9. INTERPROCESS COMMUNICATION

Introduction

The operating system supports three types of Inter-Process Communication (IPC):

- messages
- semaphores
- shared memory

This chapter describes the system calls for each type of IPC.

Included in the chapter are several example programs that show the use of the IPC system calls. All the example programs have been compiled and run.

Since there are many ways in the C Programming Language to accomplish the same task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency. Usually, system calls are embedded within a larger user-written program that makes use of a particular function that the calls provide.

Messages

The message type of IPC allows processes (executing programs) to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions called messages. Processes using this type of IPC can perform two operations:

- sending
- receiving

Before a message can be sent or received by a process, a process must have the operating system generate the necessary software mechanisms to handle these operations. A process does this by using the **msgget**(2) system call. While doing this, the process becomes the owner/creator of the message facility and specifies the initial operation permissions for all other processes, including itself. Subsequently, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl**(2) system call. However, the creator remains the creator as long as the facility exists. Other processes with permission can use **msgctl**() to perform various other control functions.

Processes that have permission and are attempting to send or receive a message can suspend execution if they are unsuccessful at performing their operation. That is, a process attempting to send a message can wait until the process which is to receive the message is ready and vice versa. A process which specifies that execution is to be suspended is performing a "blocking message operation." A process which does not allow its execution to be suspended is performing a "nonblocking message operation."

A process performing a blocking message operation can be suspended until one of three conditions occurs:

- It is successful.
- It receives a signal.
- The facility is removed.

System calls make these message capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns applicable information. Otherwise, a known error code (–1) is returned to the process, and an external error number variable **errno** is set accordingly.

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier created is called the message queue identifier (**msqid**); it is used to identify or reference the associated message queue and data structure.

The message queue is used to store header information about each message that is being sent or received. This information includes the following for each message:

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

There is one associated data structure for the uniquely identified message queue. This data structure contains the following information related to the message queue:

- operation permissions data (operation permission structure)
- pointer to first message on the queue

- pointer to last message on the queue
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time

NOTE

All include files discussed in this chapter are located in the **/usr/include** or **/usr/include/sys** directories.

The C Programming Language data structure definition for the message information contained in the message queue is as follows:

```
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 */
};
```


It is located in the **/usr/include/sys/msg.h** header file.

Likewise, the structure definition for the associated data structure is as follows.

```
struct msqid_ds
{
        struct ipc_perm  msg_perm;     /* operation permission struct */
        struct msg       *msg_first;   /* ptr to first message on q */
        struct msg       *msg_last;    /* ptr to last message on q */
        ushort           msg_cbytes;   /* current # bytes on q */
        ushort           msg_qnum;     /* # of messages on q */
        ushort           msg_qbytes;   /* max # of bytes on q */
        ushort           msg_lspid;    /* pid of last msgsnd */
        ushort           msg_lrpid;    /* pid of last msgrcv */
        time_t           msg_stime;    /* last msgsnd time */
        time_t           msg_rtime;    /* last msgrcv time */
        time_t           msg_ctime;    /* last change time */
};
```

It is located in the **#include <sys/msg.h>** header file also. Note that the **msg_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 9-1.

The definition of the **ipc_perm** data structure is as follows:

```
struct ipc_perm
{
        ushort uid;   /* owner's user id */
        ushort gid;   /* owner's group id */
        ushort cuid;  /* creator's user id */
        ushort cgid;  /* creator's group id */
        ushort mode;  /* access modes */
        ushort seq;   /* slot usage sequence number */
        key_t  key;   /* key */
};
```

Figure 9-1. ipc_perm Data Structure

It is located in the **#include <sys/ipc.h>** header file; it is common for all IPC facilities.

The **msgget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **msgflg** argument that it receives:

- to get a new **msqid** and create an associated message queue and data structure for it

- to return an existing **msqid** that already has an associated message queue and data structure

The task performed is determined by the value of the **key** argument passed to the **msgget**() system call. For the first task, if the **key** is not already in use for an existing **msqid**, a new **msqid** is returned with an associated message queue and data structure created for the **key**. This occurs provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **msqid** is always returned with an associated message queue and data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, for security reasons the KEY field for the **msqid** is all zeros.

For the second task, if a **msqid** exists for the **key** specified, the value of the existing **msqid** is returned. If you do not desire to have an existing **msqid** returned, a control command (IPC_EXCL) can be specified (set) in the **msgflg** argument passed to the system call. The details of using this system call are discussed in the "Using **msgget**" section of this chapter.

When performing the first task, the process that calls **msgget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator; see the "Controlling Message Queues" section in this chapter. The creator of the message queue also determines the initial operation permissions for it.

Once a uniquely identified message queue and data structure are created, message operations [**msgop**()] and message control [**msgctl**()] can be used.

Message operations, as mentioned previously, consist of sending and receiving messages. System calls are provided for each of these operations; they are **msgsnd**() and **msgrcv**(). Refer to the "Operations for Messages" section in this chapter for details of these system calls.

Message control is done by using the **msgctl**(2) system call. It permits you to control the message facility in the following ways:

- to determine the associated data structure status for a message queue identifier (**msqid**)
- to change operation permissions for a message queue
- to change the **size** (**msg_qbytes**) of the message queue for a particular **msqid**
- to remove a particular **msqid** from the operating system along with its associated message queue and data structure

Refer to the "Controlling Message Queues" section in this chapter for details of the **msgctl**() system call.

Getting Message Queues

This section gives a detailed description of using the **msgget**(2) system call along with an example program illustrating its use.

Using msgget

The synopsis found in the **msgget**(2) entry in the *Programmer's Reference Manual* is as follows:

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

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

All these include files are located in the **/usr/include/sys** directory of the operating system.

The following line in the synopsis:

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

informs you that **msgget**() is a function with two formal arguments that returns an integer type value on successful completion (**msqid**). The next two lines declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

```
key_t  key;
int msgflg;
```

The integer returned from this function on successful completion is the message queue identifier (**msqid**) that was discussed earlier.

As declared, the process calling the **msgget**() system call must supply two arguments to be passed to the formal **key** and **msgflg** arguments.

A new **msqid** with an associated message queue and data structure is provided if either of the following is true:

- **key** is equal to IPC_PRIVATE
- **key** is passed a unique hexadecimal integer, and **msgflg** ANDed with IPC_CREAT is TRUE

The value passed to the **msgflg** argument must be an integer type octal value and it will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **msgflg** argument. They are collectively referred to as "operation permissions." Table 9-1 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

TABLE 9-1. Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in the **msg.h** header file which can be used for the user (OWNER).

Control commands are predefined constants (represented by all uppercase letters). Table 9-2 contains the names of the constants that apply to the **msgget**() system

call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

TABLE 9-2. Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for **msgflg** is therefore a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
| ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
msgflg	=	0 1 4 0 0	0 000 001 100 000 000

The **msgflg** value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:


```
msqid = msgget (key, (IPC_CREAT | 0400));

msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **msgget**(2) page in the *Programmer's Reference Manual,* success or failure of this system call depends on the argument values for **key** and **msgflg** or system tunable parameters. The system call will attempt to return a new **msqid** if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0)
- Key does not already have a **msqid** associated with it, and (**msgflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
msqid = msgget (IPC_PRIVATE, msgflg);

              or

msqid = msgget ( 0 , msgflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the MSGMNI system tunable parameter always causes a failure. The MSGMNI system tunable parameter determines the maximum number of unique message queues (**msqid**'s) in the operating system.

The second condition is satisfied if the value for **key** is not already associated with a **msqid** and the bitwise ANDing of **msgflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the operating system for this facility type and that the IPC_CREAT flag is set (**msgflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

msgflg	=	**x 1 x x x**	**(x = immaterial)**
& IPC_CREAT	=	**0 1 0 0 0**	
result	=	**0 1 0 0 0**	**(not zero)**

Since the result is not zero, the flag is set or "true."

IPC_EXCL is another control command used with IPC_CREAT to exclusively have the system call fail if, and only if, a **msqid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **msqid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **msqid** is returned if the system call is successful.

Refer to the **msgget**(2) page in the *Programmer's Reference Manual* for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 9-2) is a menu-driven program which allows all possible combinations of using the **msgget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **msgget**(2) entry in the *Programmer's Reference Manual*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired **key**
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **msgflg** argument
- **msqid**—used for returning the message queue identification number for a successful system call or the error code (–1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be valid. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-51).

The system call is made next, and the result is stored at the address of the **msqid** variable (line 53).

Since the **msqid** variable now contains a valid message queue identifier or the error code (–1), it is tested to see if an error occurred (line 55). If **msqid** equals –1, a message indicates that an error resulted, and the external **errno** variable is displayed (lines 57, 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

The example program for the **msgget**(2) system call follows. It is suggested that the source program file be named **msgget.c** and that the executable file be named **msgget**. When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully but fail when executed.

```
 1    /*This is a program to illustrate
 2    **the message get, msgget(),
 3    **system call capabilities.*/

 4    #include    <stdio.h>
 5    #include    <sys/types.h>
 6    #include    <sys/ipc.h>
 7    #include    <sys/msg.h>
 8    #include    <errno.h>

 9    /*Start of main C language program*/
10    main()
11    {
12        key_t key;              /*declare as long integer*/
13        int opperm, flags;
14        int msqid, opperm_flags;
15        /*Enter the desired key*/
16        printf("Enter the desired key in hex = ");
17        scanf("%x", &key);


18        /*Enter the desired octal operation
19          permissions.*/
20        printf("\nEnter the operation\n");
21        printf("permissions in octal = ");
22        scanf("%o", &opperm);

23        /*Set the desired flags.*/
24        printf("\nEnter corresponding number to\n");
25        printf("set the desired flags:\n");
26        printf("No flags                    = 0\n");
27        printf("IPC_CREAT                   = 1\n");
28        printf("IPC_EXCL                    = 2\n");
29        printf("IPC_CREAT and IPC_EXCL      = 3\n");
30        printf("            Flags           = ");

31        /*Get the flag(s) to be set.*/
32        scanf("%d", &flags);
```

Figure 9-2. msgget() System Call Example (Sheet 1 of 2)

```
33          /*Check the values.*/
34          printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35              key, opperm, flags);

36          /*Incorporate the control fields (flags) with
37            the operation permissions*/
38          switch (flags)
39          {
40          case 0:     /*No flags are to be set.*/
41              opperm_flags = (opperm | 0);
42              break;
43          case 1:     /*Set the IPC_CREAT flag.*/
44              opperm_flags = (opperm | IPC_CREAT);
45              break;
46          case 2:     /*Set the IPC_EXCL flag.*/
47              opperm_flags = (opperm | IPC_EXCL);
48              break;
49          case 3:     /*Set the IPC_CREAT and IPC_EXCL flags.*/
50              opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
51          }

52          /*Call the msgget system call.*/
53          msqid = msgget (key, opperm_flags);

54          /*Perform the following if the call is unsuccessful.*/
55          if(msqid == -1)
56          {
57              printf ("\nThe msgget system call failed!\n");
58              printf ("The error number = %d\n", errno);
59          }

60          /*Return the msqid on successful completion.*/
61          else
62              printf ("\nThe msqid = %d\n", msqid);
63          exit(0);
64      }
```

Figure 9-2. msgget() System Call Example (Sheet 2 of 2)

Controlling Message Queues

This section gives a detailed description of using the **msgctl** system call along with an example program which allows all its capabilities to be exercised.

Using msgctl

The synopsis found in the **msgctl**(2) entry in the *Programmer's Reference Manual* is as follows.

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

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

The **msgctl**() system call requires three arguments to be passed to it, and it returns an integer value.

On successful completion, a zero value is returned; and when unsuccessful, it returns a –1.

The **msqid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget**() system call.

The **cmd** argument can be replaced by one of the following control commands (flags):

IPC_STAT — return the status information contained in the associated data structure for the specified **msqid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area.

IPC_SET — for the specified **msqid**, set the effective user and group identification, operation permissions, and the number of bytes for the message queue.

IPC_RMID — remove the specified **msqid** along with its associated message queue and data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read permission is required to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 9-3) is a menu-driven program which allows all possible combinations of using the **msgctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgctl**(2) entry in the *Programmer's Reference Manual*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

uid used to store the IPC_SET value for the effective user identification

gid used to store the IPC_SET value for the effective group identification

mode used to store the IPC_SET value for the operation permissions

bytes used to store the IPC_SET value for the number of bytes in the message queue (**msg_qbytes**)

rtrn used to store the return integer value from the system call

msqid used to store and pass the message queue identifier to the system call

command used to store the code for the desired control command so that subsequent processing can be performed on it

choice used to determine which member is to be changed for the IPC_SET control command

msqid_ds used to receive the specified message queue indentifier's data structure when an IPC_STAT control command is performed

***buf** a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set

Note that the **msqid_ds** data structure in this program (line 16) uses the data structure located in the **msg.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **msqid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17). Now that all the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid message queue identifier which is stored at the address of the **msqid** variable (lines 19, 20). This is required for every **msgctl** system call.

Then the code for the desired control command must be entered (lines 21-27), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 37, 38) and the status information returned is printed out (lines 39-46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 108, 109). If the system call is successful, a message indicates this along with the message queue identifier used (lines 111-114).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 50-52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53-59). This code is stored at the address of the choice variable (line 60). Now, depending on the member picked, the program prompts for the new value (lines 66-95). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made

(lines 96-98). Depending on success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 100-103), and the **msqid** along with its associated message queue and data structure are removed from the operating system. Note that the ***buf** pointer is not required as an argument to perform this control command, and its value can be zero or NULL. Depending on the success or failure, the program returns the same messages as for the other control commands.

The example program for the **msgctl**() system call follows. It is suggested that the source program file be named **msgctl.c** and that the executable file be named **msgctl**. When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully but will fail when executed.

```
 1    /*This is a program to illustrate
 2    **the message control, msgctl(),
 3    **system call capabilities.
 4    */

 5    /*Include necessary header files.*/
 6    #include    <stdio.h>
 7    #include    <sys/types.h>
 8    #include    <sys/ipc.h>
 9    #include    <sys/msg.h>

10    /*Start of main C language program*/
11    main()
12    {
13        extern int errno;
14        int uid, gid, mode, bytes;
15        int rtrn, msqid, command, choice;
16        struct msqid_ds msqid_ds, *buf;
17        buf = &msqid_ds;

18        /*Get the msqid, and command.*/
19        printf("Enter the msqid = ");
20        scanf("%d", &msqid);
21        printf("\nEnter the number for\n");
22        printf("the desired command:\n");
23        printf("IPC_STAT    =  1\n");
24        printf("IPC_SET     =  2\n");
25        printf("IPC_RMID    =  3\n");
26        printf("Entry       =  ");
27        scanf("%d", &command);

28        /*Check the values.*/
29        printf ("\nmsqid =%d, command = %d\n",
30            msqid, command);
```

Figure 9-3. msgctl() System Call Example (Sheet 1 of 4)

```
31          switch (command)
32          {
33          case 1:     /*Use msgctl() to duplicate
34                the data structure for
35                      msqid in the msqid_ds area pointed
36                      to by buf and then print it out.*/
37              rtrn = msgctl(msqid, IPC_STAT,
38                  buf);
39              printf ("\nThe USER ID = %d\n",
40                  buf->msg_perm.uid);
41              printf ("The GROUP ID = %d\n",
42                  buf->msg_perm.gid);
43              printf ("The operation permissions = 0%o\n",
44                  buf->msg_perm.mode);
45              printf ("The msg_qbytes = %d\n",
46                  buf->msg_qbytes);
47              break;
48          case 2:     /*Select and change the desired
49                          member(s) of the data structure.*/
50              /*Get the original data for this msqid
51                    data structure first.*/
52              rtrn = msgctl(msqid, IPC_STAT, buf);
53              printf("\nEnter the number for the\n");
54              printf("member to be changed:\n");
55              printf("msg_perm.uid    = 1\n");
56              printf("msg_perm.gid    = 2\n");
57              printf("msg_perm.mode   = 3\n");
58              printf("msg_qbytes      = 4\n");
59              printf("Entry           = ");

60              scanf("%d", &choice);
61              /*Only one choice is allowed per
62                pass as an illegal entry will
63                    cause repetitive failures until
64                msqid_ds is updated with
65                    IPC_STAT.*/
```

Figure 9-3. msgctl() System Call Example (Sheet 2 of 4)

```
66              switch(choice){
67              case 1:
68                  printf("\nEnter USER ID = ");
69                  scanf ("%d", &uid);
70                  buf->msg_perm.uid = uid;
71                  printf("\nUSER ID = %d\n",
72                      buf->msg_perm.uid);
73                  break;
74              case 2:
75                  printf("\nEnter GROUP ID = ");
76                  scanf("%d", &gid);
77                  buf->msg_perm.gid = gid;
78                  printf("\nGROUP ID = %d\n",
79                      buf->msg_perm.gid);
80                  break;
81              case 3:
82                  printf("\nEnter MODE = ");
83                  scanf("%o", &mode);
84                  buf->msg_perm.mode = mode;
85                  printf("\nMODE = 0%o\n",
86                      buf->msg_perm.mode);
87                  break;

88              case 4:
89                  printf("\nEnter msq_bytes = ");
90                  scanf("%d", &bytes);
91                  buf->msg_qbytes = bytes;
92                  printf("\nmsg_qbytes = %d\n",
93                      buf->msg_qbytes);
94                  break;
95              }
96              /*Do the change.*/
97              rtrn = msgctl(msqid, IPC_SET,
98                  buf);
99              break;
```

Figure 9-3. msgctl() System Call Example (Sheet 3 of 4)

```
100         case 3:     /*Remove the msqid along with its
101                            associated message queue
102                            and data structure.*/
103             rtrn = msgctl(msqid, IPC_RMID, NULL);
104         }
105         /*Perform the following if the call is unsuccessful.*/
106         if(rtrn == -1)
107         {
108             printf ("\nThe msgctl system call failed!\n");
109             printf ("The error number = %d\n", errno);
110         }
111         /*Return the msqid on successful completion.*/
112         else
113             printf ("\nMsgctl was successful for msqid = %d\n",
114                 msqid);
115         exit (0);
116     }
```

Figure 9-3. msgctl() System Call Example (Sheet 4 of 4)

Operations for Messages

This section gives a detailed description of using the **msgsnd**(2) and **msgrcv**(2) system calls, along with an example program which allows all their capabilities to be exercised.

Using msgop

The synopsis found in the **msgop**(2) entry in the *Programmer's Reference Manual* is as follows.

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

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Sending a Message

The **msgsnd** system call requires four arguments to be passed to it. It returns an integer value.

On successful completion, a zero value is returned; and when unsuccessful, **msgsnd**() returns a –1.

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget**() system call.

The **msgp** argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The **msgsz** argument specifies the length of the character array in the data structure pointed to by the **msgp** argument. This is the length of the message. The maximum **size** of this array is determined by the MSGMAX system tunable parameter.

The **msg_qbytes** data structure member can be lowered from MSGMNB by using the **msgctl**() IPC_SET control command, but only the super-user can raise it afterwards.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0); this would occur if the total number of bytes allowed on the specified message queue are in use (**msg_qbytes** or MSGMNB), or the total system-wide number of messages on all queues is equal to the system imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call will fail and return a –1.

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Receiving Messages

The **msgrcv**() system call requires five arguments to be passed to it, and it returns an integer value.

On successful completion, a value equal to the number of bytes received is returned and when unsuccessful it returns a –1.

The **msqid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **msgget**() system call.

The **msgp** argument is a pointer to a structure in the user memory area that will receive the message type and the message text.

The **msgsz** argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired; see the **msgflg** argument.

The **msgtyp** argument is used to pick the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the lowest type that is less than or equal to its absolute value is received.

The **msgflg** argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (**msgflg** & IPC_NOWAIT = 0); this would occur if there were not a message on the message queue of the desired type (**msgtyp**) to be received. If the IPC_NOWAIT flag is set, the system call will fail immediately when there is not a message of the desired type on the queue. Msgflg can also specify that the system call fail if the message is longer than the **size** to be received; this is done by not setting the MSG_NOERROR flag in the **msgflg** argument (**msgflg** & MSG_NOERROR = 0). If the MSG_NOERROR flag is set, the message is truncated to the length specified by the **msgsz** argument of **msgrcv**().

Further details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **msgget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 9-4) is a menu-driven program which allows all possible combinations of using the **msgsnd**() and **msgrcv**(2) system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **msgop**(2) entry in the *Programmer's Reference Manual.* Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

sndbuf used as a buffer to contain a message to be sent (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13) The **msgbuf1** structure (lines 10-13) is almost an exact duplicate of the **msgbuf** structure contained in the **msg.h** header file. The only difference is that the character array for **msgbuf1** contains the maximum message **size** (MSGMAX) for the computer where in **msgbuf** it is set to one (1) to satisfy the compiler. For this reason **msgbuf** cannot be used directly as a template for the user-written program. It is there so you can determine its members.

rcvbuf used as a buffer to receive a message (line 13); it uses the **msgbuf1** data structure as a template (lines 10-13)

***msgp** used as a pointer (line 13) to both the **sndbuf** and **rcvbuf** buffers

i used as a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the **msgsnd**() system call; it is also used as a counter to output the received message for the **msgrcv**() system call

c used to receive the input character from the **getchar**() function (line 50)

flag used to store the code of IPC_NOWAIT for the **msgsnd**() system call (line 61)

flags	used to store the code of the IPC_NOWAIT or MSG_NOERROR flags for the **msgrcv**() system call (line 117)
choice	used to store the code for sending or receiving (line 30)
rtrn	used to store the return values from all system calls
msqid	used to store and pass the desired message queue identifier for both system calls
msgsz	used to store and pass the **size** of the message to be sent or received
msgflg	used to pass the value of flag for sending or the value of flags for receiving
msgtyp	used for specifying the message type for sending, or used to pick a message type for receiving.

Note that a **msqid_ds** data structure is set up in the program (line 21) with a pointer which is initialized to point to it (line 22); this will allow the data structure members that are affected by message operations to be observed. They are observed by using the **msgctl**() (IPC_STAT) system call to get them for the program to print them out (lines 80-92 and lines 161-168).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation, and it is stored at the address of the choice variable (lines 23-30). Depending on the code, the program proceeds as in the following **msgsnd** or **msgrcv** sections.

msgsnd

When the code is to send a message, the **msgp** pointer is initialized (line 33) to the address of the send data structure, **sndbuf**. Next, a message type must be entered for the message; it is stored at the address of the variable **msgtyp** (line 42), and then (line 43) it is put into the mtype member of the data structure pointed to by **msgp**.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the mtext array of the data structure (lines 48-51). This will continue until an end of file is recognized, which for the **getchar**() function is a control-d (CTRL-D) immediately following a carriage return (<CR>). When this happens, the **size** of the message is determined by adding one to the **i** counter (lines 52, 53) as it stored the message beginning in the zero array element of mtext. Keep in mind that the message also contains the terminating characters, and the message will therefore appear to be three characters short of **msgsz**.

The message is immediately echoed from the mtext array of the **sndbuf** data structure to provide feedback (lines 54-56).

The next and final thing that must be decided is whether to set the IPC_NOWAIT flag. The program does this by requesting that a code of a 1 be entered for yes or anything else for no (lines 57-65). It is stored at the address of the flag variable. If a 1 is entered, IPC_NOWAIT is logically ORed with **msgflg**; otherwise, **msgflg** is set to zero.

The **msgsnd()** system call is performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70-72). If it is successful, the returned value (should be zero) is printed (lines 73-76).

Every time a message is successfully sent, there are three members of the associated data structure which are updated. They are described as follows:

msg_qnum represents the total number of messages on the message queue; it is incremented by one.

msg_lspid contains the Process Identification (PID) number of the last process sending a message; it is set accordingly.

msg_stime contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly.

These members are displayed after every successful message send operation (lines 79-92).

9

msgrcv

If the code specifies that a message is to be received, the program continues execution as in the following paragraphs.

The **msgp** pointer is initialized to the **rcvbuf** data structure (line 99).

Next, the message queue identifier of the message queue from which to receive the message is requested, and it is stored at the address of **msqid** (lines 100-103).

The message type is requested, and it is stored at the address of **msgtyp** (lines 104-107).

The code for the desired combination of control flags is requested next, and it is stored at the address of flags (lines 108-117). Depending on the selected combination, **msgflg** is set accordingly (lines 118-133).

Finally, the number of bytes to be received is requested, and it is stored at the address of **msgsz** (lines 134-137).

The **msgrcv()** system call is performed (line 144). If it is unsuccessful, a message and error number is displayed (lines 145-148). If successful, a message indicates so, and the number of bytes returned is displayed followed by the received message (lines 153-159).

When a message is successfully received, three members of the associated data structure are updated as follows:

msg_qnum contains the number of messages on the message queue; it is decremented by one.

msg_lrpid contains the process identification (PID) of the last process receiving a message; it is set accordingly.

msg_rtime contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly.

The example program for the **msgop()** system calls follows. It is suggested that the program be put into a source file called **msgop.c** and then into an executable file called **msgop**.

When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully but fail when executed.

```
1    /*This is a program to illustrate
2    **the message operations, msgop(),
3    **system call capabilities.
4    */

5    /*Include necessary header files.*/
6    #include    <stdio.h>
7    #include    <sys/types.h>
8    #include    <sys/ipc.h>
9    #include    <sys/msg.h>

10   struct msgbuf1 {
11       long    mtype;
12       char    mtext[8192];
13   } sndbuf, rcvbuf, *msgp;

14   /*Start of main C language program*/
15   main()
16   {
17       extern int errno;
18       int i, c, flag, flags, choice;
19       int rtrn, msqid, msgsz, msgflg;
20       long mtype, msgtyp;
21       struct msqid_ds msqid_ds, *buf;
22       buf = &msqid_ds;

23       /*Select the desired operation.*/
24       printf("Enter the corresponding\n");
25       printf("code to send or\n");
26       printf("receive a message:\n");
27       printf("Send              =  1\n");
28       printf("Receive           =  2\n");
29       printf("Entry             =  ");
30       scanf("%d", &choice);
31       if(choice == 1) /*Send a message.*/
32       {
33           msgp = &sndbuf; /*Point to user send structure.*/

34           printf("\nEnter the msqid of\n");
35           printf("the message queue to\n");
36           printf("handle the message = ");
37           scanf("%d", &msqid);
```

Figure 9-4. msgop() System Call Example (Sheet 1 of 5)

```
38          /*Set the message type.*/
39          printf("\nEnter a positive integer\n");
40          printf("message type (long) for the\n");
41          printf("message = ");
42          scanf("%d", &msgtyp);
43          msgp->mtype = msgtyp;

44          /*Enter the message to send.*/
45          printf("\nEnter a message: \n");

46          /*A control-d (^d) terminates as
47            EOF.*/

48          /*Get each character of the message
49            and put it in the mtext array.*/
50          for(i = 0; ((c = getchar()) != EOF); i++)
51              sndbuf.mtext[i] = c;

52          /*Determine the message size.*/
53          msgsz = i + 1;

54          /*Echo the message to send.*/
55          for(i = 0; i < msgsz; i++)
56              putchar(sndbuf.mtext[i]);

57          /*Set the IPC_NOWAIT flag if
58            desired.*/
59          printf("\nEnter a 1 if you want the\n");
60          printf("the IPC_NOWAIT flag set:  ");
61          scanf("%d", &flag);
62          if(flag == 1)
63              msgflg |= IPC_NOWAIT;
64          else
65              msgflg = 0;
66          /*Check the msgflg.*/
67          printf("\nmsgflg = 0%o\n", msgflg);
```

Figure 9-4. msgop() System Call Example (Sheet 2 of 5)

```
68              /*Send the message.*/
69              rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
70              if(rtrn == -1)
71              printf("\nMsgsnd failed.  Error = %d\n",
72                      errno);
73              else {
74                  /*Print the value of test which
75                        should be zero for successful.*/
76                  printf("\nValue returned = %d\n", rtrn);

77                  /*Print the size of the message
78                    sent.*/
79                  printf("\nMsgsz = %d\n", msgsz);

80                  /*Check the data structure update.*/
81                  msgctl(msqid, IPC_STAT, buf);

82                  /*Print out the affected members.*/

83                  /*Print the incremented number of
84                    messages on the queue.*/
85                  printf("\nThe msg_qnum = %d\n",
86                      buf->msg_qnum);
87                  /*Print the process id of the last sender.*/
88                  printf("The msg_lspid = %d\n",
89                      buf->msg_lspid);
90                  /*Print the last send time.*/
91                  printf("The msg_stime = %d\n",
92                      buf->msg_stime);
93              }
94          }

95          if(choice == 2)  /*Receive a message.*/
96          {
97              /*Initialize the message pointer
98                to the receive buffer.*/
99              msgp = &rcvbuf;
100             /*Specify the message queue which contains
101                    the desired message.*/
102             printf("\nEnter the msqid = ");
103             scanf("%d", &msqid);
```

Figure 9-4. msgop() System Call Example (Sheet 3 of 5)

```
104             /*Specify the specific message on the queue
105                   by using its type.*/
106             printf("\nEnter the msgtyp = ");
107             scanf("%d", &msgtyp);

108             /*Configure the control flags for the
109                   desired actions.*/
110             printf("\nEnter the corresponding code\n");
111             printf("to select the desired flags: \n");
112             printf("No flags                    =  0\n");
113             printf("MSG_NOERROR                 =  1\n");
114             printf("IPC_NOWAIT                  =  2\n");
115             printf("MSG_NOERROR and IPC_NOWAIT  =  3\n");
116             printf("                Flags       =  ");
117             scanf("%d", &flags);

118             switch(flags) {
119                 /*Set msgflg by ORing it with the appropriate
120                         flags (constants).*/
121             case 0:
122                 msgflg = 0;
123                 break;
124             case 1:
125                 msgflg |= MSG_NOERROR;
126                 break;
127             case 2:
128                 msgflg |= IPC_NOWAIT;
129                 break;
130             case 3:
131                 msgflg |= MSG_NOERROR | IPC_NOWAIT;
132                 break;
133             }

134             /*Specify the number of bytes to receive.*/
135             printf("\nEnter the number of bytes\n");
136             printf("to receive (msgsz) = ");
137             scanf("%d", &msgsz);

138             /*Check the values for the arguments.*/
139             printf("\nmsqid =%d\n", msqid);
140             printf("\nmsgtyp = %d\n", msgtyp);
141             printf("\nmsgsz = %d\n", msgsz);
142             printf("\nmsgflg = 0%o\n", msgflg);
```

Figure 9-4. msgop() System Call Example (Sheet 4 of 5)

```
143             /*Call msgrcv to receive the message.*/
144             rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);

145             if(rtrn == -1)  {
146                 printf("\nMsgrcv failed.  ");
147                 printf("Error = %d\n", errno);
148             }
149             else {
150                 printf ("\nMsgctl was successful\n");
151                 printf("for msqid = %d\n",
152                     msqid);

153                 /*Print the number of bytes received,
154                   it is equal to the return
155                   value.*/
156                 printf("Bytes received = %d\n", rtrn);

157                 /*Print the received message.*/
158                 for(i = 0; i<=rtrn; i++)
159                     putchar(rcvbuf.mtext[i]);
160             }
161             /*Check the associated data structure.*/
162             msgctl(msqid, IPC_STAT, buf);
163             /*Print the decremented number of messages.*/
164             printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
165             /*Print the process id of the last receiver.*/
166             printf("The msg_lrpid = %d\n", buf->msg_lrpid);
167             /*Print the last message receive time*/
168             printf("The msg_rtime = %d\n", buf->msg_rtime);
169         }
170     }
```

Figure 9-4. msgop() System Call Example (Sheet 5 of 5)

Semaphores

The semaphore type of IPC allows processes to communicate through the exchange of semaphore values. A semaphore is a positive integer (0 through 32,767). Since many applications require the use of more than one semaphore, the operating system can create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a limit set by the system administrator. The tunable parameter, SEMMSL has a default value of 25. Semaphore sets are created by using the **semget**(2) system call.

The process performing the **semget**(2) system call becomes the owner/creator, determines how many semaphores are in the set, and sets the operation permissions for the set, including itself. This process can subsequently relinquish ownership of the set or change the operation permissions using the **semctl**(), semaphore control, system call. The creating process always remains the creator as long as the facility exists. Other processes with permission can use **semctl**() to perform other control functions.

Provided a process has alter permission, it can manipulate the semaphore(s). Each semaphore within a set can be manipulated in two ways with the **semop**(2) system call (which is documented in the *Programmer's Reference Manual*):

- incremented
- decremented

To increment a semaphore, an integer value of the desired magnitude is passed to the **semop**(2) system call. To decrement a semaphore, a minus (–) value of the desired magnitude is passed.

The operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a "blocking semaphore operation." This ability is also available for a process which is testing for a semaphore to become zero or equal to zero; only read permission is required for this test, and it is accomplished by passing a value of zero to the **semop**(2) system call.

On the other hand, if the process is not successful and the process does not request to have its execution suspended, it is called a "nonblocking semaphore operation." In this case, the process is returned a known error code (–1), and the external **errno** variable is set accordingly.

The blocking semaphore operation allows processes to communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the operating system until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the **semop**(2), semaphore operation, system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore number in the set is one less than the total in the set.

An array of these "blocking/nonblocking operations" can be performed on a set containing more than one semaphore. When performing an array of operations, the "blocking/nonblocking operations" can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully. This requirement means that preceding changes made to semaphore values in the set must be undone when a "blocking semaphore operation" on a semaphore in the set cannot be completed successfully; no changes are made until they can all be made. For example, if a process has successfully completed three of six operations on a set of ten semaphores but is "blocked" from performing the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed. Additionally, any operation preceding or succeeding the "blocked" operation, including the blocked operation, can specify that at such time that all operations can be performed successfully, that the operation be undone. Otherwise, the operations are performed and the semaphores are changed or one "nonblocking operation" is unsuccessful and none are changed. All this is commonly referred to as being "atomically performed."

The ability to undo operations requires the operating system to maintain an array of "undo structures" corresponding to the array of semaphore operations to be performed. Each semaphore operation which is to be undone has an associated adjust variable used for undoing the operation, if necessary.

Remember, any unsuccessful "nonblocking operation" for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (–1) is returned to the process, and the external variable **errno** is set accordingly.

System calls make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable **errno** is set accordingly.

Using Semaphores

Before semaphores can be used (operated on or controlled) a uniquely identified **data structure** and **semaphore set** (array) must be created. The unique identifier is called the semaphore identifier (**semid**); it is used to identify or reference a particular data structure and semaphore set.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The number of semaphores (**nsems**) in a semaphore set is user selectable. The following members are in each structure within a semaphore set:

- semaphore text map address
- process identification (PID) performing last operation
- number of processes awaiting the semaphore value to become greater than its current value
- number of processes awaiting the semaphore value to equal zero

There is one associated data structure for the uniquely identified semaphore set. This data structure contains information related to the semaphore set as follows:

- operation permissions data (operation permissions structure)
- pointer to first semaphore in the set (array)
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The C Programming Language data structure definition for the semaphore set (array member) is as follows.

```
struct sem
{
        ushort  semval;         /* semaphore text map address */
        short   sempid;         /* pid of last operation */
        ushort  semncnt;        /* # awaiting semval > cval */
        ushort  semzcnt;        /* # awaiting semval = 0 */
};
```

It is located in the **#include <sys/sem.h>** header file.

Likewise, the structure definition for the associated semaphore data structure is as follows.

```
struct semid_ds
{
        struct ipc_perm sem_perm;   /* operation permission struct */
        struct sem      *sem_base;  /* ptr to first semaphore in set */
        ushort          sem_nsems;  /* # of semaphores in set */
        time_t          sem_otime;  /* last semop time */
        time_t          sem_ctime;  /* last change time */
};
```

It is also located in the **#include <sys/sem.h>** header file. Note that the **sem_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure was shown in Figure 9-1.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file. It is shown in the "Messages" section.

The **semget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **semflg** argument that it receives:

- to get a new **semid** and create an associated data structure and semaphore set for it
- to return an existing **semid** that already has an associated data structure and semaphore set

The task performed is determined by the value of the **key** argument passed to the **semget**(2) system call. For the first task, if the **key** is not already in use for an existing **semid**, a new **semid** is returned with an associated data structure and semaphore set created for it provided no system tunable parameter would be exceeded.

There is also a provision for specifying a **key** of value zero (0) which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **semid** is always returned with an associated data structure and semaphore set created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the KEY field for the **semid** is all zeros.

When performing the first task, the process which calls **semget()** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Semaphores" section in this chapter. The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a **semid** exists for the **key** specified, the value of the existing **semid** is returned. If it is not desired to have an existing **semid** returned,

a control command (IPC_EXCL) can be specified (set) in the **semflg** argument passed to the system call. The system call will fail if it is passed a value for the number of semaphores (**nsems**) that is greater than the number actually in the set; if you do not know how many semaphores are in the set, use 0 for **nsems**. The details of using this system call are discussed in the "Using **semget**" section of this chapter.

Once a uniquely identified semaphore set and data structure are created, semaphore operations [**semop**(2)] and semaphore control [**semctl**()] can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call is used to perform these operations. It is called **semop**(). Refer to the "Operations on Semaphores" section in this chapter for details of this system call.

Semaphore control is done by using the **semctl**(2) system call. These control operations permit you to control the semaphore facility in the following ways:

- to return the value of a semaphore
- to set the value of a semaphore
- to return the process identification (PID) of the last process performing an operation on a semaphore set
- to return the number of processes waiting for a semaphore value to become greater than its current value
- to return the number of processes waiting for a semaphore value to equal zero
- to get all semaphore values in a set and place them in an array in user memory
- to set all semaphore values in a semaphore set from an array of values in user memory
- to place all data structure member values, status, of a semaphore set into user memory area
- to change operation permissions for a semaphore set
- to remove a particular **semid** from the operating system along with its associated data structure and semaphore set

Refer to the "Controlling Semaphores" section in this chapter for details of the **semctl**(2) system call.

Getting Semaphores

This section contains a detailed description of using the **semget**(2) system call along with an example program illustrating its use.

Using semget

The synopsis found in the **semget**(2) entry in the *Programmer's Reference Manual* is as follows:

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

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

The following line in the synopsis:

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

informs you that **semget**() is a function with three formal arguments that returns an integer type value, on successful completion (**semid**). The next two lines:

```
key_t  key;
int nsems, semflg;
```

declare the types of the formal arguments. **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this system call on successful completion is the semaphore set identifier (**semid**) that was discussed above.

As declared, the process calling the **semget**() system call must supply three arguments to be passed to the formal **key**, **nsems**, and **semflg** arguments.

A new **semid** with an associated semaphore set and data structure is provided if either of the following is true:

- **key** is equal to IPC_PRIVATE
- **key** is passed a unique hexadecimal integer, and **semflg** ANDed with IPC_CREAT is TRUE

The value passed to the **semflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/alter attributes and execution modes determine the user/group/other attributes of the **semflg** argument. They are collectively referred to as "operation permissions." Table 9-3 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

TABLE 9-3. Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00006). There are constants **#define**'d in the **sem.h** header file which can be used for the user (OWNER). They are as follows:

```
SEM_A    0200    /* alter permission by owner */
SEM_R    0400    /* read permission by owner */
```

Control commands are predefined constants (represented by all uppercase letters). Table 9-4 contains the names of the constants which apply to the **semget**(2) system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

TABLE 9-4. Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for **semflg** is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
CWI ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
semflg	=	0 1 4 0 0	0 000 001 100 000 000

The **semflg** value can be easily set by using the names of the flags with the octal operation permissions value:


```
semid = semget (key, nsems, (IPC_CREAT | 0400));

semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **semget**(2) entry in the *Programmer's Reference Manual,* success or failure of this system call depends on the argument values for **key, nsems, semflg** or system tunable parameters. The system call will attempt to return a new **semid** if one of the following conditions is true:

- **key** is equal to IPC_PRIVATE (0)
- **key** does not already have a **semid** associated with it, and (**semflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
semid = semget (IPC_PRIVATE, nsems, semflg);
```

or

```
semid = semget ( 0, nsems, semflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified.

Exceeding the SEMMNI, SEMMNS, or SEMMSL system tunable parameters will always cause a failure. The SEMMNI system tunable parameter determines the maximum number of unique semaphore sets (**semid**'s) in the operating system. The SEMMNS system tunable parameter determines the maximum number of semaphores in all semaphore sets system wide. The SEMMSL system tunable parameter determines the maximum number of semaphores in each semaphore set.

The second condition is satisfied if the value for **key** is not already associated with a **semid**, and the bitwise ANDing of **semflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the operating system for this facility type and that the IPC_CREAT flag is set (**semflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:

```
   semflg = x 1 x x x    (x = immaterial)
& IPC_CREAT = 0 1 0 0 0

      result = 0 1 0 0 0    (not zero)
```

Since the result is not zero, the flag is set or "true." SEMMNI, SEMMNS, and SEMMSL apply here also, just as for condition one.

IPC_EXCL is another control command used with IPC_CREAT to exclusively have the system call fail if, and only if, a **semid** exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) **semid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new **semid** is returned if the system call is successful. Any value for **semflg** returns a new **semid** if the key equals zero (IPC_PRIVATE) and no system tunable parameters are exceeded.

Refer to the **semget**(2) manual page for specific associated data structure initialization for successful completion.

Example Program

The example program in this section (Figure 9-5) is a menu-driven program which allows all possible combinations of using the **semget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-8) by including the required header files as specified by the **semget**(2) entry in the *Programmer's Reference Manual.* Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

- **key**—used to pass the value for the desired key
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **semflg** argument
- **semid**—used for returning the semaphore set identification number for a successful system call or the error code (–1) for an unsuccessful one.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and the control command combinations (flags) which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be valid. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 36-52).

Then, the number of semaphores for the set is requested (lines 53-57), and its value is stored at the address of **nsems**.

The system call is made next, and the result is stored at the address of the **semid** variable (lines 60, 61).

Since the **semid** variable now contains a valid semaphore set identifier or the error code (–1), it is tested to see if an error occurred (line 63). If **semid** equals –1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 65, 66). Remember that the external **errno** variable is only set when a system call fails; it should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

The example program for the **semget**(2) system call follows. It is suggested that the source program file be named **semget.c** and that the executable file be named **semget**.

```
 1    /*This is a program to illustrate
 2    **the semaphore get, semget(),
 3    **system call capabilities.*/

 4    #include     <stdio.h>
 5    #include     <sys/types.h>
 6    #include     <sys/ipc.h>
 7    #include     <sys/sem.h>
 8    #include     <errno.h>

 9    /*Start of main C language program*/
10    main()
11    {
12        key_t key;      /*declare as long integer*/
13        int opperm, flags, nsems;
14        int semid, opperm_flags;

15        /*Enter the desired key*/
16        printf("\nEnter the desired key in hex = ");
17        scanf("%x", &key);

18        /*Enter the desired octal operation
19              permissions.*/
20        printf("\nEnter the operation\n");
21        printf("permissions in octal = ");
22        scanf("%o", &opperm);

23        /*Set the desired flags.*/
24        printf("\nEnter corresponding number to\n");
25        printf("set the desired flags:\n");
26        printf("No flags                    = 0\n");
27        printf("IPC_CREAT                   = 1\n");
28        printf("IPC_EXCL                    = 2\n");
29        printf("IPC_CREAT and IPC_EXCL      = 3\n");
30        printf("             Flags          = ");
31        /*Get the flags to be set.*/
32        scanf("%d", &flags);
```

Figure 9-5. semget() System Call Example (Sheet 1 of 2)

```
33          /*Error checking (debugging)*/
34          printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35              key, opperm, flags);
36          /*Incorporate the control fields (flags) with
37                the operation permissions.*/
38          switch (flags)
39          {
40          case 0:     /*No flags are to be set.*/
41              opperm_flags = (opperm | 0);
42              break;
43          case 1:     /*Set the IPC_CREAT flag.*/
44              opperm_flags = (opperm | IPC_CREAT);
45              break;
46          case 2:     /*Set the IPC_EXCL flag.*/
47              opperm_flags = (opperm | IPC_EXCL);
48              break;
49          case 3: /*Set the IPC_CREAT and IPC_EXCL
50                        flags.*/
51              opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
52          }

53          /*Get the number of semaphores for this set.*/
54          printf("\nEnter the number of\n");
55          printf("desired semaphores for\n");
56          printf("this set (25 max) = ");
57          scanf("%d", &nsems);

58          /*Check the entry.*/
59          printf("\nNsems = %d\n", nsems);

60          /*Call the semget system call.*/
61          semid = semget(key, nsems, opperm_flags);

62          /*Perform the following if the call is unsuccessful.*/
63          if(semid == -1)
64          {
65              printf("The semget system call failed!\n");
66              printf("The error number = %d\n", errno);
67          }
68          /*Return the semid on successful completion.*/
69          else
70              printf("\nThe semid = %d\n", semid);
71          exit(0);
72      }
```

Figure 9-5. semget() System Call Example (Sheet 2 of 2)

Controlling Semaphores

This section contains a detailed description of using the **semctl**(2) system call along with an example program which allows all its capabilities to be exercised.

Using semctl

The synopsis found in the **semctl**(2) entry in the *Programmer's Reference Manual* is as follows.

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

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun
{
        int val;
        struct semid_ds *bu;
        ushort array[];
} arg;
```

The **semctl**(2) system call requires four arguments to be passed to it, and it returns an integer value.

The **semid** argument must be a valid, non-negative, integer value that has already been created by using the **semget**(2) system call.

The **semnum** argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

The **cmd** argument can be replaced by one of the following control commands (flags):

- GETVAL—return the value of a single semaphore within a semaphore set
- SETVAL—set the value of a single semaphore within a semaphore set
- GETPID—return the Process Identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set
- GETNCNT—return the number of processes waiting for the value of a particular semaphore to become greater than its current value

- GETZCNT—return the number of processes waiting for the value of a particular semaphore to be equal to zero
- GETALL—return the values for all semaphores in a semaphore set
- SETALL—set all semaphore values in a semaphore set
- IPC_STAT—return the status information contained in the associated data structure for the specified **semid**, and place it in the data structure pointed to by the ***buf** pointer in the user memory area; **arg.buf** is the union member that contains the value of **buf**
- IPC_SET—for the specified semaphore set (**semid**), set the effective user/group identification and operation permissions
- IPC_RMID—remove the specified (**semid**) semaphore set along with its associated data structure.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Read/alter permission is required as applicable for the other control commands.

The **arg** argument is used to pass the system call the appropriate union member for the control command to be performed:

- **arg.val**
- **arg.buf**
- **arg.array**

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **semget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 9-6) is a menu-driven program which allows all possible combinations of using the **semctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **semctl**(2) entry in the *Programmer's Reference Manual* Note that in this program **errno** is declared as an external variable, and therefore the **errno.h** header file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. Those declared for this program and their purpose are as follows:

- **semid_ds**—used to receive the specified semaphore set identifier's data structure when an IPC_STAT control command is performed
- **c**—used to receive the input values from the **scanf**(3S) function (line 117), when performing a SETALL control command
- **i**—used as a counter to increment through the union **arg.array** when displaying the semaphore values for a GETALL (lines 97-99) control command, and when initializing the **arg.array** when performing a SETALL (lines 115-119) control command
- **length**—used as a variable to test for the number of semaphores in a set against the **i** counter variable (lines 97, 115)
- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer from the system call which depends on the control command or a –1 when unsuccessful
- **semid**—used to store and pass the semaphore set identifier to the system call
- **semnum**—used to store and pass the semaphore number to the system call
- **cmd**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member (**uid, gid, mode**) for the IPC_SET control command that is to be changed
- **arg.val**—used to pass the system call a value to set (SETVAL) or to store (GETVAL) a value returned from the system call for a single semaphore (union member)
- **arg.buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its

return values, or where the IPC_SET command gets the values to set (union member)

- **arg.array**—used to store the set of semaphore values when getting (GETALL) or initializing (SETALL) (union member).

Note that the **semid_ds** data structure in this program (line 14) uses the data structure located in the **sem.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The **arg** union (lines 18-22) serves three purposes in one. The compiler allocates enough storage to hold its largest member. The program can then use the union as any member by referencing union members as if they were regular structure members. Note that the array is declared to have 25 elements (0 through 24). This number corresponds to the maximum number of semaphores allowed per set (SEMMSL), a system tunable parameter.

The next important program aspect to observe is that although the ***buf** pointer member (**arg.buf**) of the union is declared to be a pointer to a data structure of the **semid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 24). Because of the way this program is written, the pointer does not need to be reinitialized later. If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

Now that all the required declarations have been presented for this program, this is how it works.

First, the program prompts for a valid semaphore set identifier, which is stored at the address of the **semid** variable (lines 25-27). This is required for all **semctl**(2) system calls.

Then, the code for the desired control command must be entered (lines 28-42), and the code is stored at the address of the **cmd** variable. The code is tested to determine the control command for subsequent processing.

If the GETVAL control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 49, 50). When it is entered, it is stored at the address of the **semnum** variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52-55). If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 195, 196); if the system call is unsuccessful, an error message is displayed along with the value of the external **errno** variable (lines 191-193).

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored at the address of the **semnum** variable (line 58). Next, a message prompts for the

value to which the semaphore is to be set, and it is stored as the **arg.val** member of the union (lines 59, 60). Then, the system call is performed (lines 61, 63). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the GETPID control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 64-67), and the PID of the process performing the last operation is displayed. Depending on success or failure, the program returns the same messages as for GETVAL above.

If the GETNCNT control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 68-72). When entered, it is stored at the address of the **semnum** variable (line 73). Then, the system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74-77). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the GETZCNT control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 78-81). When it is entered, it is stored at the address of the **semnum** variable (line 82). Then the system call is performed, and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83, 86). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 88-93). The length variable is set to the number of semaphores in the set (line 91). Next, the system call is made and, on success, the **arg.array** union member contains the values of the semaphore set (line 96). Now, a loop is entered which displays each element of the **arg.array** from zero to one less than the value of length (lines 97-103). The semaphores in the set are displayed on a single line, separated by a space. Depending on success or failure, the program returns the same messages as for GETVAL above.

If the SETALL control command is selected (code 7), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 106-108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop which takes values from the keyboard and initializes the **arg.array** union member to contain the desired values of the semaphore set (lines 113-119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The system call is then made (lines 120-122). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the IPC_STAT control command is selected (code 8), the system call is performed (line 127), and the status information returned is printed out (lines 128-139); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the **errno** variable is printed out (lines 191, 192).

If the IPC_SET control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this example program provides for changing only one member at a time, and the **semctl**(2) system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored at the address of the choice variable (line 154). Now, depending on the member picked, the program prompts for the new value (lines 155-178). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending on success or failure, the program returns the same messages as for GETVAL above.

If the IPC_RMID control command (code 10) is selected, the system call is performed (lines 183-185). The **semid** along with its associated data structure and semaphore set is removed from the operating system. Depending on success or failure, the program returns the same messages as for the other control commands.

The example program for the **semctl**(2) system call follows. It is suggested that the source program file be named **semctl.c** and that the executable file be named **semctl**.

```
1    /*This is a program to illustrate
2    **the semaphore control, semctl(),
3    **system call capabilities.
4    */

5    /*Include necessary header files.*/
6    #include    <stdio.h>
7    #include    <sys/types.h>
8    #include    <sys/ipc.h>
9    #include    <sys/sem.h>

10   /*Start of main C language program*/
11   main()
12   {
13       extern int errno;
14       struct semid_ds semid_ds;
15       int c, i, length;
16       int uid, gid, mode;
17       int retrn, semid, semnum, cmd, choice;
18       union semun  {
19           int val;
20           struct semid_ds *buf;
21           ushort array[25];
22       } arg;

23       /*Initialize the data structure pointer.*/
24       arg.buf = &semid_ds;

25       /*Enter the semaphore ID.*/
26       printf("Enter the semid = ");
27       scanf("%d", &semid);
28       /*Choose the desired command.*/
29       printf("\nEnter the number for\n");
30       printf("the desired cmd:\n");
31       printf("GETVAL      =  1\n");
32       printf("SETVAL      =  2\n");
33       printf("GETPID      =  3\n");
34       printf("GETNCNT     =  4\n");
35       printf("GETZCNT     =  5\n");
36       printf("GETALL      =  6\n");
37       printf("SETALL      =  7\n");
38       printf("IPC_STAT    =  8\n");
39       printf("IPC_SET     =  9\n");
```

Figure 9-6. semctl() System Call Example (Sheet 1 of 5)

```
40          printf("IPC_RMID     =  10\n");
41          printf("Entry        =  ");
42          scanf("%d", &cmd);
43          /*Check entries.*/
44          printf ("\nsemid =%d, cmd = %d\n\n",
45              semid, cmd);

46          /*Set the command and do the call.*/
47          switch (cmd)
48          {

49          case 1: /*Get a specified value.*/
50              printf("\nEnter the semnum = ");
51              scanf("%d", &semnum);
52              /*Do the system call.*/
53              retrn = semctl(semid, semnum, GETVAL, 0);
54              printf("\nThe semval = %d\n", retrn);
55              break;
56          case 2: /*Set a specified value.*/
57              printf("\nEnter the semnum = ");
58              scanf("%d", &semnum);
59              printf("\nEnter the value = ");
60              scanf("%d", &arg.val);
61              /*Do the system call.*/
62              retrn = semctl(semid, semnum, SETVAL, arg.val);
63              break;
64          case 3: /*Get the process ID.*/
65              retrn = semctl(semid, 0, GETPID, 0);
66              printf("\nThe sempid = %d\n", retrn);
67              break;
68          case 4: /*Get the number of processes
69                    waiting for the semaphore to
70                    become greater than its current
71                             value.*/
72              printf("\nEnter the semnum = ");
73              scanf("%d", &semnum);
74              /*Do the system call.*/
75              retrn = semctl(semid, semnum, GETNCNT, 0);
76              printf("\nThe semncnt = %d", retrn);
77              break;
```

Figure 9-6. semctl() System Call Example (Sheet 2 of 5)

```
78          case 5: /*Get the number of processes
79                    waiting for the semaphore
80                              value to become zero.*/
81              printf("\nEnter the semnum = ");
82              scanf("%d", &semnum);
83              /*Do the system call.*/
84              retrn = semctl(semid, semnum, GETZCNT, 0);
85              printf("\nThe semzcnt = %d", retrn);
86              break;

87          case 6: /*Get all the semaphores.*/
88              /*Get the number of semaphores in
89                the semaphore set.*/
90              retrn = semctl(semid, 0, IPC_STAT, arg.buf);
91              length = arg.buf->sem_nsems;
92              if(retrn == -1)
93                  goto ERROR;
94              /*Get and print all semaphores in the
95                specified set.*/
96              retrn = semctl(semid, 0, GETALL, arg.array);
97              for (i = 0; i < length; i++)
98              {
99                  printf("%d", arg.array[i]);
100                 /*Separate each
101                   semaphore.*/
102                 printf("%c", ' ');
103             }
104             break;

105         case 7: /*Set all semaphores in the set.*/
106             /*Get the number of semaphores in
107               the set.*/
108             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
109             length = arg.buf->sem_nsems;
110             printf("Length = %d\n", length);
111             if(retrn == -1)
112                 goto ERROR;
113             /*Set the semaphore set values.*/
114             printf("\nEnter each value:\n");
115             for(i = 0; i < length ; i++)
116             {
117                 scanf("%d", &c);
118                 arg.array[i] = c;
119             }
```

Figure 9-6. semctl() System Call Example (Sheet 3 of 5)

```
120             /*Do the system call.*/
121             retrn = semctl(semid, 0, SETALL, arg.array);
122             break;
123         case 8: /*Get the status for the semaphore set.*/
125             /*Get and print the current status values.*/
127             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
128             printf ("\nThe USER ID = %d\n",
129                 arg.buf->sem_perm.uid);
130             printf ("The GROUP ID = %d\n",
131                 arg.buf->sem_perm.gid);
132             printf ("The operation permissions = 0%o\n",
133                 arg.buf->sem_perm.mode);
134             printf ("The number of semaphores in set = %d\n",
135                 arg.buf->sem_nsems);
136             printf ("The last semop time = %d\n",
137                 arg.buf->sem_otime);

138             printf ("The last change time  = %d\n",
139                 arg.buf->sem_ctime);
140             break;

141         case 9:     /*Select and change the desired
142                        member of the data structure.*/
143             /*Get the current status values.*/
144             retrn = semctl(semid, 0, IPC_STAT, arg.buf);
145             if(retrn == -1)
146                 goto ERROR;
147             /*Select the member to change.*/
148             printf("\nEnter the number for the\n");
149             printf("member to be changed:\n");
150             printf("sem_perm.uid    = 1\n");
151             printf("sem_perm.gid    = 2\n");
152             printf("sem_perm.mode   = 3\n");
153             printf("Entry           = ");
154             scanf("%d", &choice);
155             switch(choice){

156             case 1: /*Change the user ID.*/
157                 printf("\nEnter USER ID = ");
158                 scanf ("%d", &uid);
159                 arg.buf->sem_perm.uid = uid;
160                 printf("\nUSER ID = %d\n",
161                     arg.buf->sem_perm.uid);
162                 break;
```

Figure 9-6. semctl() System Call Example (Sheet 4 of 5)

```
163             case 2: /*Change the group ID.*/
164                 printf("\nEnter GROUP ID = ");
165                 scanf("%d", &gid);
166                 arg.buf->sem_perm.gid = gid;
167                 printf("\nGROUP ID = %d\n",
168                     arg.buf->sem_perm.gid);
169                 break;

170             case 3: /*Change the mode portion of
171                   the operation
172                               permissions.*/
173                 printf("\nEnter MODE = ");
174                 scanf("%o", &mode);
175                 arg.buf->sem_perm.mode = mode;
176                 printf("\nMODE = 0%o\n",
177                     arg.buf->sem_perm.mode);
178                 break;
179             }
180             /*Do the change.*/
181             retrn = semctl(semid, 0, IPC_SET, arg.buf);
182             break;
183         case 10:    /*Remove the semid along with its
184                       data structure.*/
185             retrn = semctl(semid, 0, IPC_RMID, 0);
186         }
187         /*Perform the following if the call is unsuccessful.*/
188         if(retrn == -1)
189         {
190     ERROR:
191             printf ("\n\nThe semctl system call failed!\n");
192             printf ("The error number = %d\n", errno);
193             exit(0);
194         }
195         printf ("\n\nThe semctl system call was successful\n");
196         printf ("for semid = %d\n", semid);
197         exit (0);
198     }
```

Figure 9-6. semctl() System Call Example (Sheet 5 of 5)

Operations on Semaphores

This section contains a detailed description of using the **semop**(2) system call along with an example program which allows all its capabilities to be exercised.

Using semop

The synopsis found in the **semop**(2) entry in the *Programmer's Reference Manual* is as follows:

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

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

The **semop**(2) system call requires three arguments to be passed to it, and it returns an integer value.

On successful completion, a zero value is returned and when unsuccessful it returns a –1.

The **semid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **semget**(2) system call.

The **sops** argument is a pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- the semaphore number
- the operation to be performed
- the control command (flags)

The ***sops** declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. Sem**buf** is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the **#include <sys/sem.h>** header file.

The **nsops** argument specifies the length of the array (the number of structures in the array). The maximum **size** of this array is determined by the SEMOPM system tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each **semop**(2) system call.

The semaphore number determines the particular semaphore within the set on which the operation is to be performed.

The operation to be performed is determined by the following:

- a positive integer value means to increment the semaphore value by its value
- a negative integer value means to decrement the semaphore value by its value
- a value of zero means to test if the semaphore is equal to zero

The following operation commands (flags) can be used:

- IPC_NOWAIT—this operation command can be set for any operations in the array. The system call will return unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The system call 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 operation command allows any operations in the array to be undone when any operation in the array is unsuccessful and does not have the IPC_NOWAIT flag set. That is, the blocked operation waits until it can perform its operation; and when it and all succeeding operations are successful, all operations with the SEM_UNDO flag set are undone. Remember, no operations are performed on any semaphores in a set until all operations are successful. Undoing is accomplished by using an array of adjust values for the operations that are to be undone when the blocked operation and all subsequent operations are successful.

Example Program

The example program in this section (Figure 9-7) is a menu-driven program which allows all possible combinations of using the **semop**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop**(2) entry in the *Programmer's Reference Manual* Note that in this program **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since the declarations are local

to the program. The variables declared for this program and their purpose are as follows:

- **sembuf**[10]—used as an array buffer (line 14) to contain a maximum of ten **sembuf** type structures; ten equals SEMOPM, the maximum number of operations on a semaphore set for each **semop**(2) system call
- ***sops**—used as a pointer (line 14) to **sembuf**[10] for the system call and for accessing the structure members within the array
- **rtrn**—used to store the return values from the system call
- **flags**—used to store the code of the IPC_NOWAIT or SEM_UNDO flags for the **semop**(2) system call (line 60)
- **i**—used as a counter (line 32) for initializing the structure members in the array, and used to print out each structure in the array (line 79)
- **nsops**—used to specify the number of semaphore operations for the system call—must be less than or equal to SEMOPM
- **semid**—used to store the desired semaphore set identifier for the system call

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19-22). Semid is stored at the address of the **semid** variable (line 23).

A message is displayed requesting the number of operations to be performed on this set (lines 25-27). The number of operations is stored at the address of the **nsops** variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30-77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (**nsops**) to be performed for the system call, so **nsops** is tested against the **i** counter for loop control. Note that **sops** is used as a pointer to each element (structure) in the array, and **sops** is incremented just like **i**. **sops** is then used to point to each member in the structure for setting them.

After the array is initialized, all its elements are printed out for feedback (lines 78-85).

The **sops** pointer is set to the address of the array (lines 86, 87). Sem**buf** could be used directly, if desired, instead of **sops** in the system call.

The system call is made (line 89), and depending on success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using the **semctl**() GETALL control command.

The example program for the **semop**(2) system call follows. It is suggested that the source program file be named **semop.c** and that the executable file be named **semop**.

```
/*This is a program to illustrate
**the semaphore operations, semop(),
**system call capabilities.
*/

/*Include necessary header files.*/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
/*Start of main C language program*/
main()
{
    extern int errno;
    struct sembuf sembuf[10], *sops;
    char string[];
    int retrn, flags, sem_num, i, semid;
    unsigned nsops;
    sops = sembuf; /*Pointer to array sembuf.*/

    /*Enter the semaphore ID.*/
    printf("\nEnter the semid of\n");
    printf("the semaphore set to\n");
    printf("be operated on = ");
    scanf("%d", &semid);
    printf("\nsemid = %d", semid);

    /*Enter the number of operations.*/
    printf("\nEnter the number of semaphore\n");
    printf("operations for this set = ");
    scanf("%d", &nsops);
    printf("\nnosops = %d", nsops);

    /*Initialize the array for the
      number of operations to be performed.*/
    for(i = 0; i < nsops; i++, sops++)
    {
```

Figure 9-7. semop(2) System Call Example (Sheet 1 of 3)

```
34              /*This determines the semaphore in
35                the semaphore set.*/
36              printf("\nEnter the semaphore\n");
37              printf("number (sem_num) = ");
38              scanf("%d", &sem_num);
39              sops->sem_num = sem_num;
40              printf("\nThe sem_num = %d", sops->sem_num);
41              /*Enter a (-)number to decrement,
42                an unsigned number (no +) to increment,
43                or zero to test for zero.  These values
44                are entered into a string and converted
45                to integer values.*/
46              printf("\nEnter the operation for\n");
47              printf("the semaphore (sem_op) = ");
48              scanf("%s", string);
49              sops->sem_op = atoi(string);
50              printf("\nsem_op = %d\n", sops->sem_op);
51              /*Specify the desired flags.*/
52              printf("\nEnter the corresponding\n");
53              printf("number for the desired\n");
54              printf("flags:\n");
55              printf("No flags                    = 0\n");
56              printf("IPC_NOWAIT                  = 1\n");
57              printf("SEM_UNDO                    = 2\n");
58              printf("IPC_NOWAIT and SEM_UNDO     = 3\n");
59              printf("             Flags          = ");
60              scanf("%d", &flags);
61              switch(flags)
62              {
63              case 0:
64                  sops->sem_flg = 0;
65                  break;
66              case 1:
67                  sops->sem_flg = IPC_NOWAIT;
68                  break;
69              case 2:
70                  sops->sem_flg = SEM_UNDO;
71                  break;
72              case 3:
73                  sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
74                  break;
75              }
76              printf("\nFlags = 0%o\n", sops->sem_flg);
77          }
```

Figure 9-7. semop(2) System Call Example (Sheet 2 of 3)

```
78          /*Print out each structure in the array.*/
79          for(i = 0; i < nsops; i++)
80          {
81              printf("\nsem_num = %d\n", sembuf[i].sem_num);
82              printf("sem_op = %d\n", sembuf[i].sem_op);
83              printf("sem_flg = %o\n", sembuf[i].sem_flg);
84              printf("%c", ' ');
85          }

86          sops = sembuf; /*Reset the pointer to
87                           sembuf[0].*/

88          /*Do the semop system call.*/
89          retrn = semop(semid, sops, nsops);
90          if(retrn == -1)  {
91              printf("\nSemop failed.  ");
92              printf("Error = %d\n", errno);
93          }
94          else {
95              printf ("\nSemop was successful\n");
96              printf("for semid = %d\n", semid);

97              printf("Value returned = %d\n", retrn);
98          }
99      }
```

Figure 9-7. semop(2) System Call Example (Sheet 3 of 3)

Shared Memory

The shared memory type of IPC allows two or more processes (executing programs) to share memory and consequently the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which is memory management hardware dependent.

This sharing of memory provides the fastest means of exchanging data between processes.

A process initially creates a shared memory segment facility using the **shmget**(2) system call. On creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read-only) on attachment. If the

memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

There are two operations that can be performed on a shared memory segment:

- **shmat**(2) — shared memory attach
- **shmdt**(2) — shared memory detach

Shared memory attach allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.

Shared memory detach allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the **shmctl**(2) system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment using the **shmctl**(2) system call.

System calls, which are documented in the *Programmer's Reference Manual,* make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call either successfully or unsuccessfully performs its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (–1) is returned to the process, and the external variable **errno** is set accordingly.

Using Shared Memory

The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the operating system at any point in time.

Before sharing of memory can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (**shmid**); it is used to identify or reference the associated data structure. The data structure includes the following for each shared memory segment:

- operation permissions
- segment size

- segment descriptor
- process identification performing last operation
- process identification of creator
- current number of processes attached
- in memory number of processes attached
- last attach time
- last detach time
- last change time

The C Programming Language data structure definition for the shared memory segment data structure is located in the **/usr/include/sys/shm.h** header file. It is as follows:

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

struct shmid_ds {
    struct ipc_perm    shm_perm;      /* operation permission struct */
    int                shm_segsz;     /* segment size */
    struct region      *shm_reg;      /* ptr to region structure */
    char               pad[4];        /* for swap compatibility */
    ushort             shm_lpid;      /* pid of last shmop */
    ushort             shm_cpid;      /* pid of creator */
    ushort             shm_nattch;    /* used only for shminfo */
    ushort             shm_cnattch;   /* used only for shminfo */
    time_t             shm_atime;     /* last shmat time */
    time_t             shm_dtime;     /* last shmdt time */
    time_t             shm_ctime;     /* last change time */
};
```

Note that the **shm_perm** member of this structure uses **ipc_perm** as a template. The breakout for the operation permissions data structure is shown in Figure 9-1.

The **ipc_perm** data structure is the same for all IPC facilities, and it is located in the **#include <sys/ipc.h>** header file. It is shown in the introduction section of "Messages."

Table 9-5 shows the shared memory state information.

TABLE 9-5. Shared Memory State Information

Shared Memory States			
Lock Bit	Swap Bit	Allocated Bit	Implied State
0	0	0	Unallocated Segment
0	0	1	Incore
0	1	0	Unused
0	1	1	On Disk
1	0	1	Locked Incore
1	1	0	Unused
1	0	0	Unused
1	1	1	Unused

The implied states of Table 9-5 are as follows:

- **Unallocated Segment**—the segment associated with this segment descriptor has not been allocated for use.
- **Incore**—the shared segment associated with this descriptor has been allocated for use. Therefore, the segment does exist and is currently resident in memory.
- **On Disk**—the shared segment associated with this segment descriptor is currently resident on the

  swap device.
- **Locked Incore**—the shared segment associated with this segment descriptor is currently locked in memory and will not be a candidate for swapping until the segment is unlocked. Only the super-user may lock and unlock a shared segment.
- **Unused**—this state is currently unused and should never be encountered by the normal user in shared memory handling.

The **shmget**(2) system call is used to perform two tasks when only the IPC_CREAT flag is set in the **shmflg** argument that it receives:

- to get a new **shmid** and create an associated shared memory segment data structure for it
- to return an existing **shmid** that already has an associated shared memory segment data structure

The task performed is determined by the value of the **key** argument passed to the **shmget**(2) system call. For the first task, if the **key** is not already in use for an existing **shmid**, a new **shmid** is returned with an associated shared memory segment data structure created for it provided no system tunable parameters would be exceeded.

There is also a provision for specifying a **key** of value zero which is known as the private **key** (IPC_PRIVATE = 0); when specified, a new **shmid** is always returned with an associated shared memory segment data structure created for it unless a system tunable parameter would be exceeded. When the **ipcs** command is performed, the KEY field for the **shmid** is all zeros.

For the second task, if a **shmid** exists for the **key** specified, the value of the existing **shmid** is returned. If it is not desired to have an existing **shmid** returned, a control command (IPC_EXCL) can be specified (set) in the **shmflg** argument passed to the system call. The details of using this system call are discussed in the "Using **shmget**" section of this chapter.

When performing the first task, the process that calls **shmget** becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see the "Controlling Shared Memory" section in this chapter. The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely identified shared memory segment data structure is created, shared memory segment operations [**shmop**()] and control [**shmctl**(2)] can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat**(2) and **shmdt**(2). Refer to the "Operations for Shared Memory" section in this chapter for details of these system calls.

Shared memory segment control is done by using the **shmctl**(2) system call. It permits you to control the shared memory facility in the following ways:

- to determine the associated data structure status for a shared memory segment (**shmid**)
- to change operation permissions for a shared memory segment
- to remove a particular **shmid** from the operating system along with its associated shared memory segment data structure
- to lock a shared memory segment in memory
- to unlock a shared memory segment

Refer to the "Controlling Shared Memory" section in this chapter for details of the **shmctl**(2) system call.

Getting Shared Memory Segments

This section gives a detailed description of using the **shmget**(2) system call along with an example program illustrating its use.

Using shmget

The synopsis found in the **shmget**(2) entry in the *Programmer's Reference Manual* is as follows:


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

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

All these include files are located in the **/usr/include/sys** directory of the operating system. The following line in the synopsis:

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

informs you that **shmget**(2) is a function with three formal arguments that returns an integer type value, on successful completion (**shmid**). The next two lines:

```
key_t  key;
int size, shmflg;
```

declare the types of the formal arguments. The variable **key_t** is declared by a **typedef** in the **types.h** header file to be an integer.

The integer returned from this function on successful completion is the shared memory identifier (**shmid**) that was discussed earlier.

As declared, the process calling the **shmget**(2) system call must supply three arguments to be passed to the formal **key, size,** and **shmflg** arguments.

A new **shmid** with an associated shared memory data structure is provided if either of the following is true:

- **key** is equal to IPC_PRIVATE
- **key** is passed a unique hexadecimal integer, and **shmflg** ANDed with IPC_CREAT is TRUE

The value passed to the **shmflg** argument must be an integer type octal value and will specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes and execution modes determine the user/group/other attributes of the **shmflg** argument. They are collectively referred to as "operation permissions." Table 9-6 reflects the numeric values (expressed in octal notation) for the valid operation permissions codes.

TABLE 9-6. Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others is desired, the code value would be 00406 (00400 plus 00006). There are constants located in

the **shm.h** header file which can be used for the user (OWNER). They are as follows:

```
SHM_R           0400
SHM_W           0200
```

Control commands are predefined constants (represented by all uppercase letters). Table 9-7 contains the names of the constants that apply to the **shmget**() system call along with their values. They are also referred to as flags and are defined in the **ipc.h** header file.

TABLE 9-7. Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for **shmflg** is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible. It is illustrated as follows:

		Octal Value	Binary Value
IPC_CREAT	=	0 1 0 0 0	0 000 001 000 000 000
| ORed by User	=	0 0 4 0 0	0 000 000 100 000 000
shmflg	=	0 1 4 0 0	0 000 001 100 000 000

The **shmflg** value can be easily set by using the names of the flags with the octal operation permissions value:

```
shmid = shmget (key, size, (IPC_CREAT | 0400));

shmid = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the **shmget**(2) entry in the *Programmer's Reference Manual*, success or failure of this system call depends on the argument values for **key**, **size**, and

shmflg or system tunable parameters. The system call will attempt to return a new **shmid** if one of the following conditions is true:

- Key is equal to IPC_PRIVATE (0).
- Key does not already have a **shmid** associated with it, and (**shmflg** & IPC_CREAT) is "true" (not zero).

The **key** argument can be set to IPC_PRIVATE in the following ways:

```
shmid = shmget (IPC_PRIVATE, size, shmflg);
```

or

```
shmid = shmget ( 0 , size, shmflg);
```

This alone will cause the system call to be attempted because it satisfies the first condition specified. Exceeding the SHMMNI system tunable parameter always causes a failure. The SHMMNI system tunable parameter determines the maximum number of unique shared memory segments (**shmids**) in the operating system.

The second condition is satisfied if the value for **key** is not already associated with a **shmid** and the bitwise ANDing of **shmflg** and IPC_CREAT is "true" (not zero). This means that the **key** is unique (not in use) within the operating system for this facility type and that the IPC_CREAT flag is set (**shmflg** | IPC_CREAT). The bitwise ANDing (&), which is the logical way of testing if a flag is set, is illustrated as follows:


```
     shmflg = x 1 x x x   (x = immaterial)
& IPC_CREAT = 0 1 0 0 0

     result = 0 1 0 0 0   (not zero)
```

Because the result is not zero, the flag is set or "true." SHMMNI applies here also, just as for condition one.

IPC_EXCL is another control command used with IPC_CREAT to exclusively have the system call fail if, and only if, a **shmid** exists for the specified **key** provided. This is necessary to prevent the process from thinking that it has received a new (unique) **shmid** when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a unique **shmid** is returned if the system call is successful. Any value for **shmflg** returns a new **shmid** if the **key** equals zero (IPC_PRIVATE).

The system call will fail if the value for the **size** argument is less than SHMMIN or greater than SHMMAX. These tunable parameters specify the minimum and maximum shared memory segment **size**s.

Refer to the **shmget**(2) manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (Figure 9-8) is a menu-driven program which allows all possible combinations of using the **shmget**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 4-7) by including the required header files as specified by the **shmget**(2) entry in the *Programmer's Reference Manual*. Note that the **errno.h** header file is included as opposed to declaring **errno** as an external variable; either method will work.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **key**—used to pass the value for the desired **key**
- **opperm**—used to store the desired operation permissions
- **flags**—used to store the desired control commands (flags)
- **opperm_flags**—used to store the combination from the logical ORing of the **opperm** and **flags** variables; it is then used in the system call to pass the **shmflg** argument
- **shmid**—used for returning the message queue identification number for a successful system call or the error code (–1) for an unsuccessful one
- **size**—used to specify the shared memory segment size.

The program begins by prompting for a hexadecimal **key**, an octal operation permissions code, and finally for the control command combinations (flags) which are selected from a menu (lines 14-31). All possible combinations are allowed

even though they might not be valid. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored at the address of the **opperm_flags** variable (lines 35-50).

A display then prompts for the **size** of the shared memory segment, and it is stored at the address of the **size** variable (lines 51-54).

The system call is made next, and the result is stored at the address of the **shmid** variable (line 56).

Since the **shmid** variable now contains a valid message queue identifier or the error code (–1), it is tested to see if an error occurred (line 58). If **shmid** equals –1, a message indicates that an error resulted and the external **errno** variable is displayed (lines 60, 61).

If no error occurred, the returned shared memory segment identifier is displayed (line 65).

The example program for the **shmget**(2) system call follows. It is suggested that the source program file be named **shmget.c** and that the executable file be named **shmget**.

When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully, but when the program is executed it will fail.

```
1    /*This is a program to illustrate
2    **the shared memory get, shmget(),
3    **system call capabilities.*/

4    #include    <sys/types.h>
5    #include    <sys/ipc.h>
6    #include    <sys/shm.h>
7    #include    <errno.h>

8    /*Start of main C language program*/
9    main()
10   {
11       key_t key;              /*declare as long integer*/
12       int opperm, flags;
13       int shmid, size, opperm_flags;
14       /*Enter the desired key*/
15       printf("Enter the desired key in hex = ");
16       scanf("%x", &key);

17       /*Enter the desired octal operation
18         permissions.*/
19       printf("\nEnter the operation\n");
20       printf("permissions in octal = ");
21       scanf("%o", &opperm);

22       /*Set the desired flags.*/
23       printf("\nEnter corresponding number to\n");
24       printf("set the desired flags:\n");
25       printf("No flags                    = 0\n");
26       printf("IPC_CREAT                   = 1\n");
27       printf("IPC_EXCL                    = 2\n");
28       printf("IPC_CREAT and IPC_EXCL      = 3\n");
29       printf("            Flags           = ");
30       /*Get the flag(s) to be set.*/
31       scanf("%d", &flags);

32       /*Check the values.*/
33       printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
34           key, opperm, flags);
```

Figure 9-8. shmget(2) System Call Example (Sheet 1 of 2)

```
35          /*Incorporate the control fields (flags) with
36            the operation permissions*/
37          switch (flags)
38          {
39          case 0:     /*No flags are to be set.*/
40              opperm_flags = (opperm | 0);
41              break;
42          case 1:     /*Set the IPC_CREAT flag.*/
43              opperm_flags = (opperm | IPC_CREAT);
44              break;
45          case 2:     /*Set the IPC_EXCL flag.*/
46              opperm_flags = (opperm | IPC_EXCL);
47              break;
48          case 3:     /*Set the IPC_CREAT and IPC_EXCL flags.*/
49              opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
50          }

51          /*Get the size of the segment in bytes.*/
52          printf ("\nEnter the segment");
53          printf ("\nsize in bytes = ");
54          scanf ("%d", &size);

55          /*Call the shmget system call.*/
56          shmid = shmget (key, size, opperm_flags);

57          /*Perform the following if the call is unsuccessful.*/
58          if(shmid == -1)
59          {
60              printf ("\nThe shmget system call failed!\n");
61              printf ("The error number = %d\n", errno);
62          }
63          /*Return the shmid on successful completion.*/
64          else
65              printf ("\nThe shmid = %d\n", shmid);
66          exit(0);
67      }
```

Figure 9-8. shmget(2) System Call Example (Sheet 2 of 2)

Controlling Shared Memory

This section gives a detailed description of using the **shmctl**(2) system call along with an example program which allows all its capabilities to be exercised.

Using shmctl

The synopsis found in the **shmctl**(2) entry in the *Programmer's Reference Manual* is as follows:

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

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

The **shmctl**(2) system call requires three arguments to be passed to it, and **shmctl**(2) returns an integer value.

On successful completion, a zero value is returned; and when unsuccessful, **shmctl**() returns a –1.

The **shmid** variable must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget**(2) system call.

The **cmd** argument can be replaced by one of following control commands (flags):

- IPC_STAT—return the status information contained in the associated data structure for the specified **shmid** and place it in the data structure pointed to by the ***buf** pointer in the user memory area
- IPC_SET—for the specified **shmid**, set the effective user and group identification, and operation permissions
- IPC_RMID—remove the specified **shmid** along with its associated shared memory segment data structure
- SHM_LOCK—lock the specified shared memory segment in memory, must be super-user
- SHM_UNLOCK—unlock the shared memory segment from memory, must be super-user.

A process must have an effective user identification of OWNER/CREATOR or super-user to perform an IPC_SET or IPC_RMID control command. Only the

super-user can perform a SHM_LOCK or SHM_UNLOCK control command. A process must have read permission to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 9-9) is a menu-driven program that allows all possible combinations of using the **shmctl**(2) system call to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmctl**(2) entry in the *Programmer's Reference Manual*. Note in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self-explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **uid**—used to store the IPC_SET value for the effective user identification
- **gid**—used to store the IPC_SET value for the effective group identification
- **mode**—used to store the IPC_SET value for the operation permissions
- **rtrn**—used to store the return integer value from the system call
- **shmid**—used to store and pass the shared memory segment identifier to the system call
- **command**—used to store the code for the desired control command so that subsequent processing can be performed on it
- **choice**—used to determine which member for the IPC_SET control command that is to be changed
- **shmid_ds**—used to receive the specified shared memory segment identifier's data structure when an IPC_STAT control command is performed

- ***buf**—a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set.

Note that the **shmid_ds** data structure in this program (line 16) uses the data structure located in the **shm.h** header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the ***buf** pointer is declared to be a pointer to a data structure of the **shmid_ds** type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid shared memory segment identifier which is stored at the address of the **shmid** variable (lines 18-20). This is required for every **shmctl**(2) system call.

Then, the code for the desired control command must be entered (lines 21-29), and it is stored at the address of the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 39, 40) and the status information returned is printed out (lines 41-71). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out. In addition, an error message is displayed and the **errno** variable is printed out (lines 148, 149). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 151-154).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is stored at the address of the choice variable (line 99). Now, depending on the member picked, the program prompts for the new value (lines 105-127). The value is placed at the address of the appropriate member in the user memory area data structure, and the system call is made (lines 128-130). Depending on success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 132-135), and the **shmid** along with its associated message queue

and data structure are removed from the operating system. Note that the ***buf** pointer is not required as an argument to perform this control command and its value can be zero or NULL. Depending on the success or failure, the program returns the same messages as for the other control commands.

If the SHM_LOCK control command (code 4) is selected, the system call is performed (lines 137,138). Depending on the success or failure, the program returns the same messages as for the other control commands.

If the SHM_UNLOCK control command (code 5) is selected, the system call is performed (lines 140-142). Depending on the success or failure, the program returns the same messages as for the other control commands.

The example program for the **shmctl**(2) system call follows. It is suggested that the source program file be named **shmctl.c** and that the executable file be named **shmctl**.

When compiling C programs that use floating point operations, the **–f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully but will fail when executed.

```
1    /*This is a program to illustrate
2    **the shared memory control, shmctl(),
3    **system call capabilities.
4    */

5    /*Include necessary header files.*/
6    #include    <stdio.h>
7    #include    <sys/types.h>
8    #include    <sys/ipc.h>
9    #include    <sys/shm.h>

10   /*Start of main C language program*/
11   main()
12   {
13       extern int errno;
14       int uid, gid, mode;
15       int rtrn, shmid, command, choice;
16       struct shmid_ds shmid_ds, *buf;
17       buf = &shmid_ds;

18       /*Get the shmid, and command.*/
19       printf("Enter the shmid = ");
20       scanf("%d", &shmid);
21       printf("\nEnter the number for\n");
22       printf("the desired command:\n");

23       printf("IPC_STAT    =  1\n");
24       printf("IPC_SET     =  2\n");
25       printf("IPC_RMID    =  3\n");
26       printf("SHM_LOCK    =  4\n");
27       printf("SHM_UNLOCK  =  5\n");
28       printf("Entry       =  ");
29       scanf("%d", &command);

30       /*Check the values.*/
31       printf ("\nshmid =%d, command = %d\n",
32           shmid, command);
```

Figure 9-9. shmctl(2) System Call Example (Sheet 1 of 4)

```
33          switch (command)
34          {
35          case 1:     /*Use shmctl() to duplicate
36                the data structure for
37                        shmid in the shmid_ds area pointed
38                        to by buf and then print it out.*/
39              rtrn = shmctl(shmid, IPC_STAT,
40                  buf);
41              printf ("\nThe USER ID = %d\n",
42                  buf->shm_perm.uid);
43              printf ("The GROUP ID = %d\n",
44                  buf->shm_perm.gid);
45              printf ("The creator's ID = %d\n",
46                  buf->shm_perm.cuid);
47              printf ("The creator's group ID = %d\n",
48                  buf->shm_perm.cgid);
49              printf ("The operation permissions = 0%o\n",
50                  buf->shm_perm.mode);
51              printf ("The slot usage sequence\n");

52              printf ("number = 0%x\n",
53                  buf->shm_perm.seq);
54              printf ("The key= 0%x\n",
55                  buf->shm_perm.key);
56              printf ("The segment size = %d\n",
57                  buf->shm_segsz);
58              printf ("The pid of last shmop = %d\n",
59                  buf->shm_lpid);
60              printf ("The pid of creator = %d\n",
61                  buf->shm_cpid);
62              printf ("The current # attached = %d\n",
63                  buf->shm_nattch);
64              printf("The in memory # attached = %d\n",
65                  buf->shm_cnattach);
66              printf("The last shmat time = %d\n",
67                  buf->shm_atime);
68              printf("The last shmdt time = %d\n",
69                  buf->shm_dtime);
70              printf("The last change time = %d\n",
71                  buf->shm_ctime);
72              break;

                /* Lines 73 - 87 deleted */
```

Figure 9-9. shmctl(2) System Call Example (Sheet 2 of 4)

```
88          case 2:     /*Select and change the desired
89                         member(s) of the data structure.*/

90              /*Get the original data for this shmid
91                    data structure first.*/
92              rtrn = shmctl(shmid, IPC_STAT, buf);

93              printf("\nEnter the number for the\n");
94              printf("member to be changed:\n");
95              printf("shm_perm.uid    = 1\n");
96              printf("shm_perm.gid    = 2\n");
97              printf("shm_perm.mode   = 3\n");
98              printf("Entry           = ");
99              scanf("%d", &choice);
100             /*Only one choice is allowed per
101               pass as an illegal entry will
102                   cause repetitive failures until
103               shmid_ds is updated with
104                   IPC_STAT.*/

105             switch(choice){
106             case 1:
107                 printf("\nEnter USER ID = ");
108                 scanf ("%d", &uid);
109                 buf->shm_perm.uid = uid;
110                 printf("\nUSER ID = %d\n",
111                     buf->shm_perm.uid);
112                 break;

113             case 2:
114                 printf("\nEnter GROUP ID = ");
115                 scanf("%d", &gid);
116                 buf->shm_perm.gid = gid;
117                 printf("\nGROUP ID = %d\n",
118                     buf->shm_perm.gid);
119                 break;
```

Figure 9-9. shmctl(2) System Call Example (Sheet 3 of 4)

```
120             case 3:
121                 printf("\nEnter MODE = ");
122                 scanf("%o", &mode);
123                 buf->shm_perm.mode = mode;
124                 printf("\nMODE = 0%o\n",
125                     buf->shm_perm.mode);
126                 break;
127             }
128             /*Do the change.*/
129             rtrn = shmctl(shmid, IPC_SET,
130                 buf);
131             break;

132         case 3:    /*Remove the shmid along with its
133                        associated
134                        data structure.*/
135             rtrn = shmctl(shmid, IPC_RMID, NULL);
136             break;

137         case 4: /*Lock the shared memory segment*/
138             rtrn = shmctl(shmid, SHM_LOCK, NULL);
139             break;
140         case 5: /*Unlock the shared memory
141                        segment.*/
142             rtrn = shmctl(shmid, SHM_UNLOCK, NULL);
143             break;
144         }
145         /*Perform the following if the call is unsuccessful.*/
146         if(rtrn == -1)
147         {
148             printf ("\nThe shmctl system call failed!\n");
149             printf ("The error number = %d\n", errno);
150         }
151         /*Return the shmid on successful completion.*/
152         else
153             printf ("\nShmctl was successful for shmid = %d\n",
154                 shmid);
155         exit (0);
156     }
```

Figure 9-9. shmctl(2) System Call Example (Sheet 4 of 4)

Operations for Shared Memory

This section gives a detailed description of using the **shmat**(2) and **shmdt**(2) system calls, along with an example program that allows all their capabilities to be exercised.

Using shmop

The synopsis found in the **shmop**(2) entry in the *Programmer's Reference Manual* is as follows:

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

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

int shmdt (shmaddr)
char *shmaddr;
```

Attaching a Shared Memory Segment

The **shmat**(2) system call requires three arguments to be passed to it, and it returns a character pointer value.

The system call can be cast to return an integer value. On successful completion, this value will be the address in core memory where the process is attached to the shared memory segment and when unsuccessful it will be a –1.

The **shmid** argument must be a valid, non-negative, integer value. In other words, it must have already been created by using the **shmget**(2) system call.

The shm**addr** argument can be zero or user supplied when passed to the **shmat**(2) system call. If it is zero, the operating system picks the address of where the shared memory segment will be attached. If it is user supplied, the address must

be a valid address that the operating system would pick. The following illustrates some typical address ranges:

0xc00c0000
0xc00e0000
0xc0100000
0xc0120000

Note that these addresses are in chunks of 20,000 hexadecimal. To improve portability, it would be wise to let the operating system pick addresses.

The **shmflg** argument is used to pass the SHM_RND and SHM_RDONLY flags to the **shmat**() system call.

Further details are discussed in the example program for **shmop**(). If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Detaching Shared Memory Segments

The **shmdt**(2) system call requires one argument to be passed to it, and **shmdt**(2) returns an integer value.

On successful completion, zero is returned; and when unsuccessful, **shmdt**(2) returns a –1.

Further details of this system call are discussed in the example program. If you have problems understanding the logic manipulations in this program, read the "Using **shmget**" section of this chapter; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 9-10) is a menu-driven program which allows all possible combinations of using the **shmat**(2) and **shmdt**(2) system calls to be exercised.

From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the **shmop**(2) entry in the *Programmer's Reference Manual*. Note that in this program that **errno** is declared as an external variable, and therefore, the **errno.h** header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- **flags**—used to store the codes of SHM_RND or SHM_RDONLY for the **shmat**(2) system call
- **addr**—used to store the address of the shared memory segment for the **shmat**(2) and **shmdt**(2) system calls
- **i**—used as a loop counter for attaching and detaching
- **attach**—used to store the desired number of attach operations
- **shmid**—used to store and pass the desired shared memory segment identifier
- **shmflg**—used to pass the value of flags to the **shmat**(2) system call
- **retrn**—used to store the return values from both system calls
- **detach**—used to store the desired number of detach operations

This example program combines both the **shmat**(2) and **shmdt**(2) system calls. The program prompts for the number of attachments and enters a loop until they are done for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed and enters a loop until they are done for the specified shared memory segment addresses.

shmat

The program prompts for the number of attachments to be performed, and the value is stored at the address of the attach variable (lines 17-21).

A loop is entered using the attach variable and the i counter (lines 23-70) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 24-27) and it is stored at the address of the **shmid** variable (line 28). Next, the program prompts for the address where the segment is to be attached (lines 30-34), and it is stored at the address of the **addr** variable (line 35). Then, the program prompts for the desired flags to be used for the attachment (lines 37-44), and the code representing the flags is stored at the address of the flags variable (line 45). The flags variable is tested to determine the code to be stored for the **shmflg** variable used to pass them to the **shmat**(2) system call (lines 46-57). The system call is made (line 60). If successful, a message stating so is displayed

along with the attach address (lines 66-68). If unsuccessful, a message stating so is displayed and the error code is displayed (lines 62, 63). The loop then continues until it finishes.

shmdt

After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 71-75), and the value is stored at the address of the detach variable (line 76).

A loop is entered using the detach variable and the i counter (lines 78-95) to perform the specified number of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored at the address of the **addr** variable (line 84). Then, the **shmdt**(2) system call is performed (line 87). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 92,93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

The example program for the **shmop**(2) system calls follows. It is suggested that the program be put into a source file called **shmop.c** and then into an executable file called **shmop**.

When compiling C programs that use floating point operations, the **-f** option should be used on the **cc** command line. If this option is not used, the program will compile successfully but will fail when executed.

```
 1    /*This is a program to illustrate
 2    **the shared memory operations, shmop(),
 3    **system call capabilities.
 4    */

 5    /*Include necessary header files.*/
 6    #include    <stdio.h>
 7    #include    <sys/types.h>
 8    #include    <sys/ipc.h>
 9    #include    <sys/shm.h>
10    /*Start of main C language program*/
11    main()
12    {
13        extern int errno;
14        int flags, addr, i, attach;
15        int shmid, shmflg, retrn, detach;



16        /*Loop for attachments by this process.*/
17        printf("Enter the number of\n");
18        printf("attachments for this\n");
19        printf("process (1-4).\n");
20        printf("       Attachments = ");

21        scanf("%d", &attach);
22        printf("Number of attaches = %d\n", attach);

23        for(i = 1; i <= attach; i++) {
24            /*Enter the shared memory ID.*/
25            printf("\nEnter the shmid of\n");
26            printf("the shared memory segment to\n");
27            printf("be operated on = ");
28            scanf("%d", &shmid);
29            printf("\nshmid = %d\n", shmid);

30            /*Enter the value for shmaddr.*/
31            printf("\nEnter the value for\n");
32            printf("the shared memory address\n");
33            printf("in hexadecimal:\n");
34            printf("             Shmaddr = ");
35            scanf("%x", &addr);
36            printf("The desired address = 0x%x\n", addr);
```

Figure 9-10. shmop() System Call Example (Sheet 1 of 3)

```
37              /*Specify the desired flags.*/
38              printf("\nEnter the corresponding\n");
39              printf("number for the desired\n");
40              printf("flags:\n");
41              printf("SHM_RND                 = 1\n");
42              printf("SHM_RDONLY              = 2\n");
43              printf("SHM_RND and SHM_RDONLY = 3\n");
44              printf("            Flags       = ");
45              scanf("%d", &flags);

46              switch(flags)
47              {
48              case 1:
49                  shmflg = SHM_RND;
50                  break;
51              case 2:
52                  shmflg = SHM_RDONLY;
53                  break;
54              case 3:
55                  shmflg = SHM_RND | SHM_RDONLY;
56                  break;
57              }
58              printf("\nFlags = 0%o\n", shmflg);

59              /*Do the shmat system call.*/
60              retrn = (int)shmat(shmid, addr, shmflg);
61              if(retrn == -1)  {
62                  printf("\nShmat failed.  ");
63                  printf("Error = %d\n", errno);
64              }
65              else {
66                  printf ("\nShmat was successful\n");
67                  printf("for shmid = %d\n", shmid);
68                  printf("The address = 0x%x\n", retrn);
69              }
70          }

71          /*Loop for detachments by this process.*/
72          printf("Enter the number of\n");
73          printf("detachments for this\n");
74          printf("process (1-4).\n");
75          printf("       Detachments = ");

76          scanf("%d", &detach);
77          printf("Number of attaches = %d\n", detach);
78          for(i = 1; i <= detach; i++) {
```

Figure 9-10. shmop() System Call Example (Sheet 2 of 3)

```
79              /*Enter the value for shmaddr.*/
80              printf("\nEnter the value for\n");
81              printf("the shared memory address\n");
82              printf("in hexadecimal:\n");
83              printf("             Shmaddr = ");
84              scanf("%x", &addr);
85              printf("The desired address = 0x%x\n", addr);

86              /*Do the shmdt system call.*/
87              retrn = (int)shmdt(addr);
88              if(retrn == -1)  {
89                  printf("Error = %d\n", errno);
90              }
91              else {
92                  printf ("\nShmdt was successful\n");
93                  printf("for address  = 0%x\n", addr);

94              }
95          }
96      }
```

Figure 9-10. shmop() System Call Example (Sheet 3 of 3)

9