System V IPC 5

The SunOS 5.x system provides several mechanisms that allow processes to exchange data and synchronize execution. The simpler of these mechanisms are pipes, named pipes, and signals. These are limited, however, in what they can do:

- Pipes do not allow unrelated processes to communicate.
- Named pipes allow unrelated processes to communicate, but do not provide private channels for pairs of communicating processes; that is, any process with appropriate permission can read from or write to a named pipe.
- Sending signals with the `kill` function allows arbitrary processes to communicate, but the message consists only of the signal number.

The SunOS 5.x system provides an InterProcess Communication (IPC) package that supports three more versatile types of interprocess communication:

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

When implemented as a unit, these three mechanisms share common properties:

- Each mechanism contains a “get” function to create a new entry or retrieve an existing one.
- Each mechanism contains a “control” function to query the status of an entry, to set status information, and to remove the entry from the system.

- Each mechanism contains one or more “operations” functions to perform various operations on an entry.

This chapter describes the functions for each of these three forms of IPC.

This information is for programmers who write multiprocess applications. These programmers should have a general understanding of what semaphores are and how they are used.

See the following manual pages as listed in Code Example 5-1 for more information about IPC.

Table 5-1 IPC Reference Manual Pages

`ipcrm`(1)	`ipcs`(1)	`intro`(2)
`msgget`(2)	`msgctl`(2)	`msgop`(2)
`semget`(2)	`semctl`(2)	`semop`(2)
`shmget`(2)	`shmctl`(2)	`shmop`(2)
`stdipc`(3C)		

Included in this chapter are several example programs showing the use of these IPC functions. You can accomplish the same task in many ways, so keep in mind that the example programs were written for clarity and not for program efficiency. Usually, functions are embedded within a larger user-written program that uses a particular function provided by the calls.

Permissions

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

The definition of the IPC permissions data structure `ipc_perm` is given in `<sys/ipc.h>`:

Code Example 5-1 IPC Permissions Data Structure

```
struct ipc_perm
{
    uid_t   uid;    /* owner's user id */
```

```
    gid_t   gid;    /* owner's group id */
    uid_t   cuid;   /* creator's user id */
    gid_t   cgid;   /* creator's group id */
    mode_t  mode;   /* access modes */
    ulong   seq;    /* slot usage sequence number */
    key_t   key;    /* key */
    long    pad[4]; /*reserve area */
};
```

This structure is common to messages, semaphores, and shared memory. Permissions for an IPC facility are initialized by the creating process and can be modified by any process with permission to perform control operations on that facility.

Permissions are specified as octal values in the flags argument of the appropriate IPC creation or control function:

Table 5-2 Octal Permission Values

Access Permissions	Octal Value
Write by Owner	0200
Read by Owner	0400
R/W by Owner	0600
Write by Group	0020
Read by Group	0040
R/W by Group	0060
Write by Others	0002
Read by Others	0004
R/W by Others	0006

For instance, to get read access by the owner and read and write access by others, the permissions value is 0406.

IPC Functions, Key Arguments, and Creation Flags

Processes requesting access to a common IPC facility must have a way to determine the identity of the facility. To do this, functions that initialize or provide access to an IPC facility use a *key* argument (of type `key_t`).

This key is a value known to all the programs, or one that can be derived from a common seed at run time. A common way to derive the key is with `ftok` (see `stdipc`(3C)). This converts a filename to a key value that is virtually unique within the system. The key value can be used by all programs (processes) attempting to access the facility.

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

If the key argument is specified as `IPC_PRIVATE` (defined to be zero), the call initializes a new instance of an IPC facility that is private to the creating process.

When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the function attempts to create the facility if it does not exist already.

When called with both the `IPC_CREAT` and `IPC_EXCL` flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with `IPC_EXCL` in effect, only the first attempt succeeds.

If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If `IPC_CREAT` is omitted and the facility is not already initialized, the calls fail.

These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/tmp", 'A'), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a key (`'A'` in the following figure) based on the string (`"/tmp"` in the following figure). The second argument evaluates to the combined permissions and control flags.

Messages

IPC messaging allows processes to send and receive messages, and to queue messages for processing in an arbitrary order. Unlike the file byte-stream model of data flow used for pipes, each IPC message has an explicit length. More importantly, messages can be assigned a specific type. Because of this, a

server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized through the `msgget`(2) function. The owner or creator of a queue can change its ownership or permissions using `msgctl`(2). Also, any process with permission to do so can use `msgctl()` for control operations.

Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively (see `msgop`(2)). When a message is sent, its text is copied to the message queue.

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

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Structure of a Message Queue

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

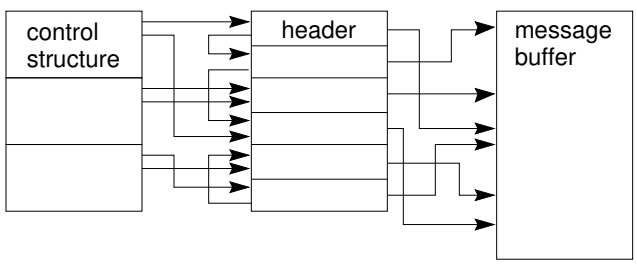

Figure 5-1 Structure of a Message Queue

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

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

Each message header contains the following information:

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

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

Code Example 5-2 Message Queue Control Structure

```
struct msqid_ds
{
    struct ipc_permmsg_perm;      /* operation permission struct */
    struct msg     *msg_first;    /* ptr to first message on q */
    struct msg     *msg_last;     /* ptr to last message on q */
    ulong          msg_cbytes;    /* current # bytes on q */
    ulong          msg_qnum;      /* # of messages on q */
    ulong          msg_qbytes;    /* max # of bytes on q */
    pid_t          msg_lspid;     /* pid of last msgsnd */
    pid_t          msg_lrpid;     /* pid of last msgrcv */
    time_t         msg_stime;     /* last msgsnd time */
    long           msg_pad1;      /* reserved to expand time_t */
    time_t         msg_rtime;     /* last msgrcv time */
    long           msg_pad2;      /* time_t expansion */
    time_t         msg_ctime;     /* last change time */
    long           msg_pad3;      /* time expansion */
    long           msg_pad4[4];   /* reserve area*/
};
```

The definition of the message-header data structure is the following

Code Example 5-3 Message Header Structure

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

Initializing a Message Queue with `msgget()`

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

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

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

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

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

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

Code Example 5-4 Example of `msgget()` call

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

main()
{
    key_t   key;    /* key to be passed to msgget() */
    int     msgflg, /* msgflg to be passed to msgget() */
```

```
            msqid;  /* return value from msgget() */

    ...
    key = ...
    msgflg = ...
    if ((msqid = msgget(key, msgflg)) == -1)
    {
        perror("msgget: msgget failed");
        exit(1);
    } else
        (void) fprintf(stderr, "msgget succeeded");
    exit(0);
}
```

Controlling Message Queues with `msgctl()`

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

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

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

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

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

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

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

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

The following code sample illustrates the `msgctl`(2) function with all its various flags:

Code Example 5-5 Example `msgctl()` calls

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

    ...
    do_msgctl(msqid, IPC_STAT, &buf);
    ...
    do_msgctl(msqid, IPC_SET, &buf);
    ...


int
do_msgctl(msqid, cmd, buf)
struct msqid_ds*buf;   /* pointer to queue descriptor buffer */
int             cmd,   /* command code */
                msqid; /* queue ID */
{
    int rtrn;   /* hold area for return value from msgctl() */

    if (rtrn = msgctl(msqid, cmd, buf) == -1) {
        perror("msgctl: msgctl failed");
        exit(1);
    } else {
        return (rtrn);
    }
}
```

Sending and Receiving Messages

The `msgsnd()` and `msgrcv()` functions (see the `msgop`(2) manual page) send and receive messages, respectively. Their synopses are:

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

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

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

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

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

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

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

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

The following code sample illustrates `msgsnd()` and `msgrcv()`:

Code Example 5-6 Example `msgsnd`() and `msgrcv`() calls

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

{
    int             msgflg;/* message flags for the operation */
    struct msgbuf  *msgp;  /* pointer to the message buffer */
    int             msgsz;  /* message size */
    long            msgtyp; /* desired message type */
    int             msqid   /* message queue ID to be used */

    ...
```

```
msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
            - sizeof msgp->mtext + maxmsgsz));
if (msgp == NULL) {
    (void) fprintf(stderr, "msgop: %s %d byte messages.\n",
            "could not allocate message buffer for", maxmsgsz);
    exit(1);

    ...
    msgsz = ...
    msgflg = ...
    if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
        perror("msgop: msgsnd failed");
    ...

    ...
    msgsz = ...
    msgtyp = first_on_queue;
    msgflg = ...
    if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
        perror("msgop: msgrcv failed");
    ...
```

Semaphores

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

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

Before a process can use a semaphore, the semaphore set must be initialized using `semget`(2). The semaphore owner or creator can change its ownership or permissions using `semctl`(2). Also, any process with permission to do so can use `semctl()` to perform control operations.

Semaphore operations are performed by the `semop`(2) function. This call accepts a pointer to an array of semaphore operation structures. Each structure in the operations array contains information about an operation to perform on a semaphore. The operations array is described in detail in the *Semaphore Operations* section.

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

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

- the semaphore operations can all finish, so the call succeeds,
- the process receives a signal, or
- the semaphore set is removed.

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

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

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

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

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

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

When performing a semaphore operation with `SEM_UNDO` in effect, you must also have it in effect for the call that would perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value.

This insures that, unless the process is aborted, the values applied to the undo structure will eventually cancel out to zero. When the undo structure reaches zero, it is removed.

Using `SEM_UNDO` inconsistently can lead to excessive resource consumption because allocated undo structures might not be freed until the system is rebooted.

Structure of a Semaphore Set

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

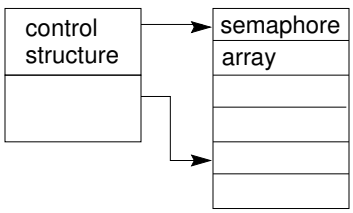

Figure 5-2 Structure of a Semaphore

The control structure for the semaphore contains the following information:

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

- The time of the last update to any semaphore in the array

Each semaphore structure in the array contains the following information:

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

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

```
struct semid_ds
{
    struct ipc_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 */
    long               sem_pad1;  /* reserved for time_t expansion */
    time_t             sem_ctime; /* last change time */
    long               sem_pad2; /* time_t expansion */
    long               sem_pad3[4]; /* reserve area */
};
```

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

The semaphore structure is defined in the same header file

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

Initializing a Semaphore Set with `semget()`

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

Code Example 5-7 Synopsis of `semget()`

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

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

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

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

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

The following program illustrates the `semget()` function. It begins by prompting for a hexadecimal key, an octal permissions code, and control command combinations selected from a menu. All possible combinations are allowed.

It then asks the number of semaphores in the array and issues the function to initialize the array. If the call succeeds, the program displays the returned semaphore ID. Otherwise, it displays an error message:

Code Example 5-8 Example `semget()` call

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

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

    ...
    key = ...
```

```
nsems = ...
semflg = ...
...
if ((semid = semget(key, nsems, semflg)) == -1) {
    perror("semget: semget failed");
    exit(1);
} else
    exit(0);
}
```

Controlling Semaphores with `semctl`**()**

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

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

union semun {
    int val;
    struct semid_ds *buf;
    ushort * array;
};

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

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

`GETVAL`
Return the value of a single semaphore.

`SETVAL`
Set the value of a single semaphore. In this case, `arg` is taken as `arg.val`, an `int`.

`GETPID`
Return the PID of the process that performed the last operation on the semaphore or array.

GETNCNT
Return the number of processes waiting for the value of a semaphore to increase.

GETZCNT
Return the number of processes waiting for the value of a particular semaphore to reach zero.

GETALL
Return the values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of `unsigned shorts`.

SETALL
Set values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of `unsigned shorts`.

IPC_STAT
Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by `arg.buf`, a pointer to a buffer of type `semid_ds`.

IPC_SET
Set the effective user and group identification and permissions. In this case, `arg` is taken as `arg.buf`.

IPC_RMID
Remove the specified semaphore set.

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

The following program illustrates `semctl()`

Code Example 5-9 Example `semctl()` call

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

    register int   i;

    i = semctl(semid, semnum, cmd, arg);
    if (i == -1) {
```

```
        perror("semctl: semctl failed");
        exit(1);
    }
```

Performing Semaphore Operations with `semop()`

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

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

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

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

- The semaphore number
- The operation to be performed
- Control flags, if any

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

```
struct sembuf {
    ushort  sem_num;    /* semaphore # */
    short   sem_op;     /* semaphore operation */
    short   sem_flg;    /* operation flags */
};
```

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

The operation to be performed is determined as follows:

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

You can use the following control flags with `semop()`:

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

`SEM_UNDO`
This command allows individual operations in the array to be undone when the process exits.

The following program illustrates the `semop()` function

Code Example 5-10 Example `semop()` call

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

    int            i;          /* work area */
    int            nsops;      /* number of operations to do */
    int            semid;      /* semid of semaphore set */
    struct sembuf  *sops;      /* ptr to operations to perform */

    ...
    if ((i = semop(semid, sops, nsops)) == -1) {
        perror("semop: semop failed");
    } else
        (void) fprintf(stderr, "semop: returned %d\n", i);
    }
```

System V Shared Memory

In the Solaris 2.x operating system, the most efficient way to implement shared memory applications is to rely on native virtual memory management and the `mmap`(2) function. However, Solaris 2.x also supports System V shared memory.

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

A process creates a shared memory segment using the `shmget`(2) function. This call can also be used to obtain the ID of an existing shared segment. The creating process sets the permissions and the size in bytes for the segment.

The original owner of a shared memory segment can assign ownership to another user with the `shmctl`(2) function; it can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`.

Once created, a shared segment can be attached to a process address space using the `shmat`() function; it can be detached using `shmdt`(). (See `shmop`(2) for details.)

The attaching process must have the appropriate permissions for `shmat()` to succeed. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process.

If the above-mentioned function fails, it returns –1 and sets the external variable `errno` to the appropriate value.

Structure of a System V Shared Memory Segment

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

Figure 5-3 Structure of a Shared Memory Segment

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

- Access permissions.
- Segment size.
- The PID of the process performing last operation.

- The PID of the creator process.
- The current number of processes to which the segment is attached.
- The time of the last attachment.
- The time of the last detachment.
- The time of the last change to the segment.
- Memory map segment descriptor pointer.

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

```
struct shmid_ds {
    struct ipc_permshm_perm;  /* operation permission struct */
    int            shm_segsz; /* size of segment in bytes */
    struct anon_map*shm_amp;  /* segment anon_map pointer */
    ushort         shm_lkcnt; /* number of times it is locked */
    pid_t          shm_lpid;  /* pid of last shmop */
    pid_t          shm_cpid;  /* pid of creator */
    ulong          shm_nattch;/* used only for shminfo */
    ulong          shm_cnattch;/* used only for shminfo */
    time_t         shm_atime; /* last shmat time */
    long           shm_pad1;  /* reserved for time_t expansion */
    time_t         shm_dtime; /* last shmdt time */
    long           shm_pad2;  /* reserved for time_t expansion */
    time_t         shm_ctime; /* last change time */
    long           shm_pad3;  /* reserved for time_t expansion */
    long           shm_pad4[4];/* reserve area */
};
```

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

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

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

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

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

The value passed as the `shmflg` argument must be an integer, which incorporates settings for the segment's permissions and control flags, as described under "Permissions" on page 66.

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

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

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

Code Example 5-11 Sample Program to Illustrate `shmget()`

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

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

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

    ...
    key = ...
    size = ...
    shmflg) = ...
    if ((shmid = shmget (key, size, shmflg)) == -1) {
        perror("shmget: shmget failed");
        exit(1);
    } else {
        (void) fprintf(stderr,
                    "shmget: shmget returned %d\n", shmid);
        exit(0);
    }
    ...
```

Controlling a Shared Memory Segment with `shmctl()`

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

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

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

Figure 5-4 Synopsis of `shmctl()`

The `shmid` argument is the ID of the shared memory segment as returned by `shmget()`.

The `cmd` argument is one of following control commands:

`SHM_LOCK`
: Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.

`SHM_UNLOCK`
: Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.

`IPC_STAT`
: Return the status information contained in the control structure and place it in the buffer pointed to by `buf`. The process must have read permission on the segment to perform this command.

`IPC_SET`
: Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.

`IPC_RMID`
: Remove the shared memory segment. The process must have an effective ID of owner, creator or superuser to perform this command.

The following program illustrates the `shmctl()` function

Code Example 5-12 Sample `shmctl()` call

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

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

    ...
    shmid = ...
    cmd = ...
    if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
        perror("shmctl: shmctl failed");
        exit(1);
    ...
```

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

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

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

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

int shmdt (void *shmaddr);
```

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

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

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

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

The following sample code illustrates calls to shmat() and shmdt():

Code Example 5-13 Sample `shmat()` and `shmdt()` calls

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

static struct state{   /* Internal record of currently attached
segments. */
    int     shmid;      /* shmid of attached segment */
    char    *shmaddr;   /* attach point */
    int     shmflg;     /* flags used on attach */
    ap[MAXnap];         /* State of current attached segments. */

static int      nap;    /* Number of currently attached segments. */

{
    register int    action;     /* action to be performed */
    char            *addr;      /* address work variable */
    register int    i;          /* work area */
    register struct state*p;    /* ptr to current state entry */

    p = &ap[nap++];
    p->shmid = ...
    p->shmaddr = ...
    p->shmflg = ...
    p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
    if(p->shmaddr == (char *)-1) {
        perror("shmop: shmat failed");
        nap--;
    } else
         (void) fprintf(stderr, "shmop: shmat returned %#8.8x\n",
                    p->shmaddr);
    ...

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

```
        (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);
        for (p = ap, i = nap; i--; p++) {
            if (p->shmaddr == addr)
                *p = ap[--nap];
        }
    }
    ...
```