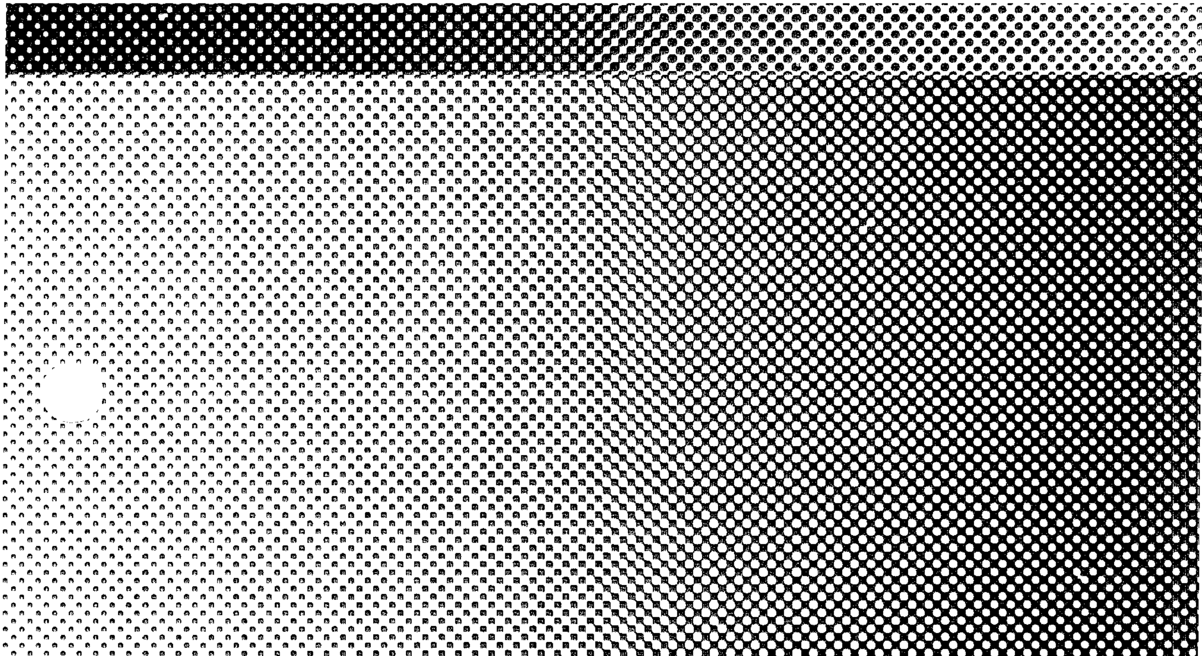
Replace this

page with the

INTER-PROCESS COMMUNICATION

tab separator.

AT&T 3B2 Computer
UNIX™ System V Release 2.0 Inter-Process Communication Utilities Guide



CONTENTS

Chapter 1. INTRODUCTION

Chapter 2. OVERVIEW OF IPC FACILITIES

Chapter 3. MESSAGES

Chapter 4. SEMAPHORES

Chapter 5. SHARED MEMORY

Chapter 6. SYSTEM TUNABLE PARAMETERS

Chapter 7. COMMAND DESCRIPTIONS

Appendix: IPC ERROR CODES

Chapter 1

INTRODUCTION

	PAGE
GENERAL	1-1
Facilities	1-1
Utilities	1-3
GUIDE ORGANIZATION	1-4

Chapter 1

INTRODUCTION

GENERAL

This guide describes the Inter-Process Communication (IPC) Facilities and Utilities available with the AT&T 3B2 Computer.

Facilities

Facilities are uniquely identifiable software mechanisms that processes (executing programs) create, control, or operate on. These software mechanisms "facilitate " or handle IPC.

There are three types of IPC facilities. These three types of IPC facilities are the heart of IPC . Each type of IPC facility allows a particular method of communication between or among cooperating processes. These methods of communication are named as follows:

- Messages
- Semaphores

- Shared memory.

Processes create, control, or operate on facilities by using system calls. Each type of IPC facility has three categories of system calls associated with it. These system calls are normally imbedded in **C** Language programs to do the following functions:

- Getting the facility
- Controlling the facility
- Operating on the facility.

There are nine IPC **UNIX*** System manual pages associated with these system calls, three manual pages for each of the three types of IPC:

msgget()	msgctl()	msgop()
semget()	semctl()	semop()
shmget()	shmctl()	shmop()

The first three letters of the IPC UNIX System manual page names represent the type of IPC: **msg** for message, **sem** for semaphore, and **shm** for shared memory. The last three letters of the names represent the action to do: **get** for getting the facility, **ctl** for controlling the facility, and **op** for operating on the facility.

These names are the system call names with two exceptions. The **msgop()** and **shmop()** UNIX System manual page names are not used to invoke the system calls. They both have two different system call names for their operations.

---

* Trademark of AT&T

For **msgop()** they are:

- **msgsnd()**, message send
- **msgrcv()**, message receive.

For **shmop()** they are:

- **shmat()**, shared memory attach
- **shmdt()**, shared memory detach.

The naming of the **msgop()** and **shmop()** UNIX System manual pages is for consistency and ease of reference.

Utilities

There are two IPC utilities (commands) that run under the UNIX System. These commands are used for the following:

- Checking the status of IPC facilities
- Removing IPC facilities.

The mnemonic names for these commands are as follows:

- **ipcs**
- **ipcrm**.

The first three letters of these commands (**ipc**) represent ''inter-process communication.'' The remaining letters denote what the command is used for: **s** stands for status, and **rm** stands for remove. These commands give you a direct interface to the IPC facilities.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

GUIDE ORGANIZATION

This guide is structured so you can easily find desired information without having to read the entire text. The remainder of this document is organized as follows:

- Chapter 2, "OVERVIEW OF IPC FACILITIES," gives an overview of each type of IPC facility. This overview allows you to understand how the types of IPC facilities work and what they can do for you.

- Chapter 3, "MESSAGES," describes the message type of IPC. The prerequisites (calling sequence) before invoking each system call and the return values for each system call are explained. A verified program listing to exercise each system call is explained.

- Chapter 4, "SEMAPHORES," describes the semaphore type of IPC. The prerequisites (calling sequence) before invoking each system call and the return values for each system call are explained. A verified program listing to exercise each system call is explained.

- Chapter 5, "SHARED MEMORY," describes the shared memory type of IPC. The prerequisites (calling sequence) before invoking each system call and the return values for each system call are explained. A verified program listing to exercise each system call is explained.

- Chapter 6, "SYSTEM TUNABLE PARAMETERS," describes the IPC system tunable parameters. The maximum or default value initially set for each tunable parameter is given. When one tunable parameter affects another parameter, the interrelationship is explained.

- Chapter 7, "COMMAND DESCRIPTIONS," contains tutorial information for using the **ipcs** and **ipcrm** utilities. The system call programs described in the MESSAGES, SEMAPHORES, and SHARED MEMORY chapters were used to develop the facilities shown in the examples.

- Appendix, "IPC ERROR CODES," explains the standard system call error numbers as they apply to IPC. They are categorized by the type of IPC and associated system calls.

Chapter 2

OVERVIEW OF IPC FACILITIES

PAGE

MESSAGES 2-2
SEMAPHORES 2-4
SHARED MEMORY 2-7

Chapter 2

OVERVIEW OF IPC FACILITIES

The UNIX System V Release 2.0 Operating System supports three types of Inter-Process Communication (IPC):

- Messages
- Semaphores
- Shared Memory.

This chapter contains a general discussion of each type of IPC. Following chapters contain detailed discussions of the associated system calls for each type of IPC. If you are unfamiliar with IPC facilities, the organization of this guide should enable you to understand and use the facilities first before using the **ipcs** and **ipcrm** utilities.

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 do two operations:

- Sending
- Receiving.

Before a message can be sent or received by a process, a process must have the UNIX System generate the necessary software mechanisms to handle these operations. A process does this by using the **msgget()** 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. Later, the owner/creator can relinquish ownership or change the operation permissions using the **msgctl()** system call. However, the creator always remains the creator as long as the facility exists. Other processes with permission can use **msgctl()** to do 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, simplistically, a process that is attempting to send a message can wait until the process that is to receive the message is ready and vice versa. A process that specifies that execution is to be suspended is performing a "blocking message operation." A process that 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. Examples of message system calls are contained in Chapter 3, "MESSAGES."

System tunable parameters that define the maximum UNIX System resources that are initially set for this type of IPC are given in Chapter 6 of this guide, "SYSTEM TUNABLE PARAMETERS." These parameters are also pointed out where they affect the usage of a system call in the "MESSAGES" chapter.

SEMAPHORES

The *semaphore* type of IPC allows processes (executing programs) 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 UNIX System is able to create sets or arrays of semaphores. A semaphore set can contain one or more semaphores up to a system tunable parameter limit, SEMMSL=25. Semaphore sets are created by using the **semget()** system call.

The process performing the **semget()** 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 later 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 do 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()** system call:

- Incremented
- Decremented.

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

The UNIX System insures 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 that is testing for a semaphore to become zero or equal to zero; only read permission is required for this test, and it is done by passing a value of zero to the **semop()** 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." A known error code (-1) is returned to the process, and the external **errno** variable is set accordingly.

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

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

> ***Note:*** 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 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 UNIX System to maintain an array of ''undo structures'' corresponding to the array of semaphore operations to be performed. Each semaphore operation that 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. The detailed usage of these system calls is contained in Chapter 4, ''SEMAPHORES.''

System tunable parameters that define the maximum UNIX System resources that are initially set for this type of IPC are given in Chapter 6 of this guide, ''SYSTEM TUNABLE PARAMETERS.'' They are also pointed out where they affect the usage of a system call in the ''SEMAPHORES'' chapter.

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 that is 3B2 Computer 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()** 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()** — shared memory attach
- **shmdt()** — 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()** system call. However,

the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can do other functions on the shared memory segment using the **shmctl()** system call.

System calls 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. The detailed usage of these system calls is contained in Chapter 5, "SHARED MEMORY."

System tunable parameters that define the maximum UNIX System resources that are initially set for this type of IPC are given in Chapter 6 of this guide, "SYSTEM TUNABLE PARAMETERS." They are also pointed out where they affect the usage of a system call in the "SHARED MEMORY" chapter.

Chapter 3

MESSAGES

PAGE

GENERAL 3-1
GETTING MESSAGE QUEUES 3-10
Using Msgget 3-10
Example Program 3-16
CONTROLLING MESSAGE QUEUES 3-21
Using Msgctl 3-21
Example Program 3-23
OPERATIONS FOR MESSAGES 3-31
Using Msgop 3-31
Example Program 3-35

Chapter 3

MESSAGES

The *message* type of Inter-Process Communication (IPC) allows processes to communicate through the exchange of data. This data is exchanged in discrete portions called messages. They are exchanged by sending or receiving; see the " OPERATIONS FOR MESSAGES " section in this chapter about sending or receiving messages.

GENERAL

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. Figure 3-1 illustrates the relationships among the msqid, 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.

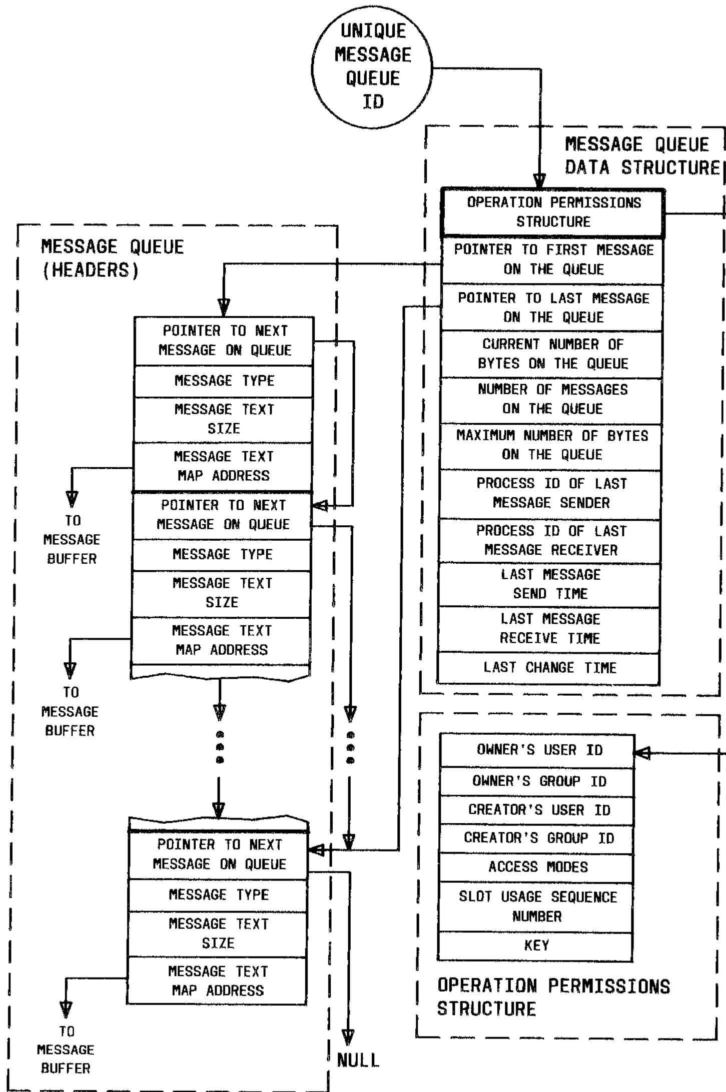


Figure 3-1. Message IPC Organization

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
- Amount 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 guide 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 */
                                      /* Times measured in sec since */
                                      /* 00:00:00 GMT, Jan. 1, 1970 */
};
```

It is located in the **#include <sys/msg.h>** header file also. Note that the **msgperm** member of this structure uses **ipc_perm** as a template. Thus, the breakout is shown in Figure 3-1 for the operation permissions data structure.

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 */
};
```

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

The **msgget()** 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 that 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, 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.

> ***Note:*** *Msgop()* is not a system call.

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()** 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 UNIX 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()** system call along with an example program illustrating its use.

Using Msgget

The synopsis of the msgget() UNIX System manual page 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 that are located in the **/usr/include/sys** directory of the UNIX 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:

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

declare the types of the formal arguments. Key_t is declared by a *typedef* in the types.h header file to be a long integer. Therefore, key and msgflg are integers (*int*) which both occupy 32-bits each in the 3B2 Computer.

The integer returned from this function on successful completion is the message queue identifier (msqid) that was discussed in the "GENERAL" section of this chapter.

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

The value passed to key must be a unique integer type hexadecimal value or zero (IPC_PRIVATE = 0) if a new msqid with an associated message queue and data structure is desired; it must be an existing key to return its msqid. This is true when only the IPC_CREAT flag is set in the msgflg argument.

Unique keys can be determined in several ways. The **STDIPC()**, standard interprocess communication package, subroutine is one method to generate unique keys to avoid undesired interference between processes. Another way could be to use the **makekey()** command. Picking a key at random is also possible but less desirable. If the key is IPC_PRIVATE, only the owner/creator process usually uses the facility.

> ***Note:*** Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this guide.

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." Figure 3-2 reflects the numeric values for the valid operation permissions codes.

OPERATION PERMISSIONS	NUMERIC 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

Figure 3-2. Operation Permissions Codes

A specific numeric value is derived by adding the numeric 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). These values are represented in octal. There are constants located in the msg.h header file that can be used for the user (OWNER). They are as follows:

```
MSG_R        0400
MSG_W        0200
```

Control commands are predefined constants (represented by all uppercase letters). Figure 3-3 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.

CONTROL COMMAND	VALUE
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 3-3. Control Commands (Flags)

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 done 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
| Read 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 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() UNIX System manual page, 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 causes a failure regardlessly. The MSGMNI system tunable parameter determines the maximum amount of unique message queues (msqid's) in the UNIX 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 UNIX 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 = don't care)
& 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." MSGMNI 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 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() UNIX System 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 is a menu driven program that allows all possible combinations of using the **msgget()** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was 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 they provide.

This program begins by including the required header files as specified by the UNIX System manual page for msgget() (lines 4-8). 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).

> ***Note:*** All possible combinations are allowed even though they might not be viable. 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() system call follows. It is suggested that the source program file be named "msgget.c" and that the executable file be named "msgget."

> ***Note:*** 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 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);

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    }
```

CONTROLLING MESSAGE QUEUES

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

Using Msgctl

The synopsis of the msgctl() UNIX System manual page 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 is a menu driven program that allows all possible combinations of using the **msgctl()** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was 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 they provide.

This program begins by including the required header files as specified by the UNIX System manual page for msgctl() (lines 5-9). 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 further processing can be performed on it
- **choice**—used to determine what 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 that 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 that 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 further 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 regardlessly; also, 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 UNIX 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."

> ***Note:*** 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 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);
```

```
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.*/

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;
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    }
```

OPERATIONS FOR MESSAGES

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

Using Msgop

The synopsis of the **msgop()** UNIX System manual page 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, and msgsnd() 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.

> ***Note:*** 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 amount of bytes allowed on the specified message queue are in use (msg_qbytes or MSGMNB), or the total system-wide amount 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 is 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 is a menu driven program that allows all possible combinations of using the **msgsnd()** and **msgrcv()** system calls to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was 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 they provide.

This program begins by including the required header files as specified by the UNIX System manual page for **msgop()** (lines 5-9). 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)

  Note: 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 3B2 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 inputing 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 inputed 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 that 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 (^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 is printed and should be zero (lines 73-76).

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

- **msg_qnum**—represents the total amount 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.

For this reason, these members are displayed after every successful message send operation (lines 79-92).

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, there are three members of the associated data structure that are updated; they are described 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."

> ***Note:*** 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 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);
```

```
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);
```

```
 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 amount 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 that contains
101                  the desired message.*/
102            printf("\nEnter the msqid = ");
103            scanf("%d", &msqid);
```

```
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);

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 amount 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    }
```

Chapter 4

SEMAPHORES

PAGE

GENERAL 4-1

GETTING SEMAPHORES 4-7

  Using Semget 4-7

  Example Program 4-12

CONTROLLING SEMAPHORES 4-17

  Using Semctl 4-17

  Example Program 4-19

OPERATIONS ON SEMAPHORES 4-30

  Using Semop 4-30

  Example Program 4-32

Chapter 4

SEMAPHORES

The *semaphore* type of Inter-Process Communication (IPC) allows processes (executing programs) to communicate through the exchange of integer values. Semaphores are created in sets of one or more and are used depending on the results of operations that are performed on them. See the " OPERATIONS ON SEMAPHORES " section of this chapter about the specific operations allowed.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this Chapter.

GENERAL

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. Figure 4-1 illustrates the relationships among the semid, data structure, and semaphore set.

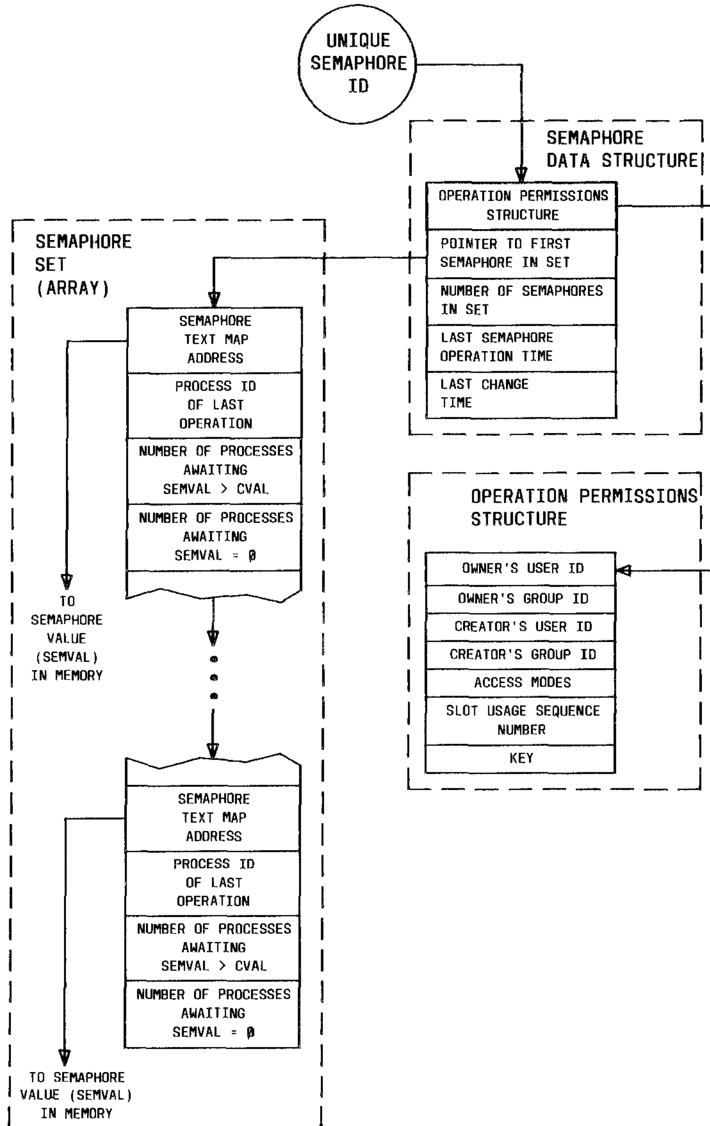


Figure 4-1. Semaphore IPC Organization

The semaphore set contains a predefined amount of structures in an array, one structure for each semaphore in the set. The amount 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
- Amount of processes awaiting the semaphore value to become greater than its current value
- Amount 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)
- Amount of semaphores in the set
- Last semaphore operation time
- Last semaphore change time.

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

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. Thus, the breakout is shown in Figure 4-1 for the operation permissions data structure.

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 " GENERAL " section of Chapter 3, " MESSAGES."

The **semget** system call is used to do 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 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) that 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 that 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**] and semaphore control [**semctl**] can be used.

Semaphore operations consist of incrementing, decrementing, and testing for zero. A single system call is used to do 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 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 Identifier (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 UNIX System along with its associated data structure and semaphore set.

Refer to the " CONTROLLING SEMAPHORES " section in this chapter for details of the semctl system call.

GETTING SEMAPHORES

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

Using Semget

The synopsis of the **semget** is as follows:

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

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

All these include files are located in the **/usr/include/sys** directory of the UNIX System.

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 a long integer. Therefore key, nsems, and semflg are integers (*int*) that occupy 32 bits each in the 3B2 Computer.

The integer returned from this system call on successful completion is the semaphore set identifier (semid) that was discussed in the " GENERAL "

section of this chapter.

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

The value passed to key must be a unique integer type hexadecimal value or zero (IPC_PRIVATE = 0) if a new semid with an associated data structure and semaphore set is desired; it must be an existing key to return its semid. This is true when only the IPC_CREAT flag is set in the semflg argument.

Unique keys can be determined in several ways. The **STDIPC**, standard inter-process communication package, subroutine is one method to generate unique keys to avoid undesired interference between processes. Another way could be to use the **makekey** command, see the manual pages for the **STDIPC** and **makekey** commands. Picking a key at random is also possible but less desirable. If the key is IPC_PRIVATE, only the owner/creator process usually uses the facility.

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." Figure 4-2 reflects the numeric values for the valid operation permissions codes.

OPERATION PERMISSIONS	NUMERIC 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

Figure 4-2. Operation Permissions Codes

A specific numeric value is derived by adding the numeric 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). These values are represented in octal. There are constants located in the sem.h header file that can be used for the user (OWNER). They are as follows:

```
SEM_R     0400
SEM_A     0200
```

Control commands are predefined constants (represented by all uppercase letters). Figure 4-3 contains the names of the constants that apply to the semget system call along with their values. They are also referred to as flags and are defined in the ipc.h header file.

CONTROL COMMAND	VALUE
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 4-3. Control Commands (Flags)

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 done by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands 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
| Read 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** success or failure of this system call depends on the *actual* argument values for key, nsems, semflg or system tunable parameters. The system call will attempt to return a new semid if a following condition 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 cause a failure regardlessly. The SEMMNI system tunable parameter determines the maximum amount of unique semaphore sets (semid's) in the UNIX System. The SEMMNS system tunable parameter determines the maximum amount of semaphores in all semaphore sets system wide. The SEMMSL system tunable parameter determines the maximum amount 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 UNIX 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 = don't care)
 & 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** manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained in the manual page.

Example Program

P The example program in this section is a menu driven program that allows all possible combinations of using the **semget** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

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

This program begins by including the required header files as specified by the manual page for semget (lines 4-8). 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) that are selected from a menu (lines 15-32).

> ***Note:*** All possible combinations are allowed even though they might not be viable. 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 shows 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 system call follows. It is suggested that the source program file be named " semget.c " and that the executable file be named " semget."

> ***Note:*** 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 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);

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    }
```

CONTROLLING SEMAPHORES

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

Using Semctl

The synopsis of the **semctl** 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 *buf;
        ushort array[];
} arg;
```

The semctl 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. In other words, it must have already been created by using the semget system call.

The semnum argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set.

> ***Note:*** 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 a following control command (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 do 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 is a menu driven program that allows all possible combinations of using the **semctl** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

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

This program begins by including the required header files as specified by the manual page for semctl (lines 5-9). 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 inputed values from the " scanf " 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 that 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 further processing can be performed on it

- **choice**—used to determine what 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 that 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 them 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 amount 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 that is stored at the address of the semid variable (lines 25-27). This is required for all semctl 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 further 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 shows 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 what 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 that 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 that 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 regardlessly; also an error message is displayed, and the **errno** variable is printed out (lines 191, 192).

If the IPC_SET control command is selected (code 8), 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 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 UNIX System. Depending on success or failure, the program returns the same messages as for the other control commands.

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

> ***Note:*** 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 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");
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;
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          }
120          /*Do the system call.*/
121          retrn = semctl(semid, 0, SETALL, arg.array);
122          break;
```

```
123      case 8: /*Get the status for the semaphore
124                    set.*/

125          /*Get the current status values and
126            print them out.*/
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;
```

```
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     }
```

OPERATIONS ON SEMAPHORES

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

Using Semop

The synopsis of the **semop** 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 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 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. Sembuf 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 system call.

The semaphore number determines the particular semaphore within the set on what 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 what 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. See " blocking/nonblocking semaphore operations " in Chapter 2.

- 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 do 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 done by using an array of adjust values for the operations that are to be undone when the blocked operation and all further operations are successful.

Example Program

The example program in this section is a menu driven program that allows all possible combinations of using the **semop** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

Since there are many ways in the **C** Programming Language to accomplish the same task or requirement, keep in mind that this example program was 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 system calls provide.

This program begins by including the required header files as specified by the manual page for **msgop** (lines 5-9). 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 amount of operations on a semaphore set for each semop 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 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 do 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). Sembuf 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 system call follows. It is suggested that the source program file be named " semop.c " and that the executable file be named " semop."

> ***Note:*** 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 semaphore operations, semop(),
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 sembuf sembuf[10], *sops;
15       char string[];
16       int retrn, flags, sem_num, i, semid;
17       unsigned nsops;
18       sops = sembuf; /*Pointer to array sembuf.*/

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

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

30       /*Initialize the array for the
31         number of operations to be performed.*/
32       for(i = 0; i < nsops; i++, sops++)
33       {

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         }
```

```
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    }
```

Chapter 5

SHARED MEMORY

PAGE

GENERAL 5-1

GETTING SHARED MEMORY SEGMENTS 5-11

- Using Shmget 5-11
- Example Program 5-16

CONTROLLING SHARED MEMORY 5-21

- Using Shmctl 5-21
- Example Program 5-22

OPERATIONS FOR SHARED MEMORY 5-30

- Using Shmop 5-30
- Example Program 5-32

Chapter 5

SHARED MEMORY

The *shared memory* type of Inter-Process Communication (IPC) allows processes (executing programs) to communicate by explicitly setting up access to a common virtual address space. 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 UNIX System at any point in time.

Refer to the *AT&T 3B2 Computer User Reference Manual* for UNIX System V manual pages supporting the commands described in this Chapter.

GENERAL

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. Figure 5-1 illustrates the relationships among the shmid, segment descriptor, and 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 amount of processes attached
- In memory the amount of processes attached
- Last attach time
- Last detach time
- Last change time.

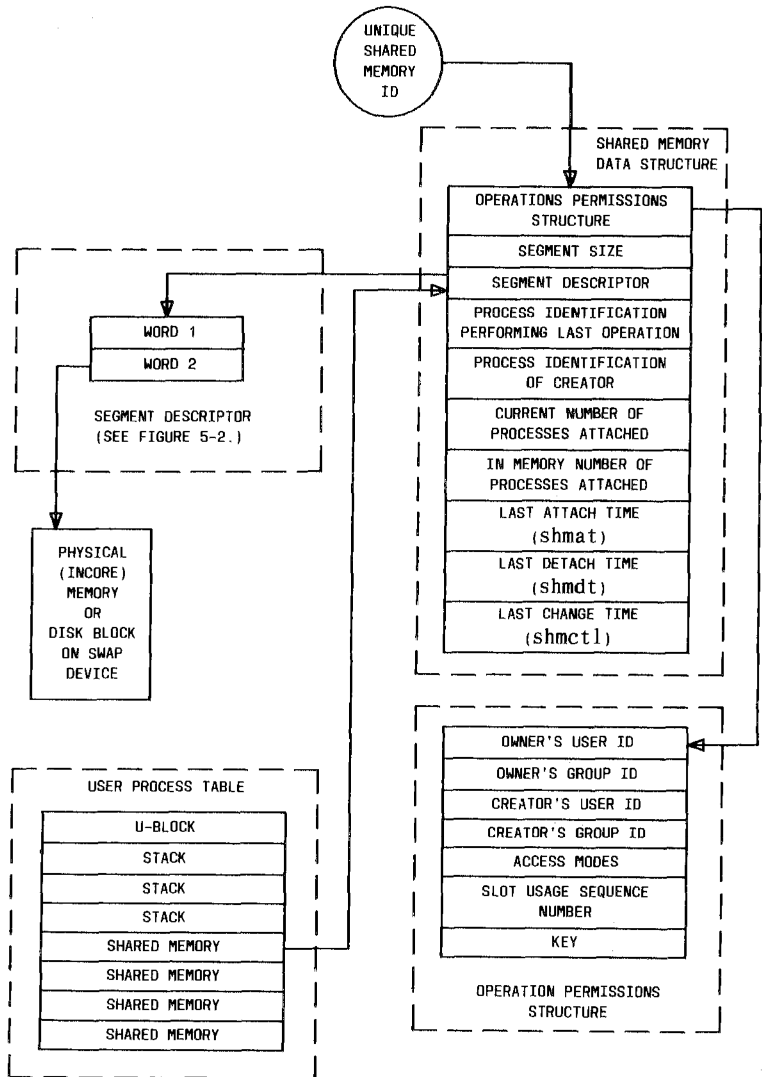


Figure 5-1. Shared Memory IPC Organization

The C Programming Language data structure definition for the shared memory segment data structure 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;       /* oper permission struct */
    int                shm_segsz;      /* segment size */
    sde_t              shm_seg;        /* segment descriptor */
    ushort             shm_lpid;       /* pid of last shmop */
    ushort             shm_cpid;       /* pid of creator */
    ushort             shm_nattch;     /* current # attached */
    ushort             shm_cnattch;    /* in memory # attached */
    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. Thus, the breakout is shown in Figure 5-1 for the operation permissions data structure.

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 "GENERAL" section of Chapter 3, "MESSAGES."

The **shm_seg** member of this data structure is defined by a **typedef** in the **/usr/include/sys/types.h** file. The definition is as follows:

```
typedef struct _SDE {                    /*  segment descriptor  */
/*                                                           */
/*  +--------+-------------+--+--------+ +--------------------------------+  */
/*  | access | maxoff  | | flags | | address                    |  */
/*  +--------+-------------+--+--------+ +--------------------------------+  */
/*       8           14        2       8                    32                */
/*                                                                            */
/*                                      +-------------------------+-+-+-+   */
/*                         (V0):        |                      |N |W |S |  */
/*                                      +-------------------------+-+-+-+   */
/*                                                   29            1 1 1    */
                                                                        */
  unsigned int access   :    8;                        /* Access rights    */
  unsigned int maxoff   :   14;                        /* Segment's max offset */
  unsigned int          :    2;                             /* Reserved          */
  unsigned int flags    :    8;                             /* Descriptor flags */
  union {
          unsigned int address;
          struct {
            unsigned int           : 29;
            unsigned int lock      :  1;               /* "N" bit  */
            unsigned int shmswap   :  1;                  /* "W" bit  */
            unsigned int alloc     :  1;               /* "S" bit  */
          } V0;
    } wd2;
} sde_t;              /* old name: SDE      */
```

Figure 5-2 represents the shared memory segment descriptor pictorially.

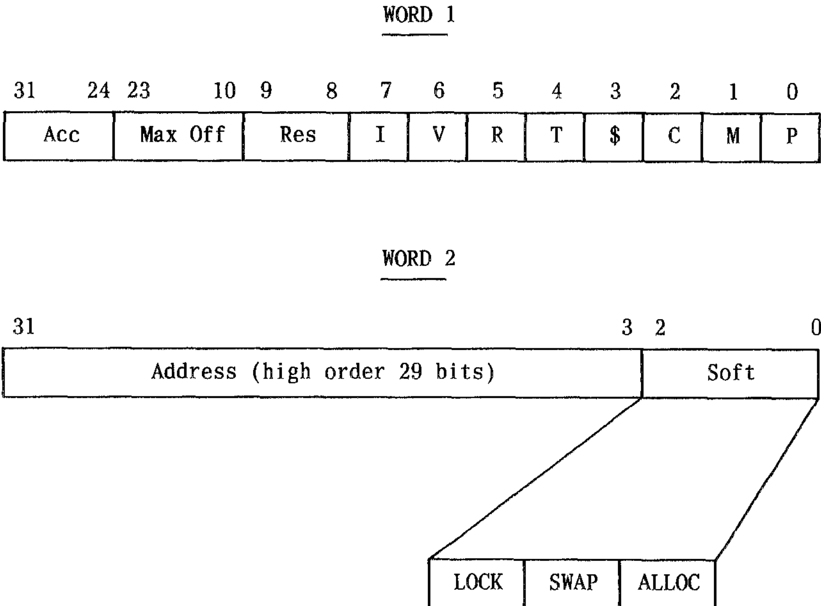


Figure 5-2. Shared Memory Segment Descriptor

Figure 5-3 is a table that shows the 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

Figure 5-3. Shared Memory State Information

The implied states of Figure 5-3 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** system call is used to do 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 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 that 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 an uniquely identified shared memory segment data structure is created, shared memory segment operations [**shmop**] and control [**shmctl**] can be used.

> ***Note:*** *Shmop* is not a system call.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat** and **shmdt**. 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 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 UNIX 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 system call.

GETTING SHARED MEMORY SEGMENTS

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

Using Shmget

The synopsis of the **shmget** 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 UNIX System.

The following line in the synopsis:

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

informs you that shmget 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. Key_t is declared by a *typedef* in the types.h header file to be a long integer. Therefore, key, size, and shmflg are integers (*int*) that occupy 32 bits each in the 3B2 Computer.

The integer returned from this function on successful completion is the shared memory identifier (shmid) that was discussed in the "GENERAL" section of this chapter.

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

The value passed to key must be a unique integer type hexadecimal value or zero (IPC_PRIVATE = 0) if a new shmid with an associated shared memory segment data structure is desired; it must be an existing key to return its shmid. This is true when only the IPC_CREAT flag is set in the shmflg argument.

Unique keys can be determined in several ways. The **STDIPC**, standard inter-process communication package, subroutine is one method to generate unique keys to avoid undesired interference between processes. Another way could be to use the **makekey** command, see the **STDIPC** and **makekey** manual pages. Picking a key at random is also possible but less desirable. If the key is IPC_PRIVATE, only the owner/creator process usually uses the facility.

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." Figure 5-4 reflects the numeric values for the valid operation permissions codes.

OPERATION PERMISSIONS	NUMERIC 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

Figure 5-4. Operation Permissions Codes

A specific numeric value is derived by adding the numeric 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). These values are represented in octal. There are constants located in the shm.h header file that 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). Figure 5-5 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.

CONTROL COMMAND	VALUE
IPC_CREAT	0001000
IPC_EXCL	0002000

Figure 5-5. Control Commands (Flags)

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 done by bitwise ORing (|) them with the operation permissions; the bit positions and values for the control commands 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
| Read 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** manual page, 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 a following condition 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 causes a failure regardlessly. The SHMMNI system tunable parameter determines the maximum amount of unique shared memory segments (shmid's) in the UNIX 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 UNIX 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 = don't care)
      & 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." 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 new 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 sizes.

Refer to the **shmget** 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 is a menu driven program that allows all possible combinations of using the **shmget** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

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

This program begins by including the required header files as specified by the manual page for shmget (lines 4-7). 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) that are selected from a menu (lines 14-31).

> ***Note:*** All possible combinations are allowed even though they might not be viable. 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 shows 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 system call follows. It is suggested that the source program file be named "shmget.c" and that the executable file be named "shmget."

> ***Note:*** 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);
```

```
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      }
```

CONTROLLING SHARED MEMORY

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

Using Shmctl

The synopsis of the **shmctl** 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 system call requires three arguments to be passed to it, and shmctl 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 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 do an IPC_SET or IPC_RMID control command. Only the super-user can do a SHM_LOCK or SHM_UNLOCK control command. A process must have read permission to do 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 is a menu driven program that allows all possible combinations of using the **shmctl** system call to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

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

This program begins by including the required header files as specified by the manual page for shmctl (lines 5-9). 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 further processing can be performed on it
- **choice**—used to determine what member for the IPC_SET control command that is to be changed
- **shmid_ds**—used to receive the specified shared memory segment indentifier's data structure when an IPC_STAT control command is performed
- ***buf**—a pointer passed to the system call that 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 that is stored at the address of the shmid variable (lines 18-20). This is required for every shmctl 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 further 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 41-86). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out regardlessly; also an error message is displayed and the **errno** variable is printed out (lines 148, 149). If the system call is successful, a message shows 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 UNIX System. Note that the *buf pointer is not required as an argument to do 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 system call follows. It is suggested that the source program file be named "shmctl.c" and that the executable file be named "shmctl."

> ***Note:*** 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 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);
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 ("Segment Descriptor:\n");
59             printf ("access = %o\n",
60                 buf->shm_seg.access);
61             printf ("maximum offset = 0%x\n",
62                 buf->shm_seg.maxoff);
63             printf ("flags = %o\n",
64                 buf->shm_seg.flags);
65             printf ("address = 0%x\n",
66                 buf->shm_seg.address);
67             printf ("lock = %o\n",
68                 buf->shm_seg.wd2.V0.lock);
69             printf ("shmswap = %o\n",
70                 buf->shm_seg.wd2.V0.shmswap);
71           printf ("alloc = %o\n",
72               buf->shm_seg.wd2.V0.alloc);
73           printf ("The pid of last shmop = %d\n",
74               buf->shm_lpid);
75           printf ("The pid of creator = %d\n",
76               buf->shm_cpid);
```

```
 77          printf ("The current # attached = %d\n",
 78              buf->shm_nattch);
 79          printf ("The in memory # attached = %d\n",
 80              buf->shm_cnattch);
 81          printf ("The last shmat time= %d\n",
 82              buf->shm_atime);
 83          printf ("The last shmdt time= %d\n",
 84              buf->shm_dtime);
 85          printf ("The last change time= %d\n",
 86              buf->shm_ctime);
 87          break;
 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;
```

```
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     }
```

OPERATIONS FOR SHARED MEMORY

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

Using Shmop

The synopsis of the **shmop** 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 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 system call.

The shmaddr argument can be zero or user supplied when passed to the shmat system call. If it is zero, the UNIX 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 UNIX System would pick.

The following illustrates some of the typical address ranges for the 3B2 Computer:

```
0xc00c0000
0xc00e0000
0xc0100000
0xc0120000
```

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

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 system call requires one argument to be passed to it, and shmdt returns an integer value.

On successful completion, zero is returned; and when unsuccessful, shmdt 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 is a menu driven program that allows all possible combinations of using the **shmat** and **shmdt** system calls to be exercised. This program was compiled and run on the 3B2 Computer; its execution has been verified.

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

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

This program begins by including the required header files as specified by the manual page for **shmop** (lines 5-9). 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 system call
- **addr**—used to store the address of the shared memory segment for the shmat and shmdt system calls
- **i**—used as a loop counter for attaching and detaching

- **attach**—used to store the desired amount 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 system call
- **retrn**—used to store the return values from both system calls
- **detach**—used to store the desired amount of detach operations.

This example program combines both the shmat and shmdt 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 do the specified amount 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 29). 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 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 do the specified amount 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 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 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."

> ***Note:*** 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 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);
```

```
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++) {
```

```
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  }
```

Chapter 6

SYSTEM TUNABLE PARAMETERS

PAGE

GENERAL 6-1

MESSAGES 6-2
- MSGMAP 6-2
- MSGMAX 6-2
- MSGMNB 6-3
- MSGMNI 6-3
- MSGSSZ 6-3
- MSGTQL 6-3
- MSGSEG 6-4

SEMAPHORES 6-5
- SEMMAP 6-5
- SEMMNI 6-6
- SEMMNS 6-6
- SEMMNU 6-6
- SEMMSL 6-6
- SEMOPM 6-6
- SEMUME 6-7
- SEMVMX 6-7
- SEMAEM 6-7

SHARED MEMORY 6-9
- SHMMAX 6-9
- SHMMIN 6-9
- SHMMNI 6-9
- SHMSEG 6-10
- SHMALL 6-10

Chapter 6

SYSTEM TUNABLE PARAMETERS

To effectively allocate the UNIX System resources to the AT&T 3B2 Computer Inter-Process Communications (IPC) facilities, system tunable parameters are used. System tunable parameters, as their name implies, can be tuned or changed to provide the most efficient UNIX System environment. However, these tunable parameters cannot be changed arbitrarily as they are interdependent. This chapter deals with the system tunable parameters for the IPC facilities.

GENERAL

System tunable parameters are initialized to their initial maximum or initial default values when the UNIX System is built. These values are contained in a directory named **/etc/master.d**. Only a process with an effective user identification of super-user (0) can change these values. The initial maximum or initial default values are given in the following sections of this chapter for each IPC facility. Additionally, information on how they interrelate is given.

MESSAGES

There are seven system tunable parameters for the *message* type facility. Each parameter and its initial value follows:

- MSGMAP-100
- MSGMAX-8192
- MSGMNB-16384
- MSGMNI-10
- MSGSSZ-8
- MSGTQL-40
- MSGSEG-1024.

The following sections describe each system tunable parameter and how they interrelate to each other.

MSGMAP

This parameter specifies the size (amount of entries) of the memory control map used to manage message segments. A warning message is sent to the **console** port if this value is insufficient to handle the message type facilities. The initial value for this parameter is a default. MSGMAP can be raised as required to accommodate the message facilities. Each map number represents eight (8) bytes.

MSGMAX

This parameter determines the maximum size of a message sent (msgsnd). The initial value is a default, but it can be raised to a maximum of 131,072 bytes (128 kilobytes) (see MSGSSZ and MSGSEG). When receiving a message (msgrcv), a value larger than this parameter can be used to insure receiving the whole message without truncation.

MSGMNB

This parameter specifies the bytes that each message queue can have for storing its message header information. The initial value is a default, and it can be tuned as desired to fit the application; each header requires 12 bytes, so keep it in multiples (regardless of the default, two raised to the fourteenth power). The OWNER of a facility can lower this value, but only the super-user can raise it afterwards (msg_qbytes).

MSGMNI

This parameter specifies the amount of message queue identifiers (msqid) system wide. MSGMNI, therefore, determines the amount of message queues that can be created (msgget) at any one time. The initial value is a default, and it can be tuned to fit the application.

MSGSSZ

This parameter determines the segment size used for storing messages in memory. Each message is stored in contiguous segments numbering enough to fit the message. The initial value of this parameter is a default. MSGSSZ can be tuned as desired to fit the application. Keep in mind that the larger the segments are, the probability of having more wasted memory at the end of each message increases. The product of this parameter and MSGSEG should be no larger than 131,072 bytes (128 kilobytes). The 128 kilobyte value is also equal to the maximum value for a single message to be sent, MSGMAX.

MSGTQL

This parameter specifies the maximum amount of message queue headers on all message queues system wide, and consequently the total amount of outstanding messages. Each header occupies 12 bytes; this relates to the length of a message queue (MSGMNB). The initial value of MSGTQL is a default. MSGTQL can be tuned to fit the application.

MSGSEG

This parameter specifies the amount of memory segments system wide for storing messages. The initial value is a default. The product of MSGSEG and MSGSSZ should be no larger than 131,072 bytes (128 kilobytes).

The following data structure is contained in the **/usr/include/sys/msg.h** header file:

```
struct msginfo {
        int     msgmap, /* # of entries in msg map */
                msgmax, /* max message size */
                msgmnb, /* max # bytes on queue */
                msgmni, /* # of message queue identifiers */
                msgssz, /* msg segment size (word size multiple) */
                msgtql; /* # of system message headers */
        ushort  msgseg; /* # of msg segments (MUST BE < 32768) */
};
```

This data structure is initialized from the **/etc/master.d/msg** file when the UNIX System is initialized.

SEMAPHORES

There are nine system tunable parameters for the *semaphore* type of IPC facility. Each parameter and its initial value follows:

- SEMMAP-10
- SEMMNI-10
- SEMMNS-60
- SEMMNU-30
- SEMMSL-25
- SEMOPM-10
- SEMUME-10
- SEMVMX-32767
- SEMAEM-16384.

The following sections describe each system tunable parameter and how they interrelate to each other.

SEMMAP

This parameter specifies the size (amount of entries) of the memory control map used to manage semaphore sets. A warning message is sent to the **console** port if this value is insufficient to handle the semaphore type facilities. The initial value for this parameter is a default. SEMMAP can be raised as required to accommodate the semaphore facilities. Each map number represents eight (8) bytes.

SEMMNI

This parameter specifies the amount of semaphore set identifiers (semid) system wide. SEMMNI, therefore, determines the amount of semaphore sets that can be created (semget) at any one time. The initial value is a default, and SEMMNI can be tuned to fit the application. This parameter occupies 32 bytes.

SEMMNS

This parameter specifies the total amount of semaphores in all semaphore sets system wide. The initial value is a default. SEMMNS can be tuned to fit the application. This parameter occupies 8 bytes.

SEMMNU

This parameter specifies the amount of semaphore undo structures system wide. The initial value is a default, and SEMMNU can be tuned to fit the application. The size of each undo structure equals [8 x (SEMUME + 2)] bytes.

SEMMSL

This parameter specifies the maximum amount of semaphores that can be in one semaphore set. The initial value is a default. SEMMSL can be tuned to fit the application.

SEMOPM

This parameter specifies the maximum amount of semaphore operations allowed for each semop() system call. The initial value is a default. SEMOPM can be tuned to fit the application. This parameter occupies 8 bytes.

SEMUME

This parameter specifies the maximum amount of undo structures per semaphore set. The initial value is a default. SEMUME can be tuned to fit the application. Keep in mind that it would be better to be able to undo as many operations as allowed per semaphore set; make SEMUME equal SEMOPM. This parameter occupies 240 bytes.

SEMVMX

This parameter specifies the maximum value that any semaphore can be. That is, the next higher number would be negative.

SEMAEM

This parameter specifies the maximum value that a semaphore adjust on exit value can be. That is, when decrementing a semaphore, this is the most that can be added to the adjust value for undoing the operation. Note that this parameter value is one more than half of SEMVMX.

The following data structure is contained in the **/usr/include/sys/sem.h** header file:

```
struct  seminfo {
        int     semmap,         /* # of entries in semaphore map */
                semmni,         /* # of semaphore identifiers */
                semmns,         /* # of semaphores in system */
                semmnu,         /* # of undo structures in system */
                semmsl,         /* max # of semaphores per id */
                semopm,         /* max # of operations per semop call */
                semume,         /* max # of undo entries per process */
                semusz,         /* size in bytes of undo structure */
                semvmx,         /* semaphore maximum value */
                semaem;         /* adjust on exit max value */
};
```

This data structure is initialized from the **/etc/master.d/sem** file when the UNIX System is initialized.

Note that the **semusz** member is not listed in the /etc/master.d/sem file as it will vary depending on the semaphore facility use.

SHARED MEMORY

There are five system tunable parameters for the *shared memory* type of IPC facility. Each parameter and its initial value follows:

- SHMMAX-8192
- SHMMIN-1
- SHMMNI-8
- SHMSEG-4
- SHMALL-32

The following sections describe each system tunable parameter and how they interrelate to each other.

SHMMAX

This parameter specifies the maximum amount of bytes that can be in a shared memory segment.

SHMMIN

This parameter specifies the minimum amount of bytes that a shared memory segment can be.

SHMMNI

This parameter specifies the total amount of shared memory facilities that can be in the UNIX System at one time. It corresponds to the amount of unique identifiers (shmid) that can be generated.

SHMSEG

This parameter specifies the maximum amount of shared memory segments that any one process can attach itself to at any one time. The default value is 4. Its maximum value is 15.

SHMALL

This parameter specifies the total amount of assigned physical pages of memory that can be in the UNIX System at one time. A page of memory equals 2048 bytes.

The following data structure is contained in the **/usr/include/sys/shm.h** header file:

```
struct  shminfo {
        int     shmmax,   /* max shared memory segment size */
                shmmin,   /* min shared memory segment size */
                shmmni,   /* # of shared memory identifiers */
                shmseg;   /* max attached shared memory segments per process */
        int     shmall;   /* maximum physical assigned simultaneously */
};
```

This data structure is initialized from the **/etc/master.d/shm** file when the UNIX System is initialized.

Chapter 7

COMMAND DESCRIPTIONS

PAGE

GENERAL 7-1

INTER-PROCESS COMMUNICATION STATUS 7-2

Ipcs Without Options 7-2

Ipcs With Options 7-5

INTER-PROCESS COMMUNICATION REMOVE 7-13

Chapter 7

COMMAND DESCRIPTIONS

GENERAL

This chapter gives usage information for the two Inter-Process Communication (IPC) Utilities. The two utilities are as follows:

- *ipcs* — Inter-Process Communication status
- *ipcrm* — Inter-Process Communication remove.

The following sections contain the usage information and examples for each command.

INTER-PROCESS COMMUNICATION STATUS

The **ipcs** command can be used in two ways:

- Without options
- With options.

Ipcs Without Options

When using **ipcs** without options, a short status format is displayed for all IPC facilities that are in the UNIX System at the time of command execution. The short status format consists of the following information for all types of facilities:

- T—type of the facility
- ID—the identifier for the facility
- KEY—the key used for creating the facility
- MODE—the operation permissions and flags
- OWNER—the login name of the owner of the facility
- GROUP—the group name of the owner of the facility.

The example that follows is the result of entering the following command line:

```
$ipcs<CR>
IPC status from /dev/kmem as of Fri July 19 15:14:45 1985
T   ID    KEY        MODE        OWNER   GROUP
Message Queues:
q    0 0x00000000 S-rw-------    hrp    other
q    1 0x0000000a -Rrw-rw----    hrp    other
q    2 0x00000001 --rw-rw-rw-    hrp    other
Shared Memory:
m    0 0x00000000 D-rw-------    hrp    other
m    1 0x0000000a -Crw-rw----    hrp    other
m    2 0x00000001 -Crw-rw-rw-    hrp    other
Semaphores:
s    0 0x00000000 --ra-------    hrp    other
s    1 0x0000000a --ra-ra----    hrp    other
s    2 0x00000001 --ra-ra-ra-    hrp    other
```

From looking at this example, you can see several points of interest.

First, note that the display is separated into **Message Queues**, **Shared Memory**, and **Semaphores**. Note also that there are common column headings for these facility types. These headings correspond to the short status format information that **ipcs** without options displays as previously discussed.

The codes for the type (T) of facility are **q**, **m**, and **s** for message queues, shared memory, and semaphores, respectively.

Identifiers (ID) are integers (zero and positive) that are returned when creating a facility using the msgget(), shmget(), and semget() system calls.

Keys (KEY) are either IPC_PRIVATE (0) or equal to the value passed to the msgget(), shmget(), or semget() system calls for the **key** argument when creating a new facility; they can be 0, hexadecimal values, or decimal values. See the example display.

Mode (MODE) gives the operation permissions for each type of facility along with flags for the message and shared memory facilities. The mode is represented by a sequence of eleven character fields.

For message queues, the first character field is an S if a process is blocked from sending a message to the facility, and the second character field is an R if a process is blocked from receiving a message from a facility.

For shared memory, the first character field is a D if the shared memory segment facility is to be removed when the last process attached to the segment detaches it, and the second character field is a C if the shared memory segment facility is to be cleared when the first attach is made.

For semaphores, these two fields are not used as **semncnt** and **semzcnt** serve the same purpose. See the /usr/include/sys/sem.h file.

The corresponding special flags are not set for message queues and shared memory when the character field is " - " . These first two character fields are always " - " for semaphores as they are not used.

Operation permissions use the remaining nine character fields. They are used in groups of three and from left-to-right they represent the permissions for OWNER, GROUP, and OTHER. Note that for message queues rw means read/write and for semaphores ra means read/alter. All fields not in use are depicted by a hyphen.

The OWNER column heading gives the owner name of the facility. Note that when using msgctl(), shmctl(), or semctl() to change ownership of a facility, a positive integer value is used to represent the owner. These values can be determined for a particular owner name by searching through the **/etc/passwd** file.

The GROUP column heading gives the group name of the owner. Changing the group is analogous to changing the owner.

Ipcs With Options

The options available for the **ipcs** command consist of facility type options and general options. The facility type options allow the short format status information to be displayed for just the facility type desired. The general options allow information about size, creator, usage, process identification, and time to be observed. The general options can be used with the facility type options to observe the general options for a particular facility type.

Facility Type Options

The options that allow the status of only a particular type of facility to be observed are as follows:

```
-q    Message Queue Type
-m    Shared Memory Type
-s    Semaphore Type
```

Proper formats for entering these options are as follows:

```
$ipcs -q<CR>   Message Queue Type
$ipcs -m<CR>   Shared Memory Type
$ipcs -s<CR>   Semaphore Type
```

The status can be displayed for selected facilities by putting the options on the same command line, separated by spaces.

The following display occurs if status is requested but no facilities exist.

```
ipcs<CR>
IPC status from /dev/kmem as of Fri July 19 09:31:16 1985
T   ID   KEY    MODE    OWNER   GROUP
Message Queues:
Shared Memory:
Semaphores:
```

General Options

The general options allow additional kinds of information to be displayed for all facility types or for specific facility types. In other words, these general options can be used with **ipcs** alone to obtain the desired information for all facility types, or they can be appended to the facility type options for specific facility type information. More than one of these general options can be specified on the command line as well.

The following options are available:

- -b Biggest allowable size
- -c Creator login name and group name
- -o Outstanding usage
- -p Process number
- -t Time
- -a All general options
- -C Use a different corefile than /dev/kmem
- -N Use a different namelist than /unix.

Of course, these options will reflect only the information applicable to each facility type.

The biggest allowable size information option is illustrated as follows:

```
$ipcs -b<CR>
IPC status from /dev/kmem as of Fri July 19 07:55:13 1985
T    ID    KEY         MODE        OWNER   GROUP QBYTES
Message Queues:
q     0 0x00000000 --rw-------   hrp   other  16384
q     1 0x0000000a --rw-rw----   hrp   other  16384
q     2 0x00000001 --rw-rw-rw-   root  other    500
T    ID    KEY         MODE        OWNER   GROUP SEGSZ
Shared Memory:
m     0 0x00000000 -Crw-------   hrp   other   8192
m     1 0x0000000a -Crw-rw----   hrp   other   1024
m     2 0x00000001 -Crw-rw-rw-   root  other   8192
T    ID    KEY         MODE        OWNER   GROUP NSEMS
Semaphores:
s     0 0x00000000 --ra-------   hrp   other    25
s     1 0x0000000a --ra-ra----   hrp   other    25
s     2 0x00000001 --ra-ra-ra-   root  other     5
```

Notice that for the message queue type of facility, QBYTES is the biggest allowable size information that is returned; it has been lowered for ID 2 to be 500. They were all initialized to the value of the system tunable parameter that specifies the maximum allowed bytes on a queue, MSGMNB.

For the shared memory type of facility, SEGSZ is the biggest allowable size information returned. SEGSZ specifies the size in bytes of the shared memory segment. The maximum is 8192 bytes (SHMMAX), and the minimum is 1 (SHMMIN).

For the semaphore type of facility, NSEMS is the biggest allowable size information that is returned. These values were determined when the facilities were created. The nsems argument passed to semget() determines these values. Remember that the system tunable parameter SEMMSL determines the maximum semaphores in a set (25).

The creator information is illustrated as follows:

```
$ipcs -c<CR>
IPC status from /dev/kmem as of Fri July 19 07:56:15 1985
T   ID   KEY        MODE       OWNER GROUP CREATOR CGROUP
Message Queues:
q    0 0x00000000 --rw-------  hrp  other  hrp  other
q    1 0x0000000a --rw-rw----  hrp  other  hrp  other
q    2 0x00000001 --rw-rw-rw-  root other  hrp  other
Shared Memory:
m    0 0x00000000 -Crw-------  hrp  other  hrp  other
m    1 0x0000000a -Crw-rw----  hrp  other  hrp  other
m    2 0x00000001 -Crw-rw-rw-  root other  hrp  other
Semaphores:
s    0 0x00000000 --ra-------  hrp  other  hrp  other
s    1 0x0000000a --ra-ra----  hrp  other  hrp  other
s    2 0x00000001 --ra-ra-ra-  root other  hrp  other
```

The results are the same for all facility types in this case. The column headings CREATOR and CGROUP show the login name and group name of the creator, respectively. The corresponding positive integer values for these names can be determined by searching the **/etc/passwd** file. Remember, the creator of a facility always remains the creator while the owner and group can change.

The outstanding usage option is as follows:

```
$ipcs -o<CR>
IPC status from /dev/kmem as of Fri July 19 07:58:13 1985
T    ID    KEY        MODE       OWNER   GROUP CBYTES QNUM
Message Queues:
q     0 0x00000000 --rw-------   hrp    other    16   1
q     1 0x0000000a --rw-rw----   hrp    other     0   0
q     2 0x00000001 --rw-rw-rw-   root   other   359  14
T    ID    KEY        MODE       OWNER   GROUP NATTCH
Shared Memory:
m     0 0x00000000 D-rw-------   hrp    other     1
m     1 0x0000000a -Crw-rw----   hrp    other     0
m     2 0x00000001 D-rw-rw-rw-   root   other     5
Semaphores:
s     0 0x00000000 --ra-------   hrp    other
s     1 0x0000000a --ra-ra----   hrp    other
s     2 0x00000001 --ra-ra-ra-   root   other
```

For message queues, the CBYTES and QNUM column headings stand for the total amount of bytes in core memory for all messages and the total amount of messages, respectively, for each message queue. The sum of all CBYTES is associated with the product of the amount of segments, MSGSEG, and the size of the segments, MSGSSZ. QNUM is associated with the total amount of bytes allowed for headers on each queue, MSGMNB. The sum of all QNUMs is associated with the total amount of message headers system wide, MSGTQL.

For shared memory, NATTCH corresponds to the amount of processes attached to the facility.

The outstanding usage option does not apply to the semaphore type facility even though the short format status information is displayed for it.

The process number option is illustrated as follows:

```
$ipcs -p<CR>
IPC status from /dev/kmem as of Fri July 19 08:12:53 1985
T   ID    KEY        MODE        OWNER  GROUP LSPID LRPID
Message Queues:
q    0 0x00000000 --rw-------   hrp  other  2275  2281
q    1 0x0000000a --rw-rw----   hrp  other     0     0
q    2 0x00000001 --rw-rw-rw-   root other     0     0
Shared Memory:
m    0 0x00000000 --rw-------   hrp  other   158  2254
m    1 0x0000000a --rw-rw----   hrp  other  2208  2254
m    2 0x00000001 --rw-rw-rw-   root other   166  2252
Semaphores:
s    0 0x00000000 --ra-------   hrp  other
s    1 0x0000000a --ra-ra----   hrp  other
s    2 0x00000001 --ra-ra-ra-   root other
```

For message queues, the LSPID and LRPID column headings represent the last process identifier that sent and received a message from the associated message queue, respectively.

For shared memory, LSPID and LRPID represent the last process identifier to attach and detach from the facility, respectively.

The process number option does not apply to the semaphore type facility even though the short format status information is displayed for it.

The time information option is illustrated as follows:

```
$ipcs -t<CR>
IPC status from /dev/kmem as of Fri July 19 08:15:57 1985
T  ID   KEY   MODE    OWNER GROUP  STIME   RTIME   CTIME
Message Queues:
q  0 0x00000000 --rw------- hrp  other  8:11:44  8:12:07 15:09:25
q  1 0x0000000a --rw-rw---- hrp  other no-entry no-entry 15:09:53
q  2 0x00000001 --rw-rw-rw- root  other no-entry no-entry  7:25:23
T  ID   KEY   MODE    OWNER GROUP  ATIME   DTIME   CTIME
Shared Memory:
m  0 0x00000000 --rw------- hrp  other  8:04:50  8:05:12 15:10:39
m  1 0x0000000a --rw-rw---- hrp  other  8:05:00  8:05:29  7:54:41
m  2 0x00000001 --rw-rw-rw- root  other  8:03:42 no-entry  7:26:30
T  ID   KEY   MODE    OWNER GROUP  OTIME   CTIME
Semaphores:
s  0 0x00000000 --ra------- hrp  other  8:14:45 15:11:56
s  1 0x0000000a --ra-ra---- hrp  other no-entry 15:12:14
s  2 0x00000001 --ra-ra-ra- root  other no-entry  7:25:57
```

The message queue type of facility has three new column headings for this option: STIME, RTIME, and CTIME. STIME represents the last time that a process sent a message. RTIME represents the last time a process received a message. CTIME represents the time of the facility creation or the last time changed with a msgctl() system call.

The shared memory type of facility has three headings also. ATIME represents the time of the last attach operation. DTIME represents the time of the last detach operation. CTIME is the time of the facility creation or the last time changed with a shmctl() system call.

The semaphore type of facility has two new column headings for this option: OTIME, and CTIME. OTIME represents the last time that a process performed operations on the associated semaphore set. CTIME represents the time of the facility creation or the last time changed with a semctl() system call.

The display all options keyletter is illustrated as follows:

> ***Note:*** The short status format information is not included in this example so the pertinent information will fit on the page. On the display screen, the status information will wrap around.

```
$ipcs -a<CR>
IPC status from /dev/kmem as of Fri July 19 08:17:09 1985
CREATOR CGROUP CBYTES QNUM QBYTES LSPID LRPID STIME    RTIME    CTIME
Message Queues:
hrp     other  0      0  16384  2275   2281  8:11:44  8:12:07 15:09:25
hrp     other  0      0  16384  0      0 no-entry no-entry 15:09:53
hrp     other  0      0  500    0      0 no-entry no-entry  7:25:23
CREATOR CGROUP NATTCH SEGSZ CPID LPID  ATIME    DTIME    CTIME
Shared Memory:
hrp     other  0      8192  158  2254  8:04:50  8:05:12 15:10:39
hrp     other  0      1024  2208 2254  8:05:00  8:05:29  7:54:41
hrp     other  0      8192  166  2252  8:03:42 no-entry  7:26:30
CREATOR CGROUP NSEMS  OTIME    CTIME
Semaphores:
hrp     other  25     8:14:45  15:11:56
hrp     other  25     no-entry 15:12:14
hrp     other  5      no-entry  7:25:57
```

The -C and -N options allow all the preceding options to be used on a different *corefile* and *namelist.* These options are useful for performing **ipcs** on a coredump file (-C) or when more than one version of the UNIX System (-N) is installed. Since the status of facilities can change while **ipcs** is running, these options allow more control.

INTER-PROCESS COMMUNICATION REMOVE

The command used to remove IPC facilities is as follows:

```
ipcrm [options]
```

There are two ways to remove a selected IPC facility from the UNIX System:

- Using the facility identifier (ID)
- Using the facility key (KEY).

The following sections illustrate how to remove IPC facilities using their IDs and KEYs.

Removal by ID

The options that are available to remove a facility by its ID are as follows:

- -q *msqid*
- -m *shmid*
- -s *semid*

An example of its use is as follows:

```
$ipcrm -q2 -s1 -q0 -m1<CR>
```

Note that the options can be repeated and placed on the command line in any order. The result of this command line will be to remove message queues 2 and 0, semaphore set 1, and shared memory segment 1.

Removal by Key

> ***Note:*** The key used for **ipcrm** must be a decimal value. The **ipcs** command reports keys in hexadecimal, however.

The options available to remove a facility by its key use the same letters as for removal by ID except that they are capital letters. They are as follows:

- -Q *msgkey*
- -M *shmkey*
- -S *semkey*

An example using these options follows:

```
$ipcrm -Q0 -S10 -Q1 -M1<CR>
```

The result of this command is to remove the message queue facilities with keys of **0** and **1**, to remove the semaphore facility with the key of **a** (10), and to remove the shared memory segment with the key of **1**.

Appendix

IPC ERROR CODES

PAGE

MESSAGE ERROR CODES A-2

Msgget() A-2

Msgctl() A-3

Msgop() A-5

SEMAPHORE ERROR CODES A-8

Semget() A-8

Semctl() A-10

Semop() A-12

SHARED MEMORY ERROR CODES A-16

Shmget() A-16

Shmctl() A-18

Shmop() A-20

Appendix

IPC ERROR CODES

This appendix contains the error codes for the 3B2 Computer Inter-Process Communication (IPC) system calls. Positive integer error codes are set in the external **errno** variable when a system call is unsuccessful.

An error has occurred when an IPC system call returns a -1 value. The value of **errno** is only valid immediately following this occurrence.

Each error code number has a corresponding mnemonic name. In this appendix, error code numbers and mnemonic names are categorized by facility type and associated system calls. The error reasons as they apply to IPC system calls are given.

These error codes are the same as those on the **intro(2)** manual page found in the *AT&T 3B2 Computer Programmer Reference Manual.* The reasons for the errors given there are more general than the reasons given in this appendix as they are used for all system calls.

MESSAGE ERROR CODES

The IPC error codes for the message type facility are contained in this section.

Msgget()

Each possible error code number, along with its mnemonic and reason(s), that the msgget() system call returns is contained in Figure A-1.

IPC (MSGGET) ERROR CODES		
NUMBER	MNEMONIC	REASON
2	ENOENT	A key not already in use is passed to the system call, but the IPC_CREAT flag is not set.
13	EACCES	Operation permissions deny the calling process.
17	EEXIST	A key already in use is passed to the system call with the IPC_CREAT and IPC_EXCL flags set. This is the exclusive create ability.
28	ENOSPC	The system wide amount of message queue identifiers would be exceeded (MSGMNI).

Figure A-1. Msgget Error Codes

Msgctl()

Each possible error code number, along with its mnemonic and reason(s), that the msgctl() system call returns is contained in Figure A-2.

IPC (MSGCTL) ERROR CODES		
NUMBER	MNEMONIC	REASON
1	EPERM	The process does not have the effective user identification of OWNER/CREATOR (msg_perm.[c]uid) of the facility or super-user when an IPC_RMID or IPC_SET control command is specified.
		The process does not have the effective user identification of super-user when using IPC_SET to increase the number of bytes (msg_qbytes) for the specified message queue.
13	EACCES	Operation permissions deny the calling process.
14	EFAULT	The pointer (buf) passed to the system call does not point to the necessary data structure (msqid_ds) in the user memory area.
22	EINVAL	The message queue identifier (msqid) is invalid; the facility does not exist.
		The value of the control command (cmd) passed to the system call is not equal to IPC_STAT, IPC_SET, or IPC_RMID.

Figure A-2. Msgctl Error Codes

Msgop()

Each possible error code number, along with its mnemonic and reason(s), that the msgsnd() and msgrcv() system calls return is contained in Figures A-3 and Figure A-4, respectively.

IPC (MSGSND) ERROR CODES		
NUMBER	MNEMONIC	REASON
4	EINTR	The process received a signal while it was performing a " blocking message operation" that was blocked. (IPC_NOWAIT is not set.)
11	EAGAIN	The process cannot send a message because there are not enough bytes on the message queue (msg_qbytes), or the total amount of messages on all message queues would be exceeded (MSGTQL) while the process is performing a " nonblocking message operation." (IPC_NOWAIT flag is set.)
13	EACCES	Operation permissions deny the calling process.
14	EFAULT	The pointer (msgp) passed to the system call does not point to the necessary data structure (msgbuf) in the user memory area. (The data structure contains the message type value and message text array.)

Figure A-3. Msgsnd Error Codes (Sheet 1 of 2)

IPC (MSGSND) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
22	EINVAL	The message queue identifier (msqid) is invalid; the facility does not exist.
		The value of the message type (msgtyp) variable passed to the system call is less than 1.
		The message size (msgsz) value passed to the system call is less than zero or greater than the system imposed limit, MSGMAX, for maximum message size.
36	EIDRM	The facility that the system call is performing a " blocking message operation" on is removed while the process is blocked. (IPC_NOWAIT is not set.)
4	EINTR	The process received a signal while it was performing a " blocking message operation" that was blocked. (IPC_NOWAIT is not set.)

Figure A-3. Msgsnd Error Codes (Sheet 2 of 2)

IPC (MSGRCV) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
7	E2BIG	The value passed to the system call for the message size (msgsz) to be received is less than the message size and the IPC_NOERROR flag is not set.
13	EACCES	Operation permissions deny the calling process.
14	EFAULT	The pointer (msgp) passed to the system call does not point to the necessary data structure (msgbuf) in the user memory area. (The data structure contains the message type value and message text array.)
22	EINVAL	The message queue identifier (msqid) is invalid; the facility does not exist.
		The value of the message size (msgsz) variable passed to the system call is less than zero.
35	ENOMSG	The specified message queue does not contain the desired message type (msgtyp), and the IPC_NOWAIT flag is set (msgflg).

Figure A-4. Msgrcv Error Codes

SEMAPHORE ERROR CODES

The IPC error codes for the semaphore type facility are contained in this section.

Semget()

Each possible error code number, along with its mnemonic and reason(s), that the semget() system call returns is contained in Figure A-5.

IPC (SEMGET) ERROR CODES		
NUMBER	MNEMONIC	REASON
2	ENOENT	A key not already in use is passed to the system call, but the IPC_CREAT flag is not set.
13	EACCES	Operation permissions deny the calling process.
17	EEXIST	A key already in use is passed to the system call with the IPC_CREAT and IPC_EXCL flags set. This is the exclusive create ability.
22	EINVAL	The number of semaphores (nsems) to be in the set is less than or equal to zero or greater than the system tunable parameter SEMMSL.
		The value passed to the system call for the number of semaphores (nsems) is greater than what is in the set.
28	ENOSPC	The system call would cause the maximum amount of semaphore identifiers (sets) system wide to be exceeded (SEMMNI).
		The system call would cause the maximum amount of semaphores in all sets to be exceeded (SEMNS).

Figure A-5. Semget Error Codes

Semctl()

Each possible error code number, along with its mnemonic and reason(s), that the semctl() system call returns is contained in Figure A-6.

IPC (SEMCTL) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
1	EPERM	The process does not have the effective user identification of OWNER/CREATOR (sem_perm.[c]uid) of the facility or super-user when an IPC_RMID or IPC_SET control command is specified.
13	EACCES	Operation permissions deny the calling process.
14	EFAULT	The pointer (arg.buf) passed to the system call does not point to the necessary union data structure (semun) in the user memory area.
22	EINVAL	The semaphore set identifier (semid) is invalid; the facility does not exist.
		The semaphore number (0 through 24, semnum) is less than zero or greater than the number of semaphores in the set (sem_nsems).
		The value of the control command (cmd) passed to the system call is invalid.
34	ERANGE	When setting a semaphore(s) value (SETVAL, SETALL), the system imposed maximum is exceeded (SEMVMX).

Figure A-6. Semctl Error Codes

Semop()

Each possible error code number, along with its mnemonic and reason(s), that the semop() system call returns is contained in Figure A-7.

IPC (SEMOP) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
4	EINTR	The process received a signal while it was performing a " blocking semaphore operation" that was blocked. (IPC_NOWAIT is not set.)
7	E2BIG	The value passed to the system call for the number of semaphore operations (nsops) to be performed exceeds the system tunable parameter SEMOPM.
11	EAGAIN	The process would be blocked from performing its semaphore operation, but the IPC_NOWAIT flag is set.
13	EACCES	Operation permissions deny the calling process.

Figure A-7. Semop Error Codes (Sheet 1 of 3)

IPC (SEMOP) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
14	EFAULT	The pointer (sops) passed to the system call does not point to an array of data structures (sembuf) in the user memory area. (Each data structure in the array contains the semaphore number, the operation to be performed, and the control command flags.)
22	EINVAL	The semaphore set identifier (semid) is invalid; the facility does not exist.
		The maximum amount of undo entries per system call (SEMUME) system tunable parameter would be exceeded.
27	EFBIG	The semaphore number (sem_num) for a data structure in the array is less than zero or greater than or equal to the total semaphores in the set.

Figure A-7. Semop Error Codes (Sheet 2 of 3)

IPC (SEMOP) ERROR CODES		
NUMBER	MNEMONIC	REASON
28	ENOSPC	The maximum amount of undo entries system wide (SEMMNU system tunable parameter) would be exceeded.
34	ERANGE	The result of the operation would cause the semaphore value to exceed the maximum value for a semaphore (SEMVMX).
		The result of the operation would cause the maximum undo data structure value for semaphore adjust (SEMAEM) to be exceeded.
36	EIDRM	The facility that the system call is performing a "blocking message operation" on is removed while the process is blocked. (IPC_NOWAIT is not set.)

Figure A-7. Semop Error Codes (Sheet 3 of 3)

SHARED MEMORY ERROR CODES

The IPC error codes for the shared memory type facility are contained in this section.

Shmget()

Each possible error code number, along with its mnemonic and reason(s), that the shmget() system call returns is contained in Figure A-8.

IPC (SHMGET) ERROR CODES		
NUMBER	MNEMONIC	REASON
2	ENOENT	A key not already in use is passed to the system call, but the IPC_CREAT flag is not set.
12	ENOMEM	There is not enough physical memory to fill the request.
13	EACCES	Operation permissions deny the calling process.
17	EEXIST	A key already in use is passed to the system call with the IPC_CREAT and IPC_EXCL flags set. This is the exclusive create ability.
22	EINVAL	The size of the shared memory segment passed to the system call is invalid. The size must be greater than or equal to 1 or less than or equal to 8192. See SHMMIN and SHMMAX.
		An identifier exists for the facility (key) but the shared memory segment size is less than the size (not zero) passed to the system call.
28	ENOSPC	The system wide amount of shared memory segment identifiers would be exceeded (SHMMNI).

Figure A-8. Shmget Error Codes

Shmctl()

Each possible error code number, along with its mnemonic and reason(s), that the shmctl() system call returns is contained in Figure A-9.

IPC (SHMCTL) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
1	EPERM	The process does not have the effective user identification of OWNER/CREATOR (shm_perm.[c]uid) of the facility or super-user when an IPC_RMID or IPC_SET control command is specified.
		The process does not have the effective user identification of super-user when using the SHM_LOCK or SHM_UNLOCK commands.
13	EACCES	Operation permissions deny the calling process.

Figure A-9. Shmctl Error Codes (Sheet 1 of 2)

IPC (SHMCTL) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
14	EFAULT	The pointer (buf) passed to the system call does not point to the necessary data structure (shmid_ds) in the user memory area.
22	EINVAL	The shared memory identifier (shmid) is invalid; the facility does not exist.
		The value of the control command (cmd) passed to the system call is not equal to IPC_STAT, IPC_SET, IPC_RMID, SHM_LOCK, or SHM_UNLOCK.
		The command is SHM_UNLOCK but the specified shared memory segment is not locked in memory.

Figure A-9. Shmctl Error Codes (Sheet 2 of 2)

Shmop()

Each possible error code number, along with its mnemonic and reason(s), that the shmat() and shmdt() system calls return is contained in Figures A-10 and Figure A-11, respectively.

IPC (SHMAT) ERROR CODES		
NUMBER	MNEMONIC	REASON
12	ENOMEM	There is not enough in core memory to accommodate the shared memory segment.
13	EACCES	Operation permissions deny the calling process.
22	EINVAL	The shared memory identifier (shmid) is invalid; the facility does not exist.
		The shared memory address (shmaddr) passed to the system call is not equal to zero and [shmaddr-(shmaddr modulus SHMLBA)] is an illegal address.
		The shared memory address (shmaddr) passed to the system call is not equal to zero, the SHM_RND flag is false, and the address is illegal.

Figure A-10. Shmat Error Codes

IPC (SHMDT) ERROR CODES		
NUMBER	**MNEMONIC**	**REASON**
13	EACCES	Operation permissions deny the calling process.
22	EINVAL	The system call detaches the shared memory segment located at the specified address (shmaddr) from the process data segment.
		The address (shmaddr) passed to the system call is not the start of a shared memory segment.
24	EMFILE	The number of shared memory segments attached to the calling process would exceed the number allowed, SHMSEG.

Figure A-11. Shmdt Error Codes