sets the condition code according to whether or not the original value was zero. If another processor uses the *read and clear* instruction simultaneously on the same memory location, one processor is guaranteed to read the original value and the other process reads the value 0: The hardware insures atomicity. Thus, the function *Pprim* can be implemented more simply with the *read and clear* instruction (Figure 12.7). A process loops using the *read and clear* instruction, until it reads a nonzero value. The semaphore lock component must be initialized to 1.

This semaphore primitive cannot be used in the kernel as is, because a process executing it keeps on looping until it succeeds: If the semaphore is being used to

```
struct semaphore
{
        int val[NUMPROCS];      /* lock---1 entry for each processor */
        int lastid;             /* ID of last processor to get semaphore */
};
int procid;                     /* processor ID, unique per processor */
int lastid;                     /* ID of last proc to get the semaphore */

INIT(semaphore)
        struct semaphore semaphore;
{
        int i;
        for (i = 0;  i < NUMPROCS;  i++)
                semaphore.val[i] = 0;
}
Pprim(semaphore)
        struct semaphore semaphore;
{
        int i, first;

    loop:
        first = lastid;
        semaphore.val[procid] = 1;
        /* continued next page */
```

Figure 12.6. Implementation of Semaphore Locking in C

lock a data structure, a process should sleep if it finds the semaphore locked, so that the kernel can switch context to another process and do useful work. Given *Pprim* and *Vprim*, it is possible to construct a more sophisticated set of kernel semaphore operations, *P* and *V*, that conform to the definitions in Section 12.3.1.

First, let us define a semaphore to be a structure that consists of a lock field to control access to the semaphore, the value of the semaphore, and a queue of processes sleeping on the semaphore. The lock field controls access to the semaphore, allowing only one process to manipulate the other fields of the structure during *P* and *V* operations. It is reset when the *P* or *V* operation completes. The value field determines whether a process should have access to the critical region protected by the semaphore. At the beginning of the *P* algorithm (Figure 12.8), the kernel does a *Pprim* operation to ensure exclusive access to the semaphore and then decrements the semaphore value. If the semaphore value is nonnegative, the executing process has access to the critical region: It resets the semaphore lock with the *Vprim* operation so that other processes can access the semaphore and returns an indication of success. If, as a result of decrementing its value, the semaphore value is negative, the kernel puts the process to sleep, following

```
    forloop:
        for (i = first;  i < NUMPROCS;  i++)
        {
                if (i == procid)
                {
                        semaphore.val[i] = 2;
                        for (i = 1;  i < NUMPROCS;  i++)
                                if (i != procid && semaphore.val[i] == 2)
                                        goto loop;
                        lastid = procid;
                        return;                 /* success! now use resource */
                }
                else if (semaphore.val[i])
                        goto loop;
        }
        first = 1;
        goto forloop;
}
Vprim(semaphore)
        struct semaphore semaphore;
{
        lastid = (procid+1) % NUMPROCS;         /* reset to next processor */
        semaphore.val[procid] = 0;
}
```

Figure 12.6. Implementation of Semaphore Locking (continued)

semantics similar to those of the regular sleep algorithm (Chapter 6): It checks for signals according to the priority value, enqueues the executing process on a first-in-first-out list of sleeping processes, and does a context switch. The *V* function (Figure 12.9) gains exclusive access to the semaphore via the *Pprim* primitive and increments the semaphore value. If any processes were on the semaphore sleep queue, the kernel removes the first one and changes its state to "ready to run."

The *P* and *V* functions are similar to the sleep and wakeup functions: The major difference in implementation is that a semaphore is a data structure, whereas the address used for sleep and wakeup is just a convenient number. A process will always sleep when doing a *P* operation on a semaphore if the initial value of the semaphore is 0, so *P* can replace the sleep function. However, the *V* operation wakes up only one process, whereas the uniprocessor wakeup function wakes up all processes asleep on an event address.

Semantically, use of the wakeup function indicates that a given system condition is no longer true, hence all processes that were asleep on the condition must wake up. For example, when a buffer is no longer in use, it is incorrect for processes to sleep on the event the buffer is busy, so the kernel awakens all processes that were

```
struct semaphore {
        int lock;
};

Init(semaphore)
        struct semaphore semaphore;
{
        semaphore.lock = 1;
}

Pprim(semaphore)
        struct semaphore semaphore;
{
        while (read_and_clear(semaphore.lock))
                ;
}

Vprim(semaphore)
        struct semaphore semaphore;
{
        semaphore.lock = 1;
}
```

Figure 12.7. Semaphore Operations Using Read and Clear Instruction

asleep on the event. As a second example, if multiple processes *write* data to a terminal, the terminal driver may put them to sleep because it cannot handle the high volume of data. Later, when the driver decides it can accept more data for output, it wakes up all processes that were asleep, waiting to output data. Use of the *P* and *V* operations is more applicable for locking operations where processes gain access to a resource one by one and other processes are granted access in the order they requested the resource. This is usually more efficient than the uniprocessor sleep-lock, because if all processes wake up on occurrence of an event, most may find the lock still set and return to sleep immediately. On the other hand, it is more difficult to use *P* and *V* for cases where all processes should be awakened at once.

Given a primitive that returns the value of a semaphore, would the following operation be the equivalent of the wakeup function?

```
while (value(semaphore) < 0)
        V(semaphore);
```

Assuming no interference from other processors, the kernel executes the loop until the value of the semaphore is greater than or equal to 0, meaning that no processes are asleep on the semaphore. However, it is possible for process A on processor A