Interprocess Communication 13

Processes and Interprocess Communication

Whenever you execute a command in the SunOS 5.x system, you initiate a process that is numbered and tracked by the operating system. A flexible feature of the SunOS 5.x system is that processes can be generated by other processes. This happens often.

Processes communicate with each other for a variety of reasons. Many applications have been implemented as collections of cooperating processes. In distributed computing and in desktop integration, the entities that communicate are processes. Increasingly, interprocess communication appears in different forms and different roles in today's computing systems.

The original concept of Interprocess Communication (IPC) was limited to a single system. Numerous higher level services, generalized to work across networks, are now available for IPC. This chapter focuses on the services that are specifically intended for single-system use. The distributed services are mentioned for comparison and to aid developers in selecting the right service to use.

Uses of IPC

This section shows the different ways in which you can use IPC services.

Shell Pipelines

Pipes, the output of one process to another with a shell command, is one of the earliest uses of interprocess communication in UNIX. Applications that read from standard input and write to standard output can easily be linked together into *pipelines*. Typically, applications that behave as *filters* get used this way. The original IPC mechanism of UNIX, *pipes*, supported this feature.

Background Processing and Daemons

A more prevalent use of IPC is communicating with *daemons*, or processes that run in the background of a system environment. Daemons can be the servers in a client/server implementation of a system service. (The client/server model is discussed in the chapter on "Networking".) In other cases, applications can "spin off" background processes to carry on processing that is not tied to activity in the user interface.

Applications Structured as Cooperating Processes

You may want to use cooperating processes for several reasons:

- Multiple processes are a means of achieving concurrency in an application. (Processes are the "heavyweight" mechanism for this; threads are the lightweight alternative.)
- Some large applications can benefit from the strong modularity that separate processes provide.
- Client/server implementations usually consist of client processes and server processes.

Integrated Applications

Integrated autonomous applications, such as in an integrated desktop, require the kind of loose coupling that IPC offers to exchange data and to coordinate with each other.

SunSoft's ToolTalk service, discussed in the chapter on application interoperability, is a higher level service designed for this purpose. Because it provides network transparency, object orientation, and numerous application-oriented features, it is not included in this chapter.

An Overview of SunOS 5.x IPC Services

The SunOS 5.x system provides several mechanisms that allow processes to exchange data and synchronize execution. The simplest of these mechanisms are pipes, named pipes, and signals.

The SunOS 5.x system also provides a standard IPC package that supports three more versatile types of interprocess communication:

- *Messages* allow processes to send formatted data STREAMS to arbitrary processes.
- *Semaphores* allow processes to synchronize execution.
- *Shared memory* allows processes to share parts of their virtual address space.

Additionally, the SunOS 5.x system offers a number of advanced techniques for achieving interprocess communication:

- *Virtual Memory Management* is the native facility of SunOS 5.x that supports sharing of memory among processes.
- *File and Record Locking* are appropriate ways of synchronizing processes where the intent is to share databases.
- *Remote Procedure Calls* are an advanced method of implementing client/server and other systems of cooperating processes. They should be considered for their power and network transparency.

Basic IPC

The basic IPC services of SunOS 5.x consist of pipes, socket-based IPC, and signals. This section provides an overview of these concepts.

Pipes

The `pipe()` system call creates an interprocess channel. (The interprocess channel created by `pipe()` is not suitable for communication between systems in a network.)

The standard programming model is that after a pipe has been set up, two (or more) cooperating processes (created by subsequent `fork()` calls) will pass data through the pipe using `read()` and `write()`. Each end of the pipe is represented by a file descriptor. In SunOS 5.x, pipes are bidirectional.

Although convenient to use because of their simplicity, pipes are limited in what they can do. For example, because of the way pipes are created, processes communicating through unnamed pipes must have a common ancestor.

Because of the limitations of unnamed pipes, several enhancements of this concept have evolved.

Named Pipes

Named pipes are pipes that have directory entries. The ability to name pipes eliminates the need for processes connected by pipes to have a common ancestor. This makes it much easier to set up pipes, say, between server processes and diverse client processes.

You can set up a named pipe by using the `mknod()` system call to create a special file known as a FIFO. The term *FIFO* refers to the first-in-first-out ordering of data passed through a pipe. All pipes, whether named or not, have this property. However, the FIFO type directory entry has led to named pipes being referred to as *FIFOs*.

Socket-Based IPC

In 4.2 BSD, the file descriptors at the endpoints of pipes are referred to *sockets*, a mechanism capable of supporting interprocess communication across a network. Sockets are discussed in the chapter on “Networking”.

In SunOS 5.x the endpoints of pipes are stream files. A related function, `socketpair()`, which provided a two-way pipe in socket-based IPC, is supported in SunOS 5.x for compatibility. However, `socketpair()` is no longer needed because pipes are now bidirectional.

Developers who are interested in socket-based IPC because of its extensibility into networking should be aware that sockets have been superseded as the native facility of SunOS. In SunOS 5.x, the implementations of both pipes and network transport protocols are based on STREAMS. While sockets are still supported in SunOS 5.x, the native interface to the STREAMS-based transports is the OSI-compatible Transport Layer Interface (TLI).

Signals

The system defines a set of signals that can be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is normally blocked from further occurrence, the current process context is saved, and a new one is built. A process can specify the handler to which a signal is delivered or specify that the signal is to be blocked or ignored. A process can also specify that an action is to be taken when a signal is received.

A process can send a tiny amount of information to another process by a signal. Signals are sometimes used in this way to “poke” another process, indicating that a certain type of event has occurred.

Standard IPC

The standard IPC functions of SunOS 5.x support interprocess messages, semaphores, and shared memory. They are useful in database management and transaction processing. Like Basic IPC, these mechanisms are not applicable to processes on separate hosts.

The three services are designed as a unified package:

- Each mechanism contains a *get* function to create a new entry or retrieve an existing one.
- Each mechanism contains a *control* function to query the status of an entry, to set status information, and to remove the entry from the system.
- Each mechanism contains one or more *operations* functions to perform various operations on an entry.

Permissions

Permissions for messages, semaphores, and shared memory can be extended to users other than the one for which the facility was created. The creating process identifies the default owner. Unlike files, however, the creator can assign ownership of the facility to another user; it can also revoke an ownership assignment. The current owner process, in turn, can grant read or write access to still other users.

Permissions for an IPC facility are initialized by the creating process and can be modified by any process with permission to perform control operations on that facility.

Key Arguments and ID Numbers

Processes requesting access to a common IPC facility must have a way to determine the identity of the facility. To do this, functions that initialize or provide access to an IPC facility use a *key* argument. The key value can be used by all processes attempting to access the facility.

Functions that initialize or get access to messages, semaphores, or shared memory return an ID number. Functions that perform read, write, and control operations use this ID.

Message Queues

IPC messaging allows processes to send and receive messages, and to queue messages for processing in an arbitrary order. Unlike the file byte-stream model of data flow used for pipes, each IPC message has an explicit length. More importantly, messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized through the `msgget()` function. The owner or creator of a queue can change its ownership or permissions using `msgctl()`. Also, any process with permission to do so can use `msgctl()` for control operations.

Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively (see `msgop()`). When a message is sent, its text is copied to the message queue.

The `msgsnd()` and `msgrcv()` functions can be performed as either blocking or non-blocking operations. A blocked message operation remains suspended until one of the following three conditions occurs:

- the call succeeds
- the process receives a signal
- the queue is removed

Structure of a Message Queue

A *message queue* contains a control structure with a unique ID, a linked list of message headers, and a buffer for the message text. The identifier for the queue is the `msqid`.

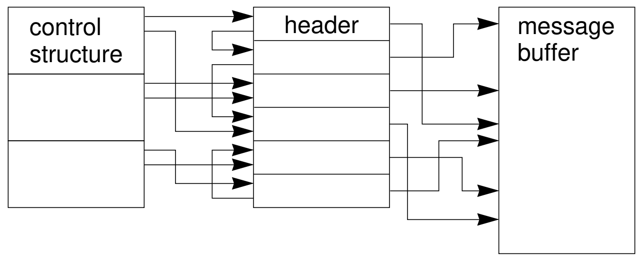

Figure 13-1 Structure of a Message Queue

The control structure for the message queue contains the following information:

- a permissions structure
- a pointer to the first message on the queue
- a pointer to the last message on the queue
- the number of bytes in the queue
- the number of messages in the queue
- the maximum number of bytes allowed in the queue
- the process ID (PID) of the last message sender
- the PID of the last message receiver
- the time the last message was sent
- the time the last message was received
- the time of the last change to the structure

Each message header contains the following information:

- a pointer to the next message on the queue
- the message type
- the message text size
- the message text address

The message queue control structure is defined in `<sys/msg.h>`.

Code Example 13-1 Message Queue Control Structure

```
struct msqid_ds
{
    struct ipc_permmsg_perm;/* operation permission struct */
    struct msg*msg_first;/* ptr to first message on q */
    struct msg*msg_last;/* ptr to last message on q */
```

Code Example 13-1 Message Queue Control Structure (Continued)

```
    ulong   msg_cbytes;/* current # bytes on q */
    ulong   msg_qnum;/* # of messages on q */
    ulong   msg_qbytes;/* max # of bytes on q */
    pid_t   msg_lspid;/* pid of last msgsnd */
    pid_t   msg_lrpid;/* pid of last msgrcv */
    time_t  msg_stime;/* last msgsnd time */
    long    msg_pad1;/* reserved for time_t expansion */
    time_t  msg_rtime;/* last msgrcv time */
    long    msg_pad2;/* time_t expansion */
    time_t  msg_ctime;/* last change time */
    long    msg_pad3;/* time expansion */
    long    msg_pad4[4];/* reserve area*/
};
```

The definition for the message-header data structure is shown in the following code example:

Code Example 13-2 Message Header Structure

```
struct msg
{
    struct msg*msg_next;/* ptr to next message on q */
    long msg_type;/* message type */
    short msg_ts;/* message text size */
    short msg_spot;/* message text map address */
};
```

Initializing a Message Queue with `msgget()`

The `msgget()` function initializes a new message queue. It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. When the call fails, it returns `-1` and sets the external variable `errno` to the appropriate error code. The `msgget()` synopsis is shown in the following code sample:

Code Example 13-3 Synopsis of msgget()

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

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

The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The MSGMNI kernel configuration option determines the maximum number of unique message queues that the kernel will support. The msgget() function fails when this limit is exceeded.

The following example is a simple exercise to illustrate the msgget() function. The program prompts for a key, an octal permissions code, and for your choice of control flags. It allows all possible combinations. When msgget succeeds, it displays the message queue ID that the call returned. When msgget() fails, the program indicates that there was an error and displays the reason for the failure.

Code Example 13-4   Program to Illustrate msgget()

```
/*
** msgget.c: Illustrate the msgget() function.
**
** This is a simple exerciser of the msgget() function.
** It prompts for the arguments, makes the call, and reports the
** results.
*/

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

extern void exit();
extern void perror();

main()
{
    key_tkey;/* key to be passed to msgget() */
    int msgflg,/* msgflg to be passed to msgget() */
        msqid;/* return value from msgget() */

    (void) fprintf(stderr,
    "All numeric input is expected to follow C conventions:\n");
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");
    (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
    (void) fprintf(stderr, "Enter key: ");
    (void) scanf("%li", &key);
    (void) fprintf(stderr, "\nExpected flags for msgflg argument are:\n");
    (void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
    (void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
    (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
    (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
```

Code Example 13-4 Program to Illustrate msgget() (Continued)

```
    (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
    (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
    (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
    (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
    (void) fprintf(stderr, "Enter msgflg value: ");
    (void) scanf("%i", &msgflg);

    (void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx, %#o)\n",
    key, msgflg);
    if ((msqid = msgget(key, msgflg)) == -1)
    {
        perror("msgget: msgget failed");
        exit(1);
    } else {
        (void) fprintf(stderr,
            "msgget: msgget succeeded: msqid = %d\n", msqid);
        exit(0);
    }
    /* NOTREACHED */
}
```

Controlling Message Queues with msgctl()

The `msgctl()` function alters the permissions and other characteristics of a message queue.

Its synopsis is as follows:

Code Example 13-5 Synopsis of msgctl()

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

int msgctl(int msqid, int cmd, .../* struct msqid_ds *buf */);
```

Upon successful completion, the call returns zero. Upon failure, it returns `-1` and sets `errno` appropriately.

The `msqid` argument must be the ID of an existing message queue. The `cmd` argument is one of the following:

- `IPC_STAT`
  Place information about the status of the queue in the data structure pointed to by `buf`. The process must have read permission for this call to succeed.

- `IPC_SET`
  Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or super-user for this call to succeed.

- `IPC_RMID`
  Remove the message queue specified by the `msqid` argument.

The following sample program illustrates the `msgctl()` function with all its various flags:

Code Example 13-6 Program to Illustrate msgctl()

```
/*
** msgctl.c:  Illustrate the msgctl() function.
**
** This is a simple exerciser of the msgctl() function.  It allows
** you to perform one control operation on one message queue.  It
** gives up immediately if any control operation fails, so be careful not
** to set permissions to preclude read permission; you won't be able to
** reset the permissions with this code if you do.
*/
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>
#include   <time.h>

static void do_msgctl();
extern void exit();
extern void perror();
static char warning_message[] = "If you remove read permission for \
yourself, this program will fail frequently!";

main()
{
    struct msqid_dsbuf;/* queue descriptor buffer for IPC_STAT
                          and IP_SET commands */
    int             cmd,/* command to be given to msgctl() */
                    msqid;/* queue ID to be given to msgctl() */

    (void fprintf(stderr,
        "All numeric input is expected to follow C conventions:\n");
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");

    /* Get the msqid and cmd arguments for the msgctl() call. */
```

Code Example 13-6 Program to Illustrate msgctl() (Continued)

```
    (void) fprintf(stderr,
        "Please enter arguments for msgctls() as requested.");
    (void) fprintf(stderr, "\nEnter the msqid: ");
    (void) scanf("%i", &msqid);
    (void) fprintf(stderr, "Valid msgctl commands are:\n");
    (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);

    (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
    (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
    (void) fprintf(stderr, "\nEnter the value for the command: ");
    (void) scanf("%i", &cmd);

    switch (cmd) {

    case IPC_SET:
    /* Modify settings in the message queue control structure. */
    (void) fprintf(stderr, "Before IPC_SET, get current values:");
    /* fall through to IPC_STAT processing */
    case IPC_STAT:
    /*
    ** Get a copy of the current message queue control structure
    ** and show it to the user.
    */
    do_msgctl(msqid, IPC_STAT, &buf);
    (void) fprintf(stderr,
    "msg_perm.uid = %d\n", buf.msg_perm.uid);
    (void) fprintf(stderr,
    "msg_perm.gid = %d\n", buf.msg_perm.gid);
    (void) fprintf(stderr,
    "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
    (void) fprintf(stderr,
    "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
    (void) fprintf(stderr, "msg_perm.mode = %#o, ",
    buf.msg_perm.mode);
    (void) fprintf(stderr, "access permissions = %#o\n",
    buf.msg_perm.mode & 0777);
    (void) fprintf(stderr, "msg_cbytes = %d\n", buf.msg_cbytes);
    (void) fprintf(stderr, "msg_qbytes = %d\n", buf.msg_qbytes);
    (void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
    (void) fprintf(stderr, "msg_lspid = %d\n", buf.msg_lspid);
    (void) fprintf(stderr, "msg_lrpid = %d\n", buf.msg_lrpid);
    (void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ?
    ctime(&buf.msg_stime) : "Not Set\n");
    (void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ?
    ctime(&buf.msg_rtime) : "Not Set\n");
```

Code Example 13-6 Program to Illustrate msgctl() (Continued)

```
    (void) fprintf(stderr, "msg_ctime = %s", ctime(&buf.msg_ctime));
    if (cmd == IPC_STAT)
    break;
    /*

    **  Now continue with IPC_SET.
    */
    (void) fprintf(stderr, "Enter msg_perm.uid: ");
    (void) scanf ("%hi", &buf.msg_perm.uid);
    (void) fprintf(stderr, "Enter msg_perm.gid: ");
    (void) scanf("%hi", &buf.msg_perm.gid);
    (void) fprintf(stderr, "%s\n", warning_message);
    (void) fprintf(stderr, "Enter msg_perm.mode: ");
    (void) scanf("%hi", &buf.msg_perm.mode);
    (void) fprintf(stderr, "Enter msg_qbytes: ");
    (void) scanf("%hi", &buf.msg_qbytes);
    do_msgctl(msqid, IPC_SET, &buf);
    break;

    case IPC_RMID:
    default:
    /* Remove the message queue or try an unknown command. */
    do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
    break;
    }
    exit(0);
    /* NOTREACHED */
}
/*
** Print indication of arguments being passed to msgctl(), call msgctl(),
** and report the results.
** If msgctl() fails, do not return; this example doesn't deal with
** errors, it just reports them.
*/
static void
do_msgctl(msqid, cmd, buf)
struct msqid_ds*buf;/* pointer to queue descriptor buffer */
int cmd,/* command code */
    msqid;/* queue ID */
{
    register intrtrn;/* hold area for return value from msgctl() */

    (void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d, %s)\n",
    msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
    rtrn = msgctl(msqid, cmd, buf);
```

Code Example 13-6 Program to Illustrate msgctl() (Continued)

```
    if (rtrn == -1) {
    perror("msgctl: msgctl failed");

    exit(1);
    /* NOTREACHED */
    } else {
    (void) fprintf(stderr, "msgctl: msgctl returned %d\n", rtrn);
    }
}
```

Using msgsnd() and msgrcv() to Send and Receive Messages

The msgsnd() and msgrcv() functions send and receive messages, respectively. Their synopses are as follows:

Code Example 13-7 Synopses of msgsnd() and msgrcv()

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

int msgsnd(int msqid, const void *msgp,
    size_t msgsz, int msgflg);

int msgrcv(int msqid, void *msgp,
    size_t msgsz, long msgtyp, int msgflg);
```

Upon successful completion, each of these functions returns zero. When unsuccessful, each call returns -1 and sets the external variable errno to the appropriate error code.

The msqid argument must be the ID of an existing message queue. The msgp argument is a pointer to a structure that contains the type of the message and its text. The msgsz argument specifies the length of the message in bytes.

Various control flags can be passed in the msgflg argument. Combine flags within the argument using the logical OR operator. When IPC_NOWAIT is set, a send or receive operation that cannot complete fails. For instance, a non-blocking msgrcv() operation fails when there is no message to receive. If MSG_NOERROR is set, then a message longer than the size specified by msgsz is truncated to that size. Note that the trailing portion of the truncated message is lost. Without the MSG_NOERROR flag, attempting to receive a message that is longer than expected results in failure.

The msgtyp argument to msgrcv() indicates the type of message to receive. When msgtyp() equals zero, the call receives the first message on the queue. When it is greater than zero, the call receives the first message of the indicated type.

When msgtyp is less than zero, the call receives the first message on the queue with the lowest type value, up to and including the absolute value of the argument. For instance, when msgtyp has a value of −3, the call retrieves the first message of type 1, if any, or the first message of type 2, if any, or the first message of type 3. It does not receive a message of type 4. This allows you to prioritize message processing according to type.

The following sample program illustrates msgsnd() and msgrcv():

Code Example 13-8 Using msgsnd() and msgrcv()

```
/*
** msgop.c: Illustrate the msgsnd() and msgrcv() functions.
**
** This is a simple exerciser of the message send and receive
** routines. It allows the user to attempt to send and receive as many
** messages as wanted to or from one message queue.
*/

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

static intask();
extern voidexit();
extern char*malloc();
extern voidperror();

charfirst_on_queue[] = "-> first message on queue",
    full_buf[] = "Message buffer overflow. Extra message text discarded.";

main()
{
    register int c;/* message text input */
    int choice;/* user's selected operation code */
    register inti;/* loop control for mtext */
    int msgflg;/* message flags for the operation */
    struct msgbuf *msgp;/* pointer to the message buffer */
    int msgsz;/* message size */
    longmsgtyp;/* desired message type */
    int msqid,/* message queue ID to be used */
        maxmsgsz,/* size of allocated message buffer */
        rtrn;/* return value from msgrcv or msgsnd */

    (void) fprintf(stderr,
        "All numeric input is expected to follow C conventions:\n");
```

Code Example 13-8 Using msgsnd() and msgrcv() (Continued)

```
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");

    /* Get the message queue ID and set up the message buffer. */
    (void) fprintf(stderr, "Enter msqid: ");
    (void) scanf("%i", &msqid);
    /*
    ** Note that <sys/msg.h> includes a definition of struct msgbuf
    ** with the mtext field defined as:
    **      charmtext[1];
    ** therefore, this definition is only a template, not a directly
    ** useable structure definition, unless you want only to send
    ** and receive messages of 0 or 1 byte.
    **  To handle this, malloc an area big enough to contain the
    ** template - the size of the mtext template field + the size of
    ** the mtext field wanted. Then you can use the pointer returned
    ** by malloc as a struct msgbuf with an mtext field of the size
    ** you want.
    **   Note also that sizeof msgp->mtext is valid even though msgp
    ** isn't pointing to anything yet. Sizeof doesn't dereference msgp,
    ** but uses its type to figure out what you are asking about.
    */
    (void) fprintf(stderr, "Enter the message buffer size you want: ");
    (void) scanf("%i", &maxmsgsz);
    if (maxmsgsz < 0) {
        (void) fprintf(stderr, "msgop: %s\n",
            "The message buffer size must be >= 0.");
        exit(1);
        /* NOTREACHED */
    }
    msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf) -
        sizeof msgp->mtext + maxmsgsz));
    if (msgp == NULL) {
        (void) fprintf(stderr, "msgop: %s %d byte messages.\n",
            "could not allocate message buffer for", maxmsgsz);
        exit(1);
        /* NOTREACHED */
    }
    /* Loop through message operations until the user is ready to quit. */
    while (choice = ask()) {
        switch (choice) {
        case 1: /* msgsnd() requested: Get the arguments, make the
                call, and report the results. */
            (void) fprintf(stderr, "Valid msgsnd message %s\n",
```

Code Example 13-8 Using msgsnd() and msgrcv() (Continued)

```
                "types are positive integers.");
            (void) fprintf(stderr, "Enter msgp->mtype: ");
            (void) scanf("%li", &msgp->mtype);


            if (maxmsgsz) {
                /* Since you've been using scanf, you need the
                    following loop to throw away the rest of
                    the input on the line after the entered
                    mtype before you start reading the mtext. */
                while ((c = getchar()) != '\n' && c != EOF)
                    ;
                (void) fprintf(stderr, "Enter a %s:\n",
                    "one line message");
                for (i = 0; ((c = getchar()) != '\n'); i++) {
                    if (i >= maxmsgsz) {
                        (void) fprintf(stderr,
                            "\n%s\n", full_buf);
                        while ((c = getchar()) != '\n')
                            ;
                        break;
                    }
                    msgp->mtext[i] = c;
                }
                msgsz = i;
            } else
                msgsz = 0;

            (void) fprintf(stderr,
                "\nMeaningful msgsnd flag is:\n");
            (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
                IPC_NOWAIT);
            (void) fprintf(stderr, "Enter msgflg: ");
            (void) scanf("%i", &msgflg);

            (void) fprintf(stderr, "%s(%d, msgp, %d, %#o)\n",
                "msgop: Calling msgsnd", msqid, msgsz, msgflg);
            (void) fprintf(stderr, "msgp->mtype = %ld\n",
                msgp->mtype);
            (void) fprintf(stderr, "msgp->mtext = \"");
            for (i = 0; i < msgsz; i++)
                (void) fputc(msgp->mtext[i], stderr);
            (void) fprintf(stderr, "\"\n");

                rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
```

Code Example 13-8 Using msgsnd() and msgrcv() (Continued)

```
            if (rtrn == -1)
                perror("msgop: msgsnd failed");
            else

                    (void) fprintf(stderr,
                            "msgop: msgsnd returned %d\n", rtrn);
            break;
    case 2: /* msgrcv() requested: Get the arguments, make the
            call, and report the results. */
        for (msgsz = -1; msgsz < 0 || msgsz > maxmsgsz;
            (void) scanf("%i", &msgsz))
            (void) fprintf(stderr,
                "%s (0 <= msgsz <= %d): ",
                "Enter msgsz", maxmsgsz);

        (void) fprintf(stderr, "msgtyp meanings:\n");
        (void) fprintf(stderr, "\t 0 %s\n", first_on_queue);
        (void) fprintf(stderr, "\t>0 %s of given type\n",
            first_on_queue);
        (void) fprintf(stderr,
            "\t<0 %s with type <= |msgtyp|\n",
            first_on_queue);
        (void) fprintf(stderr, "Enter msgtyp: ");
        (void) scanf("%li", &msgtyp);

        (void) fprintf(stderr,
            "Meaningful msgrcv flags are:\n");
        (void) fprintf(stderr, "\tMSG_NOERROR =\t%#8.8o\n",
            MSG_NOERROR);
        (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
            IPC_NOWAIT);
        (void) fprintf(stderr, "Enter msgflg: ");
        (void) scanf("%i", &msgflg);

        (void) fprintf(stderr, "%s(%d, msgp, %d, %ld, %#o);\n",
            "msgop: Calling msgrcv",
            msqid, msgsz, msgtyp, msgflg);

        rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);

        if (rtrn == -1)
            perror("msgop: msgrcv failed");
        else {
            (void) fprintf(stderr, "msgop: %s %d\n",
                "msgrcv returned", rtrn);
```

Code Example 13-8 Using msgsnd() and msgrcv() (Continued)

```
                (void) fprintf(stderr, "msgp->mtype = %ld\n",
                    msgp->mtype);
                (void) fprintf(stderr, "msgp->mtext is: \"");
                for (i = 0; i < rtrn; i++)
                    (void) fputc(msgp->mtext[i], stderr);
                (void) fprintf(stderr, "\"\n");
            }
            break;

        default:
            (void) fprintf(stderr, "msgop: operation unknown\n");
            break;
        }
    }
    exit(0);
    /* NOTREACHED */
}
/*
**  Ask the user what to do next. Return the user's choice code.
**  Don't return until the user selects a valid choice.
*/
static
ask()
{
    int response;/* User's response. */

    do {
        (void) fprintf(stderr, "Your options are:\n");
        (void) fprintf(stderr, "\tExit =\t0 or Control-D\n");
        (void) fprintf(stderr, "\tmsgsnd =\t1\n");
        (void) fprintf(stderr, "\tmsgrcv =\t2\n");
        (void) fprintf(stderr, "Enter your choice: ");

        /* Preset response so "^D" will be interpreted as exit. */
        response = 0;
        (void) scanf("%i", &response);
    } while (response < 0 || response > 2);

    return(response);
}
```

See the `msgop()` man page for more details on these functions.

Semaphores

Semaphores provide a way for processes to query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments. Semaphores can be operated on as individual units or as elements in a set.

A semaphore set consists of a control structure and an array of individual semaphores. By default, a set of semaphores can contain up to 25 elements. Your system administrator can alter this limit through the `SEMMSL` system configuration option.

Before a process can use a semaphore, the semaphore set must be initialized using `semget()`. The semaphore owner or creator can change its ownership or permissions using `semctl()`. Also, any process with permission to do so can use `semctl()` to perform control operations.

Semaphore operations are performed by the `semop()` function. This call accepts a pointer to an array of semaphore operation structures. Each structure in the operations array contains information about an operation to perform on a semaphore. The operations array is described in detail in "Performing Semaphore Operations with semop()" later in this chapter.

Any process with read permission can test to see whether or not a semaphore has a zero value by supplying a `0` in the `sem_op` field of the operation structure. Operations to increment or decrement a semaphore require write permission.

When an attempt to perform any of the requested operations fails, none of the semaphores is altered. The process blocks unless the `IPC_NOWAIT` flag is set, and remains blocked until one of the following occurs:

- the semaphore operations all complete, so the call succeeds
- the process receives a signal
- the semaphore set is removed

When a semaphore operation fails, the call returns `-1` and sets `errno` appropriately.

Only one process can update a semaphore set at a time. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a `semop()` call, the updates are made atomically. That is, no updates are done until all operations in the array can complete in order successfully.

When a process performs an operation on a semaphore, the system does not usually keep track of whether or not that operation has been undone. If a process with exclusive use of a semaphore terminates abnormally and neglects to undo the operation or free the semaphore, the semaphore remains locked in memory.

To prevent this, `semop()` accepts the `SEM_UNDO` control flag. When this flag is in effect, `semop()` allocates an `undo` structure for each semaphore operation. That structure contains the operation needed to return the semaphore to its previous state.

When the process dies, the system applies the operations in the undo structures so an aborted process need not leave a semaphore set in an inconsistent state.

If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with `SEM_UNDO` in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this in order to restore the resource to a consistent state.

When performing a semaphore operation with `SEM_UNDO` in effect, you must also have it in effect for the call that would perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This insures that, unless the process is aborted, the values applied to the undo structure will eventually cancel out to zero. When the undo structure reaches zero, it is removed. Using `SEM_UNDO` inconsistently can lead to excessive resource consumption, because allocated undo structures might not be freed until the system is rebooted.

Structure of a Semaphore Set

A semaphore set is composed of a control structure with a unique ID and an array of semaphores. The identifier for the semaphore or array is called the `semid`.

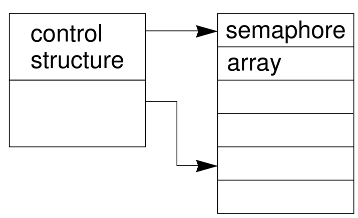

Figure 13-2 Structure of a Semaphore

The control structure for the semaphore contains the following information:

- the permissions structure
- a pointer to the first semaphore in the array
- the number of semaphores in the array
- the time of the last operation on any semaphore in the array
- the time of the last update to any semaphore in the array

Each semaphore structure in the array contains the following information:

- the semaphore value
- the PID of the process performing the last successful operation
- the number of processes waiting for the semaphore to increase
- the number of processes waiting for the semaphore to reach zero

The control structure is defined in `<sys/sem.h>`:

```
struct semid_ds
{
    struct ipc_permsem_perm;/* operation permission struct */
    struct sem*sem_base;/* ptr to first semaphore in set */
    ushortsem_nsems;/* # of semaphores in set */
    time_tsem_otime;/* last semop time */
    longsem_pad1;/* reserved for time_t expansion */
    time_tsem_ctime;/* last change time */
    longsem_pad2;/* time_t expansion */
    longsem_pad3[4];/* reserve area */
};
```

The `sem_perm` member of this structure uses `ipc_perm` (defined in `<sys/ipc.h>`) as a template.

The semaphore structure is defined in the same header file:

```
struct sem
{
    ushortsemval;/* semaphore text map address */
    pid_tsempid;/* pid of last operation */
    ushortsemncnt;/* # awaiting semval > cval */
    ushortsemzcnt;/* # awaiting semval = 0 */
};
```

Initializing a Semaphore Set with semget()

The `semget()` function initializes or gains access to a semaphore. When the call succeeds, it returns the semaphore ID (`semid`). When the call fails, it returns `-1` and sets the external variable `errno` to the appropriate error code.

The `semget()` function has the following synopsis:

Code Example 13-9 Synopsis of semget()

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

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

The `key` argument is a value associated with the semaphore ID.

The `nsems` argument specifies the number of elements in a semaphore array. The call fails when `nsems` is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument assures that it will succeed. The `semflg` argument specifies the initial access permissions and creation control flags.

The `SEMMNI` system configuration option determines the maximum number of semaphore arrays allowed. The `SEMMNS` option determines the maximum possible number of individual semaphores across all semaphore sets. `semget()` fails when one of these limits is exceeded. Because of fragmentation between semaphore sets, it might not be possible to allocate all available semaphores.

The following program illustrates the `semget()` function. It begins by prompting for a hexadecimal key, an octal permissions code, and control command combinations selected from a menu. All possible combinations are allowed. It then asks the number of semaphores in the array and issues the function to initialize the array. If the call succeeds, the program displays the returned semaphore ID. Otherwise, it displays an error message.

Code Example 13-10 Program to Illustrate semget()

```
/*
** semget.c: Illustrate the semget() function.
**
** This is a simple exerciser of the semget() function.
** It prompts for the arguments, makes the call, and reports the
** results.
*/

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

extern void    exit();
extern void    perror();
```

Code Example 13-10 Program to Illustrate semget() (Continued)

```
main()
{
    key_t   key;       /* key to pass to semget() */
    int     semflg;    /* semflg to pass to semget() */
    int     nsems;     /* nsems to pass to semget() */
    int     semid;     /* return value from semget() */

    (void) fprintf(stderr,
        "All numeric input must follow C conventions:\n");
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");
    (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
    (void) fprintf(stderr, "Enter key: ");
    (void) scanf("%li", &key);

    (void) fprintf(stderr, "Enter nsems value: ");
    (void) scanf("%i", &nsems);
    (void) fprintf(stderr, "\nExpected flags for semflg are:\n");
    (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
    (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
    (void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
    (void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);
    (void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
    (void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);
    (void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
    (void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);
    (void) fprintf(stderr, "Enter semflg value: ");
    (void) scanf("%i", &semflg);

    (void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %d, %#o)\n",
        key, nsems, semflg);

    if ((semid = semget(key, nsems, semflg)) == -1) {
        perror("semget: semget failed");
        exit(1);
    } else {
        (void) fprintf(stderr, "semget: semget succeeded: semid = %d\n",
            semid);
        exit(0);
    }
    /*NOTREACHED*/
}
```

Controlling Semaphores with semctl( )

The `semctl()` function allows a process to alter permissions and other characteristics of a semaphore set. Its synopsis is as follows:

Code Example 13-11 Synopsis of semctl()

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

unionsemun {
    int val;
    struct semid_ds *buf;
    ushort * array;
};

int semctl(int semid, int semnum, int cmd, union semun arg)
```

The `semid` function is a valid semaphore ID. The `semnum` function selects a semaphore within an array by its index. The `cmd` argument is one of the following control flags. What you supply for `arg` depends upon the control flag given in `cmd`.

- `GETVAL`
  Return the value of a single semaphore.
- `SETVAL`
  Set the value of a single semaphore. In this case, `arg` is taken as `arg.val`, an `int`.
- `GETPID`
  Return the PID of the process that performed the last operation on the semaphore or array.
- `GETNCNT`
  Return the number of processes waiting for the value of a semaphore to increase.
- `GETZCNT`
  Return the number of processes waiting for the value of a particular semaphore to reach zero.
- `GETALL`
  Return the values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of `unsigned shorts`.
- `SETALL`
  Set values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of `unsigned shorts`.

- IPC_STAT
  Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by arg.buf, a pointer to a buffer of type semid_ds.
- IPC_SET
  Set the effective user and group identification and permissions. In this case, arg is taken as arg.buf.
- IPC_RMID
  Remove the specified semaphore set.

A process must have an effective user identification of OWNER, CREATOR, or superuser to perform an IPC_SET or IPC_RMID command. Read and write permission is required as for the other control commands.

The following program illustrates semctl():

Code Example 13-12 Using semctl()

```
/*
** semctl.c:Illustrate the semctl() function.
**
** This is a simple exerciser of the semctl() function. It
** allows you to perform one control operation on one semaphore set.
** It gives up immediately if any control operation fails, so be careful not
** to set permissions to preclude read permission; you won't be able to reset
** the permissions with this code if you do.
*/

#include       <stdio.h>
#include       <sys/types.h>
#include       <sys/ipc.h>
#include       <sys/sem.h>
#include       <time.h>

struct semid_ds semid_ds;

static void do_semctl();
static void do_stat();
extern char *malloc();
extern void exit();
extern void perror();

char           warning_message[] = "If you remove read permission for\
       yourself, this program will fail frequently!";

main()
```

Code Example 13-12 Using semctl() (Continued)

```
{
    union semunarg;                /* union to pass to semctl() */
    int                 cmd,       /* command to give to semctl() */
                        i,         /* work area */
                        semid,     /* semid to pass to semctl() */
                        semnum;/* semnum to pass to semctl() */

    (void) fprintf(stderr,

        "All numeric input must follow C conventions:\n");
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");
    (void) fprintf(stderr, "Enter semid value: ");
    (void) scanf("%i", &semid);

    (void) fprintf(stderr, "Valid semctl cmd values are:\n");
    (void) fprintf(stderr, "\tGETALL = %d\n", GETALL);
    (void) fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
    (void) fprintf(stderr, "\tGETPID = %d\n", GETPID);
    (void) fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
    (void) fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
    (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
    (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
    (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
    (void) fprintf(stderr, "\tSETALL = %d\n", SETALL);
    (void) fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
    (void) fprintf(stderr, "\nEnter cmd: ");
    (void) scanf("%i", &cmd);

    /* Perform some setup operations needed by multiple commands. */
    switch (cmd) {
    case GETVAL:
    case SETVAL:
    case GETNCNT:
    case GETZCNT:
        /* Get the semaphore number for these commands. */
        (void) fprintf(stderr, "\nEnter semnum value: ");
        (void) scanf("%i", &semnum);
        break;

    case GETALL:
    case SETALL:
        /* Allocate a buffer for the semaphore values. */
        (void) fprintf(stderr,
```

Code Example 13-12 Using semctl() (Continued)

```
            "Get number of semaphores in the set.\n");
        arg.buf = &semid_ds;
        do_semctl(semid, 0, IPC_STAT, arg);
        if (arg.array =
            (ushort *)malloc((unsigned)
                (semid_ds.sem_nsems * sizeof(ushort)))) {
            /* Break out if you got what you needed. */
            break;

        }
        (void) fprintf(stderr,
            "semctl: unable to allocate space for %d values\n",
            semid_ds.sem_nsems);
        exit(2);
        /*NOTREACHED*/
    }

    /* Get the rest of the arguments needed for the specified command. */
    switch (cmd) {
    case SETVAL:
        /* Set value of one semaphore. */
        (void) fprintf(stderr, "\nEnter semaphore value: ");
        (void) scanf("%i", &arg.val);
        do_semctl(semid, semnum, SETVAL, arg);

        /* Fall through to verify the result. */
        (void) fprintf(stderr,
            "Perform semctl GETVAL command to verify results.\n");

    case GETVAL:
        /* Get value of one semaphore. */
        arg.val = 0;
        do_semctl(semid, semnum, GETVAL, arg);
        break;

    case GETPID:
        /* Get PID of last process to successfully complete a
            semctl(SETVAL), semctl(SETALL), or semop() on the semaphore. */
        arg.val = 0;
        do_semctl(semid, 0, GETPID, arg);
        break;

    case GETNCNT:
        /* Get number of processes waiting for semaphore value to increase. */
        arg.val = 0;
```

Code Example 13-12 Using semctl() (Continued)

```
        do_semctl(semid, semnum, GETNCNT, arg);
        break;

    case GETZCNT:
        /* Get number of processes waiting for semaphore value to become zero.
*/
        arg.val = 0;

        do_semctl(semid, semnum, GETZCNT, arg);
        break;

    case SETALL:
        /* Set the values of all semaphores in the set. */
        (void) fprintf(stderr, "There are %d semaphores in the set.\n",
            semid_ds.sem_nsems);
        (void) fprintf(stderr, "Enter semaphore values:\n");
        for (i = 0; i < semid_ds.sem_nsems; i++) {
            (void) fprintf(stderr, "Semaphore %d: ", i);
            (void) scanf("%hi", &arg.array[i]);
        }
        do_semctl(semid, 0, SETALL, arg);

        /* Fall through to verify the results. */
        (void) fprintf(stderr,
            "Perform semctl GETALL command to verify results.\n");

    case GETALL:
        /* Get and print the values of all semaphores in the set.*/
        do_semctl(semid, 0, GETALL, arg);
        (void) fprintf(stderr, "The values of the %d semaphores are:\n",
            semid_ds.sem_nsems);
        for (i = 0; i < semid_ds.sem_nsems; i++)
            (void) fprintf(stderr, "%d ", arg.array[i]);
        (void) fprintf(stderr, "\n");
        break;

    case IPC_SET:
        /* Modify mode and/or ownership. */
        arg.buf = &semid_ds;
        do_semctl(semid, 0, IPC_STAT, arg);
        (void) fprintf(stderr, "Status before IPC_SET:\n");
        do_stat();

        (void) fprintf(stderr, "Enter sem_perm.uid value: ");
        (void) scanf("%hi", &semid_ds.sem_perm.uid);
```

Code Example 13-12 Using semctl() (Continued)

```
        (void) fprintf(stderr, "Enter sem_perm.gid value: ");
        (void) scanf("%hi", &semid_ds.sem_perm.gid);

        (void) fprintf(stderr, "%s\n", warning_message);
        (void) fprintf(stderr,

            "Enter sem_perm.mode value: ");
        (void) scanf("%hi", &semid_ds.sem_perm.mode);

        do_semctl(semid, 0, IPC_SET, arg);

        /* Fall through to verify changes. */
        (void) fprintf(stderr, "Status after IPC_SET:\n");

    case IPC_STAT:
        /* Get and print current status. */
        arg.buf = &semid_ds;
        do_semctl(semid, 0, IPC_STAT, arg);
        do_stat();
        break;

    case IPC_RMID:
        /* Remove the semaphore set. */
        arg.val = 0;
        do_semctl(semid, 0, IPC_RMID, arg);
        break;

    default:
        /* Pass unknown command to semctl. */
        arg.val = 0;
        do_semctl(semid, 0, cmd, arg);
        break;
    }
    exit(0);
    /*NOTREACHED*/
}

/*
** Print indication of arguments being passed to semctl(), call semctl(),
** and report the results.
** If semctl() fails, do not return; this example doesn't deal with
** errors, it just reports them.
*/
static void
```

Code Example 13-12 Using semctl() (Continued)

```
do_semctl(semid, semnum, cmd, arg)
union semun arg;
int     cmd,
            semid,
            semnum;
{

    register int              i;         /* work area */

    void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d, ",
        semid, semnum, cmd);
    switch (cmd) {
    case GETALL:
            (void) fprintf(stderr, "arg.array = %#x)\n", arg.array);
            break;
    case IPC_STAT:
    case IPC_SET:
        (void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
        break;
    case SETALL:
            (void) fprintf(stderr, "arg.array = [", arg.buf);
            for (i = 0;i < semid_ds.sem_nsems;) {
                (void) fprintf(stderr, "%d", arg.array[i++]);
                if (i < semid_ds.sem_nsems)
                        (void) fprintf(stderr, ", ");
            }
            (void) fprintf(stderr, "])\n");
            break;
    case SETVAL:
    default:
            (void) fprintf(stderr, "arg.val = %d)\n", arg.val);
            break;
    }
    i = semctl(semid, semnum, cmd, arg);
    if (i == -1) {
        perror("semctl: semctl failed");
            exit(1);
            /* NOTREACHED */
    }
    (void) fprintf(stderr, "semctl: semctl returned %d\n", i);
    return;
}

/*
**  Display contents of commonly used pieces of the status structure.
```

Code Example 13-12 Using semctl() (Continued)

```
*/
static void
do_stat()
{
    (void) fprintf(stderr, "sem_perm.uid = %d\n", semid_ds.sem_perm.uid);
    (void) fprintf(stderr, "sem_perm.gid = %d\n", semid_ds.sem_perm.gid);
    (void) fprintf(stderr, "sem_perm.cuid = %d\n", semid_ds.sem_perm.cuid);
    (void) fprintf(stderr, "sem_perm.cgid = %d\n", semid_ds.sem_perm.cgid);
    (void) fprintf(stderr, "sem_perm.mode = %#o, ",
        semid_ds.sem_perm.mode);
    (void) fprintf(stderr, "access permissions = %#o\n",
        semid_ds.sem_perm.mode & 0777);
    (void) fprintf(stderr, "sem_nsems = %d\n", semid_ds.sem_nsems);
    (void) fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
        ctime(&semid_ds.sem_otime) : "Not Set\n");
    (void) fprintf(stderr, "sem_ctime = %s", ctime(&semid_ds.sem_ctime));
}
```

Performing Semaphore Operations with `semop()`

The `semop()` function performs operations on a semaphore set. Its synopsis is as follows:

Code Example 13-13 Synopsis of semop()

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

    int semop(int semid, struct sembuf *sops, size_t nsops);
```

The `semid` argument is the semaphore ID returned by a previous `semget()` call. The `sops` argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

- the semaphore number
- the operation to be performed
- control flags, if any

`The sembuf` structure specifies a semaphore operation, as defined in `<sys/sem.h>`.

```
struct sembuf {
    ushortsem_num;/* semaphore # */
    shortsem_op;/* semaphore operation */
    shortsem_flg;/* operation flags */
};
```

The nsops argument specifies the length of the array, the maximum size of which is determined by the SEMOPM configuration option; this is the maximum number of operations allowed by a single semop() call, and is set to 10 by default.

The operation to be performed is determined as follows:

- A positive integer increments the semaphore value by that amount.
- A negative integer decrements the semaphore value by that amount. However, a semaphore can never take on a negative value. An attempt to set a semaphore to a value below zero either fails or blocks, depending on whether or not IPC_NOWAIT is in effect.
- A value of zero means to wait for the semaphore value to reach zero.

The following control flags can be used with semop():

- IPC_NOWAIT
  This operation command can be set for any operations in the array. The function returns unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The function will be unsuccessful when trying to decrement a semaphore more than its current value or when testing for a semaphore to be equal to zero when it is not.
- SEM_UNDO
  This command allows individual operations in the array to be undone when the process exits.

The following program illustrates the semop() function:

Code Example 13-14 Program to Illustrate semop()

```
/*
** semop.c: Illustrate the semop() function.
**
** This is a simple exerciser of the semop() function. It allows
** you to set up arguments for semop() and make the call. It then reports
** the results repeatedly on one semaphore set. You must have read
** permission on the semaphore set or this exerciser will fail. (It needs
** read permission to get the number of semaphores in the set and to report
** the values before and after calls to semop().)
*/

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

static int  ask();
```

Code Example 13-14 Program to Illustrate semop() (Continued)

```
extern void exit();
extern void free();
extern char *malloc();
extern void perror();

static struct semid_dssemid_ds;                /* status of semaphore set */

static charerror_mesg1[] = "semop: Can't allocate space for %d\
    semaphore values. Giving up.\n";
static charerror_mesg2[] = "semop: Can't allocate space for %d\
    sembuf structures. Giving up.\n";
main()
{
    register int              i;          /* work area */
    int                       nsops;      /* number of operations to perform */
    int                       semid;      /* semid of semaphore set */
    struct         sembuf     *sops;      /* ptr to operations to perform */

    (void) fprintf(stderr,
            "All numeric input must follow C conventions:\n");
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");
    /* Loop until the invoker doesn't want to do anymore. */

    while (nsops = ask(&semid, &sops)) {
        /* Initialize the array of operations to be performed.*/
        for (i = 0; i < nsops; i++) {
            (void) fprintf(stderr,
                "\nEnter values for operation %d of %d.\n",
                i + 1, nsops);
            (void) fprintf(stderr,
                "sem_num(valid values are 0 <= sem_num < %d): ",
                semid_ds.sem_nsems);
            (void) scanf("%hi", &sops[i].sem_num);
            (void) fprintf(stderr, "sem_op: ");
            (void) scanf("%hi", &sops[i].sem_op);
            (void) fprintf(stderr,
                "Expected flags in sem_flg are:\n");
            (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",
                IPC_NOWAIT);
            (void) fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",
                SEM_UNDO);
            (void) fprintf(stderr, "sem_flg: ");
            (void) scanf("%hi", &sops[i].sem_flg);
```

Code Example 13-14 Program to Illustrate semop() (Continued)

```
        }

        /* Recap the call to be made. */
        (void) fprintf(stderr,
            "\nsemop: Calling semop(%d, &sops, %d) with:",
            semid, nsops);
        for (i = 0; i < nsops; i++)
        {
            (void) fprintf(stderr, "\nsops[%d].sem_num = %d, ", i,
                sops[i].sem_num);
            (void) fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
            (void) fprintf(stderr, "sem_flg = %#o\n",
                sops[i].sem_flg);
        }

        /* Make the semop() call and report the results. */
        if ((i = semop(semid, sops, nsops)) == -1) {
            perror("semop: semop failed");
        } else {
            (void) fprintf(stderr, "semop: semop returned %d\n", i);
            }
    }
    /*NOTREACHED*/
}
/*

** Ask user if (s)he wants to continue.
**
** On the first call:
**  Get the semid to be processed and supply it to the caller.
** On each call:
**  1. Print current semaphore values.
**  2. Ask user how many operations are to be performed on the next call to
**   semop. Allocate an array of sembuf structures sufficient for the
**    job and set caller-supplied pointer to that array. (The array
**    is reused on subsequent calls if it is big enough. If
**    it isn't, it is freed and a larger array is allocated.)
*/
static
ask(semidp, sopsp)
int        *semidp;/* pointer to semid (used only the first time) */
struct sembuf**sopsp;
{
    ushort *array;/* argument to semctl */
    int            i;      /* work area */
```

Code Example 13-14 Program to Illustrate semop() (Continued)

```
    static int     nsops = 0;/* size of currently allocated
                               sembuf array */
    static int     semid = -1;    /* semid supplied by user */
    static struct sembuf*sops;        /* pointer to allocated array */

    if (semid < 0) {

        /* First call; get semid from user and the current state of the
semaphore set. */
        (void) fprintf(stderr,
            "Enter semid of the semaphore set you want to use: ");
        (void) scanf("%i", &semid);
        *semidp = semid;
        arg.buf = &semid_ds;
        if (semctl(semid, 0, IPC_STAT, arg) == -1) {
            perror("semop: semctl(IPC_STAT) failed");

            /* Note that if semctl fails, semid_ds remains filled with
                zeros, so later test for number of semaphores will be zero. */
            (void) fprintf(stderr,
                "Before and after values will not be printed.\n");
        } else {
            if ((array = (ushort *)malloc(
                (unsigned)(sizeof(ushort) * semid_ds.sem_nsems)))
                == NULL) {
                (void) fprintf(stderr, error_mesg1,
                    semid_ds.sem_nsems);
                exit(1);
            }
        }
    }
    /* Print current semaphore values. */
    if (semid_ds.sem_nsems) {
        (void) fprintf(stderr, "There are %d semaphores in the set.\n",
            semid_ds.sem_nsems);
        if (semctl(semid, 0, GETALL, array) == -1) {
            perror("semop: semctl(GETALL) failed");
        } else {
            (void) fprintf(stderr, "Current semaphore values are:");
            for (i = 0; i < semid_ds.sem_nsems;
                (void) fprintf(stderr, " %d", array[i++]))
                ;
            (void) fprintf(stderr, "\n");
        }
    }
```

Code Example 13-14 Program to Illustrate semop() (Continued)

```
    /* Find out how many operations are going to be done in the next
       call and allocate enough space to do it. */
    (void) fprintf(stderr, "How many semaphore operations do you want %s\n",
            "on the next call to semop()?");
    (void) fprintf(stderr, "Enter 0 or control-D to quit: ");
    i = 0;
    if (scanf("%i", &i) == EOF || i == 0)
        exit(0);
    if (i > nsops) {
        if (nsops)
            free((char *)sops);
        nsops = i;
        if ((sops = (struct sembuf *)malloc((unsigned)(nsops *
            sizeof(struct sembuf)))) == NULL) {
            (void) fprintf(stderr, error_mesg2, nsops);
            exit(2);
        }
    }
    *sopsp = sops;
    return (i);
}
```

Shared Memory

Developers intending to use shared memory as an IPC mechanism have a choice in SunOS 5.x. The first choice is the mechanism belonging to the unified IPC package being described. The second choice is to use the native virtual memory management facilities of SunOS 5.x. The latter choice, to be discussed in a later section of this chapter is recommended for new application development.

The shared memory mechanism of the IPC package allows more than one process at a time to attach a segment of physical memory to its virtual address space. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

In the SunOS 5.x operating system, the most efficient way to implement shared memory applications is to rely on native virtual memory management and the `mmap()` function.

Shared memory allows more than one process at a time to attach a segment of physical memory to its virtual address space. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using the `shmget()` function. This call can also be used to obtain the ID of an existing shared segment. The creating process sets the permissions and the size in bytes for the segment.

The original owner or creator of a shared memory segment can assign ownership to another user with the `shmctl()` function; it can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`.

Once created, a shared segment can be attached to a process's address space using the `shmat()` function; it can be detached using `shmdt()`. (For more details, see the man page for `shmop()`.) The attaching process must have the appropriate permissions for `shmat()` to succeed. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment may be attached multiple times by the same process.

If the `shmat()` function fails, it returns `-1`, and sets the external variable `errno` to the appropriate value.

Structure of a Shared Memory Segment

A *shared memory segment* is composed of a control structure with a unique ID that points to an area of physical memory. The identifier for the segment is referred to as the `shmid`.

Figure 13-3 Structure of a Shared Memory Segment

The data structure includes the following information about the memory segment:

- access permissions
- segment size
- the PID of the process performing last operation
- the PID of the creator process
- the current number of processes to which the segment is attached
- the time of the last attachment
- the time of the last detachment
- the time of the last change to the segment
- memory map segment descriptor pointer

The structure definition for the shared memory segment control structure can be found in <sys/shm.h>. This structure definition is shown in the following code sample:

Code Example 13-15 Structure Definition for shmid

```
/*
 *  There is a shared mem id data structure for each segment in the system.
 */

struct shmid_ds {
    struct ipc_permshm_perm;/* operation permission struct */
    int shm_segsz;/* size of segment in bytes */
    struct anon_map*shm_amp;/* segment anon_map pointer */
    ushortshm_lkcnt;/* number of times it is being locked */
    pid_tshm_lpid;/* pid of last shmop */
    pid_tshm_cpid;/* pid of creator */
    ulongshm_nattch;/* used only for shminfo */
    ulongshm_cnattch;/* used only for shminfo */
    time_tshm_atime;/* last shmat time */
    longshm_pad1;/* reserved for time_t expansion */
    time_tshm_dtime;/* last shmdt time */
    longshm_pad2;/* reserved for time_t expansion */
    time_tshm_ctime;/* last change time */
    longshm_pad3;/* reserved for time_t expansion */
    longshm_pad4[4];/* reserve area */
};
```

Note that the shm_perm member of this structure uses ipc_perm as a template, as defined in <sys/ipc.h>.

Using **shmget()** to Get Access to a Shared Memory Segment

The shmget() function is used to obtain access to a shared memory segment. When the call succeeds, it returns the shared memory segment ID (shmid). When it fails, it returns -1 and sets errno to the appropriate error code. shmget() has the following synopsis:

Code Example 13-16 Synopsis of shmget

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

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

The value passed as the `shmflg` argument must be an integer, which incorporates settings for the segment's permissions and control flags, as described under "Permissions" earlier in this chapter.

The `SHMMNI` system configuration option determines the maximum number of shared memory segments that are allowed, 100 by default.

The function will fail if the `size` value is less than `SHMMIN` or greater than `SHMMAX`, the configuration options for the minimum and maximum segment sizes. By default, `SHMIN` is 1, `SHMAX` is 131072.

The following sample program illustrates the `shmget()` function:

Code Example 13-17 Program to Illustrate shmget()

```
/*
** shmget.c: Illustrate the shmget() function.
**
**  This is a simple exerciser of the shmget() function.
** It prompts for the arguments, makes the call, and reports the results.
*/

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

extern voidexit();
extern voidperror();

main()
{
    key_tkey;/* key to be passed to shmget() */
    int shmflg;/* shmflg to be passed to shmget() */
    int shmid;/* return value from shmget() */
    int size;/* size to be passed to shmget() */

    (void) fprintf(stderr,
        "All numeric input is expected to follow C conventions:\n");
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");

    /* Get the key. */
    (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
    (void) fprintf(stderr, "Enter key: ");
    (void) scanf("%li", &key);
```

Code Example 13-17 Program to Illustrate shmget() (Continued)

```
    /* Get the size of the segment. */
    (void) fprintf(stderr, "Enter size: ");
    (void) scanf("%i", &size);

    /* Get the shmflg value. */
    (void) fprintf(stderr, "Expected flags for the shmflg argument are:\n");
    (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
    (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
    (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
    (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
    (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
    (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
    (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
    (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
    (void) fprintf(stderr, "Enter shmflg: ");
    (void) scanf("%i", &shmflg);

    /* Make the call and report the results. */
    (void) fprintf(stderr, "shmget: Calling shmget(%#lx, %d, %#o)\n",
        key, size, shmflg);
    if ((shmid = shmget (key, size, shmflg)) == -1) {
        perror("shmget: shmget failed");
        exit(1);
    } else {
        (void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
        exit(0);
    }
    /*NOTREACHED*/
}
```

Controlling a Shared Memory Segment with **`shmctl()`**

The `shmctl()` function is used to alter the permissions and other characteristics of a shared memory segment. Its synopsis is as follows:

Code Example 13-18 Synopsis of shmctl()

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

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

The shmid argument is the ID of the shared memory segment as returned by shmget(). The cmd argument is one of following control commands:

- SHM_LOCK
  Lock the specified shared memory segment in memory. The process must have an effective ID of super-user to perform this command.
- SHM_UNLOCK
  Unlock the shared memory segment. The process must have an effective ID of superuser to perform this command.
- IPC_STAT
  Return the status information contained in the control structure, and place it in the buffer pointed to by buf. The process must have read permission on the segment to perform this command.
- IPC_SET
  Set the effective user and group identification, and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.
- IPC_RMID
  Remove the shared memory segment. The process must have an effective ID of owner, creator or superuser to perform this command.

The example program below allows you to illustrate shmctl():

Code Example 13-19 Program to Illustrate shmctl()

```
/*
** shmctl.c: Illustrate the shmctl() function.
**
** This is a simple exerciser of the shmctl() function. It allows
** you to perform one control operation on one shared memory segment.
** (Some operations are done for the user whether requested or not. It gives
** up immediately if any control operation fails. Be careful not to set
** permissions to preclude read permission; you won't be able to reset the
** permissions with this code if you do.)
*/

#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<time.h>


static voiddo_shmctl();
```

Code Example 13-19 Program to Illustrate shmctl() (Continued)

```
extern voidexit();
extern voidperror();

main()
{
    int     cmd;/* command code for shmctl() */
    int     shmid;/* segment ID */
    struct shmid_dsshmid_ds;/* shared memory data structure to hold results */

    (void) fprintf(stderr,
        "All numeric input is expected to follow C conventions:\n");
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");

    /* Get shmid and cmd. */
    (void) fprintf(stderr, "Enter the shmid for the desired segment: ");
    (void) scanf("%i", &shmid);
    (void) fprintf(stderr, "Valid shmctl cmd values are:\n");
    (void) fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
    (void) fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
    (void) fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
    (void) fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
    (void) fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
    (void) fprintf(stderr, "Enter the desired cmd value: ");
    (void) scanf("%i", &cmd);

    switch (cmd) {
    case IPC_STAT:
        /* Get shared memory segment status. */
        break;

    case IPC_SET:
        /* Set owner UID and GID and permissions. */
        /* Get and print current values. */
        do_shmctl(shmid, IPC_STAT, &shmid_ds);

        /* Set UID, GID, and permissions to be loaded. */
        (void) fprintf(stderr, "\nEnter shm_perm.uid: ");

            (void) scanf("%hi", &shmid_ds.shm_perm.uid);
            (void) fprintf(stderr, "Enter shm_perm.gid: ");
            (void) scanf("%hi", &shmid_ds.shm_perm.gid);
            (void) fprintf(stderr,
                "Note: Keep read permission for yourself.\n");
```

Code Example 13-19 Program to Illustrate shmctl() (Continued)

```
        (void) fprintf(stderr, "Enter shm_perm.mode: ");
        (void) scanf("%hi", &shmid_ds.shm_perm.mode);
        break;

    case IPC_RMID:
            /* Remove the segment when the last attach point is detached. */
            break;

    case SHM_LOCK:
            /* Lock the shared memory segment. */
            break;

    case SHM_UNLOCK:
            /* Unlock the shared memory segment. */
            break;

    default:
            /* Unknown command will be passed to shmctl. */
            break;
    }
    do_shmctl(shmid, cmd, &shmid_ds);
    exit(0);
    /*NOTREACHED*/
}
/*
** Display the arguments being passed to shmctl(), call shmctl(), and report
the results.
** If shmctl() fails, do not return; this example doesn't deal with
** errors, it just reports them.
*/
static void
do_shmctl(shmid, cmd, buf)
int         shmid, /* attach point */
            cmd;   /* command code */
struct shmid_ds*buf;/* pointer to shared memory data structure */
{
register intrtrn;  /* hold area */
    (void) fprintf(stderr, "shmctl: Calling shmctl(%d, %d, buf)\n",
        shmid, cmd);
    if (cmd == IPC_SET) {
        (void) fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
            buf->shm_perm.uid);
        (void) fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
            buf->shm_perm.gid);
        (void) fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
```

Code Example 13-19 Program to Illustrate shmctl() (Continued)

```
            buf->shm_perm.mode);
    }
    if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
        perror("shmctl: shmctl failed");
        exit(1);
    } else {
        (void) fprintf(stderr, "shmctl: shmctl returned %d\n", rtrn);
    }
    if (cmd != IPC_STAT && cmd != IPC_SET)
        return;

    /* Print the current status. */
    (void) fprintf(stderr, "\nCurrent status:\n");
    (void) fprintf(stderr, "\tshm_perm.uid = %d\n", buf->shm_perm.uid);
    (void) fprintf(stderr, "\tshm_perm.gid = %d\n", buf->shm_perm.gid);
    (void) fprintf(stderr, "\tshm_perm.cuid = %d\n", buf->shm_perm.cuid);
    (void) fprintf(stderr, "\tshm_perm.cgid = %d\n", buf->shm_perm.cgid);
    (void) fprintf(stderr, "\tshm_perm.mode = %#o\n", buf->shm_perm.mode);
    (void) fprintf(stderr, "\tshm_perm.key = %#x\n", buf->shm_perm.key);
    (void) fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
    (void) fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
    (void) fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
    (void) fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
    (void) fprintf(stderr, "\tshm_atime = %s",
        buf->shm_atime ? ctime(&buf->shm_atime) : "Not Set\n");
    (void) fprintf(stderr, "\tshm_dtime = %s",
        buf->shm_dtime ? ctime(&buf->shm_dtime) : "Not Set\n");
    (void) fprintf(stderr, "\tshm_ctime = %s", ctime(&buf->shm_ctime));
}
```

Attaching and Detaching a Shared Memory Segment with `shmat()` and `shmdt()`

The `shmat()` and `shmdt()` functions are used to attach and detach shared memory segments. Their synopses are as follows:

Code Example 13-20 Synopses of shmat() and shmdt()

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

void *shmat(int shmid, void *shmaddr, int shmflg);

int shmdt (void *shmaddr);
```

Upon successful completion, the `shmat()` function returns a pointer to the head of the shared segment; when unsuccessful, it returns `(void *)-1` and sets the external variable `errno` to the appropriate error code.

The `shmid` argument is the ID of an existing shared memory segment. The `shmaddr` argument is the address at which to attach the segment. If supplied as zero, the system provides a suitable address. For portability, it is usually better to allow the system to determine the address.

The `shmflg` argument is a control flag used to pass the `SHM_RND` and `SHM_RDONLY` flags to the `shmat()` function.

The `shmdt()` function detaches the shared memory segment located at the address indicated by `shmaddr`. Upon successful completion, `schmdt()` returns zero; when unsuccessful, it returns `-1` and sets the external variable `errno` to the appropriate error code.

The following sample program illustrates `shmat()` and `shmdt()`:

Code Example 13-21 Using shmat() and shmdt()

```
/*
** shmop.c: Illustrate the shmat() and shmdt() functions.
**
** This is a simple exerciser for the shmat() and shmdt() system
** calls. It allows you to attach and detach segments and to
** write strings into and read strings from attached segments.
*/

#include<stdio.h>
#include<setjmp.h>
#include<signal.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

#define MAXnap4 /* Maximum number of concurrent attaches. */

static  ask();
static voidcatcher();
extern voidexit();
static  good_addr();
extern voidperror();
extern char*shmat();

static struct state{/* Internal record of currently attached segments. */
    int     shmid;/* shmid of attached segment */
    char    *shmaddr;/* attach point */
```

Code Example 13-21 Using shmat() and shmdt() (Continued)

```
    int     shmflg;/* flags used on attach */
}   ap[MAXnap];/* State of current attached segments. */

static int nap;/* Number of currently attached segments. */
static jmp_bufsegvbuf;/* Process state save area for SIGSEGV catching. */

main()
{
    register intaction;/* action to be performed */
    char       *addr;  /* address work area */
    register inti;                    /* work area */
    register struct state*p;/* ptr to current state entry */
    void       (*savefunc)();/* SIGSEGV state hold area */

    (void) fprintf(stderr,
           "All numeric input is expected to follow C conventions:\n");
    (void) fprintf(stderr, "\t0x... is interpreted as hexadecimal,\n");
    (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
    (void) fprintf(stderr, "\totherwise, decimal.\n");
    while (action = ask()) {
           if (nap) {
               (void) fprintf(stderr,
                      "\nCurrently attached segment(s):\n");
           (void) fprintf(stderr, " shmid address\n");
           (void) fprintf(stderr, "------ ----------\n");
        p = &ap[nap];
           while (p-- != ap) {
               (void) fprintf(stderr, "%6d", p->shmid);
               (void) fprintf(stderr, "%#11x", p->shmaddr);
               (void) fprintf(stderr, " Read%s\n",
                   (p->shmflg & SHM_RDONLY) ?
                   "-Only" : "/Write");
           }
        } else
           (void) fprintf(stderr,
               "\nNo segments are currently attached.\n");
        switch (action) {
        case 1:    /* Shmat requested. */
           /* Verify that there is space for another attach. */
           if (nap == MAXnap) {
               (void) fprintf(stderr, "%s %d %s\n",
                      "This simple example will only allow",
                      MAXnap, "attached segments.");
               break;
           }
```

Code Example 13-21 Using shmat() and shmdt() (Continued)

```
            p = &ap[nap++];

            /* Get the arguments, make the call, report the
                results, and update the current state array. */
            (void) fprintf(stderr,
                "Enter shmid of segment to attach: ");
            (void) scanf("%i", &p->shmid);

            (void) fprintf(stderr, "Enter shmaddr: ");
            (void) scanf("%i", &p->shmaddr);

            (void) fprintf(stderr,
                "Meaningful shmflg values are:\n");
            (void) fprintf(stderr, "\tSHM_RDONLY = \t%#8.8o\n",
                SHM_RDONLY);
            (void) fprintf(stderr, "\tSHM_RND = \t%#8.8o\n",
                SHM_RND);
            (void) fprintf(stderr, "Enter shmflg value: ");
            (void) scanf("%i", &p->shmflg);

            (void) fprintf(stderr,
                "shmop: Calling shmat(%d, %#x, %#o)\n",
                p->shmid, p->shmaddr, p->shmflg);
            p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
            if(p->shmaddr == (char *)-1) {
                perror("shmop: shmat failed");
                nap--;
            } else {
                (void) fprintf(stderr,
                    "shmop: shmat returned %#8.8x\n",
                    p->shmaddr);
            }
            break;

        case 2:     /* Shmdt requested. */
            /* Get the address, make the call, report the results,
                and make the internal state match. */
            (void) fprintf(stderr,
                "Enter detach shmaddr: ");
            (void) scanf("%i", &addr);

            i = shmdt(addr);
            if(i == -1) {
                perror("shmop: shmdt failed");
            } else {
```

Code Example 13-21 Using shmat() and shmdt() (Continued)

```
                (void) fprintf(stderr,
                    "shmop: shmdt returned %d\n", i);
                for (p = ap, i = nap; i--; p++) {
                    if (p->shmaddr == addr)
                        *p = ap[--nap];
                }
            }
            break;
        case 3:/* Read from segment requested. */
            if (nap == 0)

                break;

            (void) fprintf(stderr, "Enter address of an %s",
                "attached segment: ");
            (void) scanf("%i", &addr);

            if (good_addr(addr))
                (void) fprintf(stderr, "String @ %#x is ‘%s’\n",
                    addr, addr);
            break;

        case 4:/* Write to segment requested. */
            if (nap == 0)
                break;

            (void) fprintf(stderr, "Enter address of an %s",
                "attached segment: ");
            (void) scanf("%i", &addr);

            /* Set up SIGSEGV catch routine to trap attempts to
                write into a read-only attached segment. */
            savefunc = signal(SIGSEGV, catcher);

            if (setjmp(segvbuf)) {
                (void) fprintf(stderr, "shmop: %s: %s\n",
                    "SIGSEGV signal caught",
                    "Write aborted.");
            } else {
                if (good_addr(addr)) {
                    (void) fflush(stdin);
                    (void) fprintf(stderr, "%s %s %#x:\n",
                        "Enter one line to be copied",
                        "to shared segment attached @",
                        addr);
```

Code Example 13-21 Using shmat() and shmdt() (Continued)

```
                    (void) gets(addr);
                }
            }
            (void) fflush(stdin);

            /* Restore SIGSEGV to previous condition. */
            (void) signal(SIGSEGV, savefunc);
            break;
        }
    }

    exit(0);
    /*NOTREACHED*/
}
/*
**  Ask for next action.
*/
static
ask()
{
    int     response;  /* user response */
    do {
            (void) fprintf(stderr, "Your options are:\n");
            (void) fprintf(stderr, "\t^D = exit\n");
            (void) fprintf(stderr, "\t 0 = exit\n");
            (void) fprintf(stderr, "\t 1 = shmat\n");
            (void) fprintf(stderr, "\t 2 = shmdt\n");
            (void) fprintf(stderr, "\t 3 = read from segment\n");
            (void) fprintf(stderr, "\t 4 = write to segment\n");
            (void) fprintf(stderr,
                "Enter the number corresponding to your choice: ");

            /* Preset response so "^D" will be interpreted as exit. */
            response = 0;
            (void) scanf("%i", &response);
    } while (response < 0 || response > 4);
    return (response);
}
/*
** Catch signal caused by attempt to write into shared memory segment
** attached with SHM_RDONLY flag set.
*/
/*ARGSUSED*/
static void
catcher(sig)
```

Code Example 13-21 Using shmat() and shmdt() (Continued)

```
{
    longjmp(segvbuf, 1);
    /*NOTREACHED*/
}
/*
** Verify that given address is the address of an attached segment.
** Return 1 if address is valid; 0 if not.
*/
static
good_addr(address)
char*address;
{
    register struct state                *p;        /* ptr to state of
attached segment */

    for (p = ap; p != &ap[nap]; p++)
            if (p->shmaddr == address)
                return(1);
    return(0);
}
```

Advanced Methods for IPC

In addition to the basic and standard IPC packages, several important alternatives exist. For many applications, these alternatives have clear advantages over traditional IPC methods.

Virtual Memory Management

This section discusses the native virtual memory management facilities of SunOS 5.x. These facilities are the recommended way of implementing the shared memory model of interprocess communication.

The SunOS 5.x system provides a complete set of memory management mechanisms, providing applications complete control over the construction of their address space and permitting a wide variety of operations on both process address spaces and the variety of memory objects in the system.

Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated. Typically, the system presents you with mappings that simulate the traditional UNIX process memory environment, but other views of memory are useful as well.

The SunOS memory-management facilities do the following:

- Unify system operations on memory.
- Provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services without special-purpose kernel support.
- Maintain consistency with the existing environment, in particular using the file system as the namespace for named virtual-memory objects.

The fundamental function for defining the contents of an address space is `mmap()`. It establishes a mapping between a process address space and an object in the system virtual memory.

The advantage of using `mmap()` for sharing memory among processes is that it is the native mechanism of SunOS for managing process address spaces. The kernel also uses `mmap()` to perform disk file I/O in the `read()` and `write()` functions—it maps a file into the address space of the process and then copies data into and out of buffers. To share memory among processes, a file must be mapped into the address spaces of the processes. The most direct and efficient way to do this is with `mmap()`.

As in the Standard IPC Shared Memory mechanism, you must use an outside protocol or mechanism such as a semaphore to coordinate write access to shared memory. The `mmap()` function lets you specify protection (permissions) on mapped pages. The permissions are read, write, and execute. To use a page as shared memory, specify `MAP_SHARED` as the mapping type.

The `munmap()` operation removes mappings for a page or a range of pages.

For more information, see Chapter 7 of the *SunOS 5.2 System Services* and the *Virtual Memory Architecture in SunOS* (a white paper).

Locking

When cooperating processes share files they must coordinate access to the files just as they must with shared memory. File and record locking are sometimes used for this coordination.

File and record locking provide a synchronization mechanism for programs simultaneously accessing the same stores of data. Such processing is characteristic of many multiuser applications.

File and record locking is discussed in Chapter 3 of *SunOS 5.2 System Services*.

Multithreading Functions

The new multithreading facilities, native to SunOS 5.x, provide a means of synchronization that works not only between threads but also between processes. Use of these synchronization primitives, especially in combination with `mmap()` memory mapping, leads to a unified style of programming for both multithread and cooperating-process implementations.

Threads in two or more processes can use a single synchronization variable jointly. This requires that the variable be allocated in shared memory. Sharing the synchronization variable must also be specified when it is initialized. Specify `USYNC_PROCESS` to allow interprocess use; otherwise, for local use only, specify `USYNC_THREAD`.

The synchronization primitives in Solaris multithreading are mutual exclusion locks (mutexes), condition variables, semaphores, and read/write locks. For more information, see the chapter Multithreading and Multiprocessing in this book.

Using Remote Procedure Calls for IPC

Developers should consider using the Remote Procedure Call (RPC) facility for interprocess communication. RPC is discussed in the chapter on "Networking". Even though an application may be intended for single-system operation, it could benefit from using RPC as its interprocess communication service.

Network-Ready IPC

You can use RPC to develop applications that are "network-ready", that is, easily made into distributed applications. RPC—especially the Transport Independent RPC in Solaris 2.x—hides much of the networking details from the application. Therefore, a single-system application using RPC would not need to contain a lot of superfluous networking logic.

On the other hand, RPC provides extensive support of client/server interaction. Thus RPC is especially suitable as an IPC service for applications having a client/server structure.

Using XDR for Portability

RPC's companion service, eXternal Data Representation (XDR), is useful for IPCs involving the exchange of structured data. If the data structures are machine-dependent, the communicating processes can benefit from using XDR to encode the data. Such applications will then be ready for heterogeneous distributed computing, where the communicating processes could be running on machines with different processor architectures.

Administering IPC

SunOS 5.x provides the following facilities for you to manage the IPC services in your system.

System Limits and Kernel Tuning

A number of tunable kernel parameters apply to the advanced IPC package (messages, semaphores, and shared memory). The procedure for changing or tuning these parameters is described in Appendix A of *SunOS 5.2 Administering Security, Performance, and Accounting*.

For each mechanism, the maximum number of identifiers can be set. Other examples of maximums that can be set include message size, queue length, number of semaphores in the system, shared memory segment size, and number of shared memory segments per process.

Checking Message and Semaphore Activity

The `ipcs` utility reports a broad range of status information about active interprocess communication facilities. See the man page for the `ipcs` utility for more details.

The `ipcrm` utility removes one or more messages, semaphores, or shared memory identifiers.

The `sar -m` command reports current message and semaphore activities in number of operations per second. These figures will usually be zero, unless applications that use messages or semaphores are running.

Managing the IPC Loadable Modules

The IPC modules, `msgsys`, `semsys`, and `shmsys`, are not loaded into the system until they are used by an application, unless you explicitly load them or force load them at boot time with an entry in `/etc/system`.

If you issue the `ipcs` command when one or more of these modules has not been loaded, `ipcs` will report, for example:

```
Message Queue facility not in system
```

As soon as an application attempts to use one of the facilities, it will be loaded automatically.

Bibliography

SunOS 5.2 Administering Security, Performance, and Accounting, Part Number 800-7362-10, SunSoft, Inc., Mountain View, CA.

SunOS 5.2 System Services, Part Number 800-6542-10, SunSoft, Inc., Mountain View, CA.

SunOS Virtual Memory Implementation, Joseph P. Moran, Sun Microsystems, Inc., Mountain View, CA, 1988.

Virtual Memory Architecture in SunOS, Robert A. Gingell, Joseph P. Moran, and William A. Shannon, Sun Microsystems, Inc., Mountain View, CA, 1987.