Chapter 2

Interprocess Communication

The term *interprocess communication* (*IPC*) describes any method of sending data from one running process to another. IPC is commonly used to allow processes to coordinate the use of shared data objects, for instance, to let two programs update the same data in memory without interfering with each other, or to make data acquired by one process available to others. IRIX supports a wide variety of IPC mechanisms.

This chapter describes IPC and covers the following topics:

- "Types of Interprocess Communication Available" on page 34 gives a brief overview of the types of IPC available.
- "IRIX IPC" on page 35 explains IRIX shared arenas, locks, semaphores, and shared memory.
- "System V IPC" on page 50 describes System V messages, semaphores, and shared memory.

Types of Interprocess Communication Available

IRIX supports IPC mechanisms of the types shown in Table 2-1.

Table 2-1 Types of IPC and Available Implementations

Type of IPC	Purpose	Implementations
Shared memory	A way to create a segment of memory that is mapped into the address space of two or more processes, each of which can access and alter the memory contents	SVR4, IRIX
Semaphores	The traditional mechanism for coordinating access to a shared resource, by which concurrent processes ensure mutually exclusive use of memory when necessary	SVR4, IRIX
Locks	A mechanism for mutual exclusion that is less general than the semaphore but that has less overhead	IRIX
Barriers	A mechanism for ensuring that all processes are ready before any of them proceed	IRIX
Message Queues	A set of functions by which processes can exchange messages	SVR4
Sockets	Virtual data connections between processes that may be in different systems	BSD

These IPC features are available in different implementations:

- Shared memory, semaphores, locks, and barriers are available using IRIX-unique library calls. The implementation is highly tuned, takes advantage of concurrent hardware in multiprocessors, and has a number of special features.
- Shared memory, semaphores, and message queues are also provided through system function calls that are compatible with AT&T System V Release 4. This implementation is portable, but lacks some features of the IRIX IPC and is sometimes not as fast.
- Socket support is compatible with BSD conventions.

Use System V IPC functions for code that must comply with the MIPS ABI, code that needs to be portable, or code that you're porting from another System V operating system. Use IRIX arena-based IPC for applications that require speed, ease of implementation, or multiprocessing ability. Socket-based IPC is necessary only for code being ported from or to a BSD system, and for network IPC.

This chapter describes the available types of System V and IRIX IPC, and provides examples of each.

IRIX IPC

To meet the demands of parallel programming, IRIX provides a set of fast, low-overhead interprocess communication mechanisms, implemented as extended functions in the C library (which means they operate in user space, often without system calls). Although these mechanisms are powerful and easy to use, remember that they are IRIX-specific, so code using them is not portable to other systems.

The System V IPC mechanisms use their own name space (see "System V IPC" on page 50), but IRIX IPC mechanisms use the filesystem name space. To begin using IRIX IPC, a process must specify the name of a file to be used as a *shared arena*. The named file acts as the backing store for the arena segment (see "Page Validation" on page 9).

Processes gain access to the arena by specifying its filename. All processes using the same arena have access to the same set of IPC mechanisms. This makes it relatively easy for unrelated processes to communicate using IRIX IPC. The shared arena file is mapped into each process's address space, which means that most of the shared arena IPC functions are done in library code, without need for system calls. This is one reason that the overhead on IRIX IPC is lower than the overhead on System V IPC.

IRIX IPC comprises four main mechanisms, two of which—semaphores and shared memory—resemble their System V counterparts. The other mechanisms are *locks*, simple busy-wait locks for low-level synchronization, and *barriers*, which provide rendezvous points for multiple processes.

Note: Modules using IRIX IPC routines should include <*stdio.h*> and <*ulocks.h*>, and should be linked with the *libmpc.so* shared object (*ld* option *-lmpc*).

For more information on any of the routines presented here, see the appropriate reference pages. (Click on the name of a reference page to open it online.)

This section explains how to use IRIX IPC. Topics include the following:

- "Initializing a Shared Arena" describes how to prepare a shared memory arena.
- "Allocating in a Shared Arena" describes how to allocate and free shared memory.
- "Exchanging the First Datum" explains how to communicate the location of an object in an arena to another process using the arena.
- "Using IRIX Semaphores" explains how to allocate and change the value of a shared-arena semaphore.
- "Using Locks" covers how to allocate and use locks.
- "Using Barriers" covers how to allocate, use, and free a barrier.

Initializing a Shared Arena

All IRIX IPC mechanisms require a shared memory arena. A program creates a shared memory arena with the **usinit()** function. However, many attributes of the arenas that **usinit()** creates are set by preceding calls to **usconfig()**.

Configuring an Arena

You often call **usconfig()** prior to creating an arena in order to establish the following features:

CONF_INITSIZE	Set the initial size of the arena segment. The default is 64 KB, and often you know that more is needed immediately.
CONF_INITUSERS	Set the largest number of concurrent processes that can use the arena. The default is 8; if the program creates more child processes than this, the limit must be set higher.
CONF_CHMOD	Set effective file permissions on arena access. The default is 600, allowing only processes with the effective UID of the creating process to attach the arena.
CONF_ARENATYPE	Set optional arena handling choices.
CONF_LOCKTYPE	Set lock metering on for locks created in this arena.

See the usconfig(3P) reference page for a complete list of attributes. In programs that start a varying number of child processes, it is a common mistake to find that the eighth child cannot be created simply because **usconfig()** has not been used to set a higher number of users before an arena was created.

Creating an Arena

Having set the arena attributes, the program calls **usinit()**, specifying a file pathname string.

Tip: The **mktemp()** library function can be used to create a unique temporary filename (see the mktemp(3C) reference page).

If the specified file doesn't exist, **usinit()** creates it. If a shared arena already exists based on that name, **usinit()** joins that shared arena. If the file exists but isn't a shared arena, **usinit()** overwrites it. In any case, **usinit()** is subject to normal filesystem permission checks: it returns an error if the process doesn't have read and write permission on the file (if it already exists) or permission to create the file (if it doesn't exist).

Code to prepare an arena is shown in Example 2-1.

Example 2-1 Initializing a Shared Memory Arena

```
usptr_t
makeArena(size_t initSize, int nProcs)
{
   int ret;
   char * tmpname = "/var/tmp/arenaXXXXXX";
   if (ret = usconfig(CONF_INITUSERS, nProcs))
   { perror("usconfig(#users)"); return 0; }
   if (ret = usconfig(CONF_INITSIZE, initSize))
   { perror("usconfig(size)"); return 0; }
   return usinit(mktemp(tmpname));
}
```

Joining an Arena

Only one process creates a shared arena—other processes "join" or "attach" the arena. There are three ways of doing this. Normally, the arena is not restricted to a single process family. In this case, any process that calls **usinit()** and passes the same pathname string gains access to the same arena at the same virtual base address. This process need not be related in any way to the process that created the arena.

You can restrict arena access to a single process and to the children it creates with **sproc()** by calling **usconfig()** to set the arena type before creating the arena. When this is done, a call to **usinit()** with the same pathname creates a new arena at a possibly different base address. The two arenas are related through the contents of the file, and unpredictable results can occur if the processes do not coordinate updates to the data.

An arena is a segment in the address space of a process. When that process creates a new process using **sproc()**, the child process usually shares the same address space (see the sproc(2) reference page). The child process has access to the arena segment on the same basis as the parent process. However, the child process needs to join the arena formally.

The child process should join the arena by calling **usadd()**, passing the address of the arena. The child should test the return code of this function, since it can reflect an error in either of two cases:

- The arena has not been created, or an incorrect arena address was passed.
- The arena was not configured to allow enough using processes, and no more users can be allowed.

A child process can join an arena automatically, simply by using a semaphore, lock, or barrier that is allocated within that arena. These function calls perform an automatic call to **usadd()**. However, they can also encounter the error that too many processes are already using the arena. It is best for the child process to check for this condition with an explicit call to **usadd()**.

Allocating in a Shared Arena

Allocating shared memory from a shared arena is much like the regular process of allocating memory using the **malloc()** and **free()** library routines. A full set of memory-management functions is available for use with shared arenas, including:

`void *usmalloc (size_t` *size*`, usptr_t *`*handle*`);`	Allocate *size* bytes in arena **handle*.
`void *usmemalign (size_t` *align*`, size_t` *size*`, usptr_t *`*handle*`);`	Allocate *size* bytes on an *align* boundary in arena **handle*.
`void *uscalloc (size_t` *nelem*`, size_t` *elsize*`, usptr_t *`*handle*`);`	Allocate *nelem* zero-filled adjacent units of size *elsize* in arena **handle*.
`void *usrealloc (void *`*ptr*`, size_t` *size*`, usptr_t *`*handle*`);`	Change the size of object **ptr* in arena **handle* to *size*.
`void usfree (void *`*ptr*`, usptr_t *`*handle*`);`	Release object at **ptr* in arena **handle*.

See the usmalloc(3P) reference page for details and additional functions.

The address of an object allocated using **usmalloc()** or a related function is a valid address in any process that is attached to the shared arena. If the address is passed to a process that has not attached the arena, the address is not valid for that process and its use causes a SIGSEGV.

The **usmalloc()** family of functions is based on the arena-allocation function family described in the amalloc(3P) reference page. The **usmallopt()** function is the same as the **amallopt()** function, and both provide several options for modifying the memory allocation methods in a particular arena. In a similar way, **usmallinfo()** is the same as **amallinfo()**, and both return detailed statistics on usage of memory allocation in one arena.

Exchanging the First Datum

The processes using a shared arena frequently need to locate some fundamental data structure that has been allocated within the arena. For example, the parent process might create a data structure in the arena, and fill it with pointers to other objects within the arena. Any process starting to use the arena needs the address of the base structure in order to find all the other objects.

The shared arena has a special one-pointer field for storing such a basic address. This area is accessed using the following calls:

`void usputinfo (usptr_t *`*handle*`, void *`*info*`);`	Set the shared pointer field of arena **handle* to *info*.
`void *usgetinfo (usptr_t *`*handle*`);`	Return the contents of the shared pointer field of arena **handle*.
`int uscasinfo (usptr_t *`*handle*`, void *`*old*`, void *`*new*`);`	Change the contents of the shared pointer field in arena **handle*. Returns 1 when the contents matched *old* and new *was* stored.

The purpose of **uscasinfo()** is to change the contents of the field in an atomic fashion, avoiding any race condition between concurrent processes in a multiprocessor. All three functions are discussed in detail in the usputinfo(3P) reference page.

Tip: The data type of the shared pointer field is void*, a 64-bit value when the program is compiled to the 64-bit model. If you need to cast the value to an integer, use the type *__psint_t*, a pointer-sized integer in any model.

Often, the parent process creates and initializes the arena before it creates any of the child processes that will share the arena. In this case, you expect no race conditions. The parent can set the shared pointer using **usputinfo()**. Each child process can fetch the value with **usgetinfo()**.

In the less-common case when an arena is shared by unrelated processes, each process that calls **usinit()** might be the first one to create the arena, or might not. If the calling process is the first, it should initialize the basic contents and set the shared pointer. If it is not the first, it should use the initialized contents that another process has already prepared. This problem is resolved with **uscasinfo()**, as sketched by the code in Example 2-2.

Example 2-2 Setting Up an Arena with uscasinfo()

```
typedef struct arenaStuff {
   ulock_t   updateLock; /* exclusive use of this structure */
   short     joinedProcs; /* number of processes joined */
   ...pointers to other things allocated by setUpArena()...
} arenaStuff_t;
/*
|| The following function preforms the one-time setup of the
|| arenaStuff structure. It assumes that updateLock is held.
*/
extern void
setUpArena(usptr_t *arena, arenaStuff_t *stuff);
/*
|| The following function joins a specified arena, creating it
|| and initializing it if necessary.  It could be extended with
|| values to pass to usconfig(3) before the arena is created.
*/
usptr_t*
joinArena(char *arenaPath)
{
   usptr_t *arena;
   arenaStuff_t *stuff;
   int ret;
   /*
   || Join the arena, creating it if necessary. Exit on error.
   */
   if (!arena = usinit(arenaPath))
   {
      perror("usinit");
      return arena;
   }
   /*
   || Do the following as many times as necessary until the arena
   || has been initialized.
   */
   for(ret=0; !ret; )
   {
      if (stuff = (arenaStuff_t *)usgetinfo(arena))
      {
         /*
         || Another process has created the arena, and either has
         || initialized it or is initializing it right now. Acquire
         || the lock, which will block us until initializing is done.
         */
```

```
            ussetlock(stuff->updateLock);
            /* here do anything needing exclusive use of arena */
            ++stuff->joinedProcs; /* another process has joined */
            usunsetlock(stuff->updateLock); /* release arena */
            ret = 1; /* end the loop */
         }
         else
         {
            /*
            || This process appears to be first to call usinit().
            || Allocate an arenaStuff structure with its updateLock
            || already held and 1 process joined, and try to swap it
            || into place as the active one. We expect no errors
            || in setting up arenaStuff. If one occurs, the arena is
            || simply unusable, and we return a NULL to the caller.
            */
            if (! (stuff = usmalloc(sizeof(arenaStuff_t),arena) ) )
               return stuff; /* should never occur */
            if (! (stuff->updateLock = usnewlock(arena) ) );
               return (usptr_t*)0; /* should never occur */
            if (! uscsetlock(stuff->updateLock, 1) )
               return (usptr_t*)0; /* should never occur */
            stuff->joinedProcs = 1;
            if (ret = uscasinfo(arena,0,stuff))
            {
               /*
               || Our arenaStuff is now installed. Initialize it.
               || The loop ends because ret is now nonzero.
               */
               setUpArena(arena,stuff);
               usunsetlock(stuff->updateLock);
            }
            else
            {
               /*
               || uscasinfo() either did not find a current value of 0
               || (indicates a race with another process executing this
               || code) or it failed for some other reason. In any case,
               || release allocated stuff and repeat the loop (ret==0).
               */
               usfreelock(stuff->updatelock,arena);
               usfree(stuff,arena);
            }
         } /* usgetinfo returned 0 */
      } /* while uscasinfo swap fails */
```

```
    /* arena->initialized arena, updateLock not held */
    return arena;
}
```

Example 2-2 assumes that everything allocated in the arena will be accessed through a collection of pointers, *arenaStuff*. The two problems to be solved are:

- Which asynchronous process is the first to call **usinit()**, and therefore should allocate *arenaStuff* and initialize it with pointers to other objects?
- How can the second and subsequent processes know when the initialization of *arenaStuff* (which might take some time) is complete and the arena is completely ready for use?

The solution in Example 2-2 is based on the discussion in the uscasinfo(3P) reference page. Each process calls function **joinArena()**. If a call to **usgetinfo()** returns nonzero, it is the address of an *arenaStuff_t* that has been allocated by some other process. Possibly that process is concurrently executing, initializing the arena. The current process waits until the lock in the *arenaStuff_t* is released. On return from the **ussetlock()** call, the process has exclusive use of *arenaStuff* until it releases the lock. It uses this exclusive control to increment the count of processes using the arena.

When **usgetinfo()** returns 0, the calling process is probably the first to create the arena, so it allocates an *arenaStuff* structure, and also allocates the essential lock and puts it in a locked state. Then it calls **uscasinfo()** to swap the *arenaStuff* address for the expected value of 0. When the swap succeeds, the process completes initializing the arena and releases the lock.

The call to **uscasinfo()** could fail if, between the time the process receives a 0 from **usgetinfo()** and the time it calls **uscasinfo()**, another process executes this same code and installs its own *arenaStuff*. The process handles this unusual event by releasing the items it allocated and repeating the whole process.

When unrelated processes join an arena with code like that shown in Example 2-2, they should terminate their use of the arena with code similar to Example 2-3.

Example 2-3 Resigning From an Arena

```
/*
|| The following function reverses the operation of joinArena.
|| Even if the calling process is the last one to hold the arena,
|| nothing drastic is done. This is because it is impossible to
|| perform {usinit(); usgetinfo(); ussetlock();} as an atomic
|| sequence.  Once an arena comes into being it must remain
|| usable until the entire application shuts down. Unlinking the
|| arena file can be the last thing that main() does.
*/
void
resignArena(usptr_t *arena)
{
   arenaStuff_t *stuff = (arenaStuff_t *)usgetinfo(arena);
   ussetlock(stuff->updateLock);
   -- stuff->joinedProcs;
   usunsetlock(stuff->updateLock);
}
```

It might seem that, when the function **resignArena()** in Example 2-3 finds that it has reduced the *joinedProcs* count to 0, it could deinitialize the arena, for example unlinking the file on which the arena is based. This is not a good idea because of the remote chance of the following sequence of events:

1. Process A executes **joinArena()**, initializing the arena.
2. Unrelated process B executes **joinArena()** through the **usinit()** call, but is suspended for a higher-priority process before executing **usgetinfo()**.
3. Process A detects some error unrelated to arena use, and as part of termination, calls **resignArena()**.
4. Process B resumes execution with the call to **usgetinfo()**.

If the **resignArena()** function did something irrevocable, such as unlinking or truncating the arena file, it would leave process B in an unexpected state.

Using IRIX Semaphores

A *semaphore* is an integer count that is accessed atomically using two operations conventionally called P and V:

- The P operation (mnemonic de*p*lete) decrements the count. If the result is not negative, the operation succeeds and returns. If the result is negative, the operation suspends the calling process until the count has been made nonnegative by another process doing a V operation.
- The V operation (mnemonic re*viv*e) increments the count. If this changes the value from negative to nonnegative, one process that was waiting in a P operation is unblocked.

Two kinds of semaphores are supported: normal and polled.

Creating Normal Semaphores

The following functions are provided for managing normal semaphores:

usnewsema(3P)	Allocate a semaphore and give it an initial value.
usfreesema(3P)	Release a semaphore (does not release any process waiting on the semaphore).
usinitsema(3P)	Reset a semaphore value and its metering information (does not release any process waiting on the semaphore).
usctlsema(3P)	Set and reset semaphore metering information and other attributes.
usdumpsema(3P)	Dump semaphore metering information to a file.

To allocate a new shared-arena semaphore and set its initial value, call **usnewsema()**. Use **usctlsema()** to enable recursive use of the semaphore and to enable the collection of metering information. You can use the metering information to find out whether a semaphore is a bottleneck or not.

Tip: When reading the reference pages cited above, notice that **usnewsema()** returns the address of a *usema_t* object, and all the other functions take the address of a *usema_t*. That is, *usema_t* represents the type of the semaphore itself, and you refer to a semaphore by its address. This is different from locks, which are discussed later in this chapter.

Creating Polled Semaphores

A polled semaphore differs from a normal semaphore in the P operation. When decrementing the semaphore value produces a negative number, the calling process is not blocked; instead, it receives a return code. The process then has to include the address of the semaphore in the list of events passed to **poll()** (see the poll(2) reference page). The V operation, applied to a polled semaphore, does not release a block process but rather causes a **poll()** operation to end.

You can use polled semaphores to integrate semaphore handling with other events such as file operations, for which you wait with **poll()**. You cannot combine the use of normal semaphores with the use of polled devices, since one process cannot wait in a **poll()** call and in a **uspsema()** call at the same time. The following functions are supplied for creating and controlling polled semaphores:

usnewpollsema(3P)	Create a polled semaphore and give it an initial value.
usopenpollsema(3P)	Assign a file descriptor to a polled semaphore. The file descriptor can be passed to **poll()** or **select()**. This must be done before the semaphore can be used.
usclosepollsema(3P)	Release a file descriptor assigned with **usopenpollsema()**.
usfreepollsema(3P)	Release the space occupied by a polled semaphore and invalidate any file descriptors assigned to it.

Using Semaphores

The following functions are provided for semaphore operations:

uspsema(3P)	Perform the P operation on either type of semaphore.
usvsema(3P)	Perform the V operation on either type of semaphore.
ustestsema(3P)	Return the current (instantaneous) value of a semaphore.
uscpsema(3P)	Perform the P operation only if the resulting count will be nonnegative.
usinitsema(3P)	Reset a semaphore value and its metering information (does not release any process waiting on the semaphore).
usctlsema(3P)	Set and reset semaphore metering information and other attributes.
usdumpsema(3P)	Dump semaphore metering information to a file.

To perform the P operation on a semaphore of either type, use **uspsema()**. When the decremented semaphore value is nonnegative the function returns 0. The action when the decremented count would be negative differs between the polled and normal semaphores (see the uspsema(3P) reference page):

- When a normal semaphore count remains or becomes negative, the calling process is blocked; the function does not return until the count is nonnegative.
- When a polled semaphore count remains or becomes negative, the function returns 0 and the calling process must use **poll()** to find out when it becomes nonnegative.

To perform the V operation on a semaphore of either type, call **usvsema()**.

The **uscpsema()** function provides a conditional P operation: it performs a P operation on the semaphore only if it can do so without making the value negative. The **ustestsema()** function returns the current value of the semaphore—which of course is immediately out of date.

The **usinitsema()** function reinitializes the semaphore to a specified value. Note that if you reinitialize a semaphore on which processes are waiting, the processes continues to wait. You should reinitialize a semaphore only in unusual circumstances.

You can call **usctlsema()** to enable the keeping of either metering information—cumulative counts of usage—or a history trace. The metering information shows whether a semaphore is a bottleneck in the program's operations. The history trace can be used to analyze bugs.

Using Locks

Locks are like semaphores, but where semaphores are generalized to manage any countable resource, locks are optimized for mutual exclusion from a single resource. Instead of having an integer value, a lock has a binary value: it is either locked or unlocked.

The way that locks are implemented depends upon the hardware architecture of the computer using them. On multiprocessor computers, locks are busy-wait locks, so the processor continually tries to acquire the lock until it succeeds. This implementation makes sense only on multiprocessor systems, in which one processor can release the lock while another processor is "spinning," trying to acquire the lock. On single processor systems, locks are implemented using the same algorithm as semaphores—that is, processes waiting to acquire a lock may be put to sleep until the lock is released by another process.

The following functions are provided for creating and controlling locks:

usnewlock(3P)	Allocate a lock in a specified arena.
usfreelock(3P)	Release lock memory (does not release any process waiting on the lock).
usinitlock(3P)	Reset a lock and its metering information (does not release any process waiting on the lock).
usctllock(3P)	Fetch and reset semaphore metering information or debugging information.
usdumplock(3P)	Dump lock metering information to a file.

You decide whether the locks in an arena will have metering information or not. You specify this before creating the arena, to **usconfig()** (see "Initializing a Shared Arena" on page 36). When metering information is enabled, you can retrieve it at any time to find out whether a lock is a bottleneck in a program.

The following functions are provided for using locks:

ussetlock(3P)	Seize a lock, suspending the callerk if necessary, until the lock is available.
usunsetlock(3P)	Release a lock, making it available for other processes.
uscsetlock(3P)	Seize a lock if it is available; otherwise return a 1.
uswsetlock(3P)	Seize a lock, suspending the caller if necessary; takes a specified number of spins as an argument.
ustestlock(3P)	Test a lock, returning 0 if it is instantaneously available and 1 if it is not available.

On uniprocessors, none of the functions **us[c,w]setlock()** spin; if the lock is available they return immediately, and if it is not, they give up the CPU. On multiprocessors, **ussettlock()** spins for a default number of times before it suspends the process. The function **uswsetlock()** is the same except that you can specify the number of spins.

Tip: When reading the reference pages cited above, notice that **usnewlock()** returns a *ulock_t* object, which is simply a pointer. All the functions that operate on locks take a *ulock_t* object—not a pointer to a *ulock_t*. That is, the *ulock_t* type represents a handle or reference to a lock, not a lock itself. This differs from the treatment of semaphores, which is described in a preceding topic.

A process can call **usunsetlock()** on a lock that is either not locked or locked by another process. In either case, the lock is unlocked. "Double tripping"—calling a set-lock function twice with the same lock—is also permissible. The caller blocks until another process unsets the lock.

Using Barriers

Barriers provide a convenient way of synchronizing parallel processes on multiprocessor systems.

To understand barriers, think of the times when you have planned to go to lunch with other people at your workplace. You agree to meet in the lobby of the building. Some of your coworkers reach the lobby early, and others arrive later. One comes running in last, apologizing. Finally, when all of you have gathered and you know that everyone is ready, you all leave the building in a group.

A barrier is the software equivalent of the lobby where you waited. A group of processes are going to work on a problem. None should start until all the data has been initialized. However, starting each process is part of the initialization, and they cannot all be started at the same time. Each process must be created; each must join an arena and perhaps open a file; and you cannot predict when they will all be ready. To coordinate them, you create a barrier. Each process, when it is ready to start the main operation, calls **barrier()**, passing the address of the barrier and the number of processes that will meet. When that many processes have called **barrier()**, all of them are released to begin execution.

The following functions are provided to manage and use barriers:

new_barrier(3P)	Allocate and initialize a barrier in a specified arena.
free_barrier(3P)	Release the storage associated with a barrier.
barrier(3P)	Wait at a barrier until a specified number of processes have gathered.
init_barrier(3P)	Reinitialize a barrier (does not release any processes waiting).

The main process allocates a barrier in some arena. To use the barrier, each process calls **barrier()**, passing the number of processes that are supposed to meet before proceeding. In a multiprocessor, each arriving process passes time by spinning.

Note: The **barrier()** function assumes that it is used on a multiprocessor. It always passes time by spinning in an empty loop. When used on a uniprocessor (or when used on a multiprocessor with fewer available CPUs than barrier processes), a call to **barrier(*n*)** can be quite inefficient. The waiting functions spin until each in turn uses up its time-slice. In general it is not a good idea to use **barrier()** except in a multiprocessor with a number of CPUs approximately equal to the number of coordinating processes.

System V IPC

This section covers System V IPC and includes:

- “System V IPC Overview”
- “System V Messages”
- “Controlling Message Queues: msgctl()”
- “Operations for Messages: msgsnd() and msgrcv()”
- “System V Semaphores”
- “System V Shared Memory”

System V IPC Overview

System V IPC comprises three interprocess communication mechanisms:

- *Messages* allow processes to send and receive buffers full of data.
- *Semaphores* allow processes to turn on and off a set of flags.
- *Shared memory* gives multiple processes access to the same data area in memory.

These are IRIX system functions, implemented by calls to the kernel.

System V Messages

The message mechanism allows processes to exchange data stored in buffers. This data is transmitted between processes in discrete units called messages. Processes using this type of IPC can perform two operations: sending messages and receiving messages.

Before a process can send or receive messages, the process must request that the operating system generate a new *message queue* (the mechanism used to control and keep track of messages) and an associated data structure. A process makes this request by using the **msgget()** system call. The requesting process becomes the owner and creator of the resulting message queue, and specifies the initial operation permissions for all processes that might use that queue (including itself). Subsequently, the owning process can relinquish ownership or change the operation permissions using the **msgctl()** system call. However, the creator remains the creator as long as the queue exists. Other processes with permission can use **msgctl()** to perform various other control functions, as described in "Controlling Message Queues: msgctl()."

A process that is attempting to send a message can suspend execution temporarily in order to wait until the process that is to receive the message is ready; similarly, a receiving process can suspend execution until the sending process is ready. Message operations that suspend execution in this fashion are called "blocking message operations." A process that specifies that its execution is not to be suspended—that is, a process that does not wait for communication if such is not immediately available—is performing a "nonblocking message operation."

A blocking message operation can be told to suspend a calling process until one of three conditions occurs:

- The operation is successful.
- The operation receives a signal.
- The message queue is removed.

To request a message operation, the calling process passes arguments to a system call, and the system call attempts to perform its function. If the system call is successful, it returns its results. Otherwise, it returns a known error code (-1), and an external error variable, *errno*, is set accordingly.

Message Structures

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 reference the associated message queue and data structure.

Each message queue has a data structure associated with it. This data structure is declared as *struct msqid_ds* in the */usr/include/sys/msg.h* header file and contains the following fields:

msg_perm	Permisssions information as *struct ipc_perm*
msg_first, *msg_last*	Pointers to queue of message structures
msg_cbytes	Number of bytes on the queue
msg_qnum	Number of messages on the queue
msg_qbytes	Maximum number of bytes on the queue
msg_lspid	Process ID (PID) of last message sender
msg_lrpid	PID of last message receiver
msg_stime	Last message sent time
msg_rtime	Last message receive time.
msg_ctime	Last change time

The permissions structure *msg_perm* is based on another structure called *ipc_perm*, which is also used as a template for permissions for other forms of IPC. The *ipc_perm* data structure format can be found in the */usr/include/sys/ipc.h* header file.

The message queue points to a chain of header structures that define the messages in the queue. This per-message header is declared as *struct msg* in */usr/include/msg.h*, and includes the following fields:

msg_next	Pointer to next message in the queue
msg_type	Message type code
msg_ts	Text size of message
msg_spot	Message text address

Message Operations Overview

The following functions are supplied to manage messages and message queues:

msgget(2)	Get access to a message queue.
msgctl(2)	Get information about a message queue; change ownership of a queue; or destroy the queue.
msgsnd(2)	Send a message to a queue.
msgrcv(2)	Receive a message from a queue.

The **msgget()** system call receives an argument *msgflg*, which can be set to indicate various flags. When only the IPC_CREAT flag is set in *msgflg*, **msgget()** performs one of two tasks:

- It gets a new *msqid* and creates an associated message queue and data structure for it; or
- It returns 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*, unless some system tuning parameter (such as the maximum allowable number of message queues) would be exceeded.

Instead of requesting a specific *key* number, you may indicate a *key* of value zero, which is known as the private *key* (the constant IPC_PRIVATE is defined to be zero). When you specify IPC_PRIVATE for the *key* value, a new *msqid* is always returned with an associated message queue and data structure created for it unless a system tuning parameter would be exceeded. When the *ipcs* command is performed, the KEY field for the *msqid* is all zeros, for security reasons.

For the second task, if a *msqid* exists for the *key* specified, the value of the existing *msqid* is returned. If you do not want an existing *msqid* returned, you can specify a control command (IPC_EXCL) in the *msgflg* argument passed to the system call. The details of using this system call are discussed in “Getting Message Queues With msgget()”in 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 “Controlling Message Queues: msgctl().” 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 and message control can be used.

The available message operations are sending and receiving. System calls are provided for these operations: **msgsnd()** and **msgrcv()**, respectively. Refer to “Operations for Messages: msgsnd() and msgrcv(),” 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 sample fields of the data structure associated with a message queue
- to change access permissions for a message queue
- to change the size (*msg_qbytes*) of the message queue for a particular queue
- to remove a particular *msqid* from the UNIX operating system along with its associated message queue and data structure

Refer to “Controlling Message Queues: msgctl()” for details of the **msgctl()** system call.

Getting Message Queues With msgget()

This section details the **msgget()** system call and provides an example program illustrating its use. The prototype of the function is as follows (see also the msgget(2) reference page):

`int msgget (key_t key, int msgflg)`

The type of the *key* parameter, *key_t*, is defined in the *types.h* header file to be equivalent to an integer.

Upon successful completion, **msgget()** returns a message queue identifier. A new *msqid* with an associated message queue and data structure is provided if either of the following is true:

- *key* is equal to IPC_PRIVATE.
- *key* is a unique hexadecimal integer, and IPC_CREAT is set in *msgflg.*

The value passed to the *msgflg* argument is an integer that specifies access permissions, execution modes, and control fields (commands). Access permissions determine the read/write attributes, while execution modes determine the user/group/other attributes of the *msgflg* argument. The permissions and modes are collectively referred to as "operation permissions." Table 2-2 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 2-2 Message Queue Operation Permissions Codes

Operation Permissions	Octal Value
Read by Owner (MSG_R)	00400
Write by Owner (MSG_W)	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. For instance, if you want a message queue to be readable by its owner and both readable and writable by others, use the code value 00406 (00400 plus 00004 plus 00002).

The constants MSG_R and MSG_W, defined in the *msg.h* header file, can be used instead of 00400 and 00200, respectively.

Control commands are constants defined in the *ipc.h* header file. See Table 2-3, which contains the names of the constants that apply to the **msgget()** system call and their values.

Table 2-3 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for *msgflg* is therefore a combination of operation permissions and control commands. To accomplish this combination, perform a bitwise OR (|) on the flags with the operation permissions. The bit positions and values for the control commands in relation to those of the operation permissions make this possible.

Two examples:

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

The **msgget()** system call attempts to return a new *msqid* if either of the following conditions is true:

- *key* is equal to IPC_PRIVATE.
- *key* does not already have an *msqid* associated with it, and IPC_CREAT is set in *msgflg*.

To satisfy the first condition, simply pass IPC_PRIVATE as the *key* argument when calling **msgget()**:

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

The second condition is satisfied if the value for *key* is not already associated with an *msqid* and a bitwise AND of *msgflg* and IPC_CREAT gives "true" (1). This means that the given *key* is not currently being used to refer to any message queue on the computer the program is running on, and that the IPC_CREAT flag is set in *msgflg*.

Note: The system tuning parameter MSGMNI determines the maximum number of unique message queues (*msqid*s) in the UNIX operating system. Attempting to exceed MSGMNI always causes a failure.

IPC_EXCL is another control flag 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 intro(2) reference page for details about the data structures associated with a message queue. The specific failure conditions, with error names, are listed there as well.

Message Queue Example Program

The example program in this section (Example 2-4) is a menu-driven program that exercises all possible combinations of the **msgget()** system call.

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

This program begins (lines 4-8) by including the required header files as specified by the msgget(2) reference page. Note that the *errno.h* header file is included instead of declaring *errno* as an external variable; either method works.

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. This choice of names makes the program more readable, and it is perfectly legal since the variables are local to this program.

The variables in this program and their purposes are as follows:

key — The value for the desired message queue key

opperm — The desired operation permissions

opperm_flags — The combination from the logical ORing of the *opperm* and *flags* variables; used in the system call to pass the *msgflg* argument

msqid — The message queue identification number for a successful system call or the error code (-1) for an unsuccessful one

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and finally for the control command combinations (flags), which are selected from a menu (lines 15-32). All possible combinations are allowed even though they might not be 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 in the *opperm_flags* variable (lines 36-51).

The system call is made next, and the result is stored in 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).

Example 2-4 msgget() System Call Example

```
1  /*This is a program to illustrate the capabilities of
2   *the msgget() (message-get) system call.
3   */
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 program*/
10 main()
11 {
12     key_t key;
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 ");
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");
```

```
    printf("IPC_CREAT = 1\n");
    printf("IPC_EXCL = 2\n");
    printf("IPC_CREAT and IPC_EXCL = 3\n");
    printf(" Flags = ");
    /*Get the flag(s) to be set.*/
    scanf("%d", &flags);
    /*Check the values.*/
    printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
        key, opperm, flags);
    /*Incorporate the control fields (flags) with
      the operation permissions*/
    switch (flags)
    {
    case 0: /*No flags are to be set.*/
        opperm_flags = (opperm | 0);
        break;
    case 1: /*Set the IPC_CREAT flag.*/
        opperm_flags = (opperm | IPC_CREAT);
        break;
    case 2: /*Set the IPC_EXCL flag.*/
        opperm_flags = (opperm | IPC_EXCL);
        break;
    case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
        opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
    }
    /*Call the msgget() system call.*/
    msqid = msgget (key, opperm_flags);
    /*Perform the following if the call failed.*/
    if(msqid == -1)
    {
        printf ("\nThe msgget system call failed!\n");
        printf ("The error number was %d.\n", errno);
    }
    /*Return the msqid upon successful completion.*/
    else
        printf ("\nThe msqid is %d.\n", msqid);
    exit(0);
}
```

Controlling Message Queues: msgctl()

This section gives a detailed description of using the **msgctl()** system call and an example program that exercises its capabilities. The prototype of **msgctl()** is as follows (see also the msgctl(2) reference page):

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

On successful completion, the function returns zero; when unsuccessful, it returns a -1.

The *msqid* variable must be a nonnegative, integer value previously established as a message queue by using **msgget()**.

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 superuser 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 trouble understanding the logic manipulations in this program, read the "Getting Message Queues With msgget()" section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Example 2-5) is a menu-driven program that exercises all possible combinations of the **msgctl()** system call.

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

This program begins (lines 5-9) by including the required header files as specified by the msgctl(2) reference page. 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. The variables declared for this program and their purposes are as follows:

uid	The IPC_SET value for the effective user identification
gid	The IPC_SET value for the effective group identification
mode	The IPC_SET value for the operation permissions
bytes	The IPC_SET value for the number of bytes in the message queue (*msg_qbytes*)
rtrn	The return integer value from the system call
msqid	The message queue identifier
command	The code for the desired control command
choice	A variable used to determine which member is to be changed for the IPC_SET control command
msqid_ds	A variable that receives the specified message queue identifier's data structure when an IPC_STAT control command is performed
buf	A variable that bases 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 of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid message queue identifier, which is stored in 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 in the command variable. The code is tested to determine the control command for subsequent processing.

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

If the IPC_SET control command is selected (case 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 in the *choice* variable (line 60). Now, depending upon the member picked, the program prompts for the new value (lines 66-95). The value is placed in the appropriate member in the user memory area data structure, and the system call is made (lines 96-98). Depending upon 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 operating system. Note that the **buf* pointer is not required as an argument to perform this control command, and its value can be zero or NULL. Depending upon 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 in Example 2-5.

Example 2-5 msgctl() System Call Example

```
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>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
 /*Start of main C language program*/
 main()
 {
   extern int errno;
   int uid, gid, mode, bytes;
   int rtrn, msqid, command, choice;
   struct msqid_ds msqid_ds, *buf;
   buf = &msqid_ds;
   /*Get the msqid, and command.*/
   printf("Enter the msqid = ");
   scanf("%d", &msqid);
   printf("\nEnter the number for\n");
   printf("the desired command:\n");
   printf("IPC_STAT = 1\n"); printf("IPC_SET = 2\n");
   printf("IPC_RMID = 3\n");
   printf("Entry = ");
   scanf("%d", &command);
   /*Check the values.*/
  printf (\nmsqid =%d, command = %\n", msqid, command);
   switch (command)
   {
   case 1: /*Use msgctl() to duplicate
     the data structure for
       msqid in the msqid_ds area pointed
       to by buf and then print it out.*/
     rtrn = msgctl(msqid, IPC_STAT,
      buf);
     printf ("\nThe USER ID = %d\n",
       buf->msg_perm.uid);
     printf ("The GROUP ID = %d\n",
       buf->msg_perm.gid);
     printf ("The operation permissions = 0%o\n",
       buf->msg_perm.mode);
     printf ("The msg_qbytes = %d\n",
       buf->msg_qbytes);
     break;
    case 2: /*Select and change the desired
          member(s) of the data structure.*/
       /*Get the original data for this msqid
          data structure first.*/
       rtrn = msgctl(msqid, IPC_STAT, buf);
       printf("\nEnter the number for the\n");
```

```
        printf("member to be changed:\n");
        printf("msg_perm.uid = 1\n");
        printf("msg_perm.gid = 2\n");
        printf("msg_perm.mode = 3\n");
        printf("msg_qbytes = 4\n");
        printf("Entry = ");
        scanf("%d", &choice);
        /*Only one choice is allowed per pass
           as an illegal entry will
           cause repetitive failures until
           msqid_ds is updated with
           IPC_STAT.*/
        switch(choice){
        case 1:
          printf("\nEnter USER ID = ");
          scanf ("%d", &uid);
          buf->msg_perm.uid = uid;
          printf("\nUSER ID = %d\n",
            buf->msg_perm.uid);
          break;
        case 2:
          printf("\nEnter GROUP ID = ");
          scanf("%d", &gid);
          buf->msg_perm.gid = gid;
          printf("\nGROUP ID = %d\n",
            buf->msg_perm.gid);
          break;
        case 3:
          printf("\nEnter MODE = ");
          scanf("%o", &mode);
          buf->msg_perm.mode = mode;
          printf("\nMODE = 0%o\n",
            buf->msg_perm.mode);
          break;
        case 4:
          printf("\nEnter msq_bytes = ");
          scanf("%d", &bytes);
          buf->msg_qbytes = bytes;
          printf("\nmsg_qbytes = %d\n",
          buf->msg_qbytes);
          break;
        }
        /*Do the change.*/
        rtrn = msgctl(msqid, IPC_SET,
          buf);
```

```
     break;
   case 3: /*Remove the msqid along with its
       associated message queue
       and data structure.*/
     rtrn = msgctl(msqid, IPC_RMID, NULL);
   }
   /*Perform the following if call is unsuccessful.*/
   if(rtrn == -1)
   {
     printf ("\nThe msgctl system call failed!\n");
     printf ("The error number = %d\n", errno);
   }
   /*Return the msqid upon successful completion.*/
   else
     printf ("\nMsgctl was successful for msqid = %d\n",
       msqid);
   exit (0);
}
```

Operations for Messages: msgsnd() and msgrcv()

This section describes the **msgsnd()** and **msgrcv()** system calls and presents an example program that exercises all of their capabilities.

The prototypes of **msgsnd()** and **msgrcv()** are as follows (see also the reference page, which can be accessed as msgsnd(2), msgrcv(2), or as msgops(2)):

```
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);
```

Sending a Message

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

Upon successful completion, a zero value is returned; when unsuccessful, **msgsnd()** returns a -1.

The *msqid* argument must be a nonnegative integer established as a message queue id by **msgget()**.

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 tuning parameter.

The *msg_qbytes* data structure member can be lowered from MSGMNB by using the **msgctl()** IPC_SET control command, but only the superuser 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 occurs if the total number of bytes allowed on the specified message queue are in use (*msg_qbytes* or MSGMNB), or the total system-wide number of messages on all queues is equal to the system imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call fails and returns -1.

Further details of this system call are discussed in the example program for it. If you don't understand the logic manipulations in this program, read the "Getting Message Queues With msgget()" section of this chapter; it goes into more detail than 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. Upon successful completion, **msgrcv()** returns a value equal to the number of bytes received returned. When unsuccessful it returns -1.

The *msqid* argument must be a nonnegative integer established as a message queue id by **msgget()**.

The *msgp* argument is a pointer to a structure in the user memory area that receives 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 size of 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 *msgtyp* 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 occurs 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 fails immediately when a message of the desired type is not on the queue. *msgflg* can also specify that the system call fails 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 don't understand the logic manipulations in this program, read the "Getting Message Queues With msgget()" section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Example 2-6) is a menu-driven program that exercises all possible combinations of the **msgsnd()** and **msgrcv()** system calls.

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

This program begins (lines 5-9) by including the required header files as specified by the msgop(2) reference page. Note that in this program, *errno* is declared as an external variable. Therefore, the *errno.h* header file does not have to be included.

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

The variables declared for this program and their purposes are as follows: :

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

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

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

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

c — Used to receive the input character from the **getchar()** function (line 50).

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

flags — Used to store the code of the IPC_NOWAIT or MSG_NOERROR flags for the **msgrcv()** system call (line 117).

choice — Used to store the code for sending or receiving (line 30).

rtrn — Used to store the return values from all system calls.

msqid — Used to store and pass the desired message queue identifier for both system calls.

msgsz — Used to store and pass the size of the message to be sent or received.

msgflg — Used to pass the value of flag for sending or the value of flags for receiving.

msgtyp — Used for specifying the message type for sending, or used to pick a message type for receiving.

Note that a *msqid_ds* data structure is set up in the program (line 21) with a pointer that is initialized to point to it (line 22); this allows the data structure members that are affected by message operations to be observed. They are observed by using the **msgctl()** system call (with IPC_STAT) to get them so the program can 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 in the *choice* variable (lines 23-30). Depending upon the code, the program proceeds as in the following **msgsnd()** or **msgrcv()** sections.

Example Use of 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 in 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 continues until an end of file is reached (for the **getchar()** function, this is a <Ctrl-D> immediately following a carriage return). When this happens, the size of the message is determined by adding 1 to the *i* counter (lines 52, 53) as the is stored beginning in the zero array element of *mtext*. Keep in mind that the message also contains the terminating characters; therefore, the message appears 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 1 be entered for yes or anything else for no (lines 57-65). The result is stored in 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 then 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 (which should be zero) is printed (lines 73-76).

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

msg_qnum The total number of messages on the message queue.

msg_lspid The process identification (PID) number of the last process sending a message.

msg_stime The time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent.

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

Example Use of 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 in *msqid* (lines 100-103).

The message type is requested, and it is stored in *msgtyp* (lines 104-107).

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

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

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

When a message is successfully received, three members of the associated data structure are updated; 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 is shown in Example 2-6.

Example 2-6 msgsnd() and msgrcv() System Call Example

```
/*This is a program to illustrate
 * the message operations, msgop(),
 * system call capabilities: msgsnd() and msgrcv().
 */
 /*Include necessary header files.*/
 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/msg.h>
 struct msgbuf1 {
    long mtype;
    char mtext[8192];
} sndbuf, rcvbuf, *msgp;
  /*Start of main C language program*/
  main()
  {
    extern int errno;
    int i, c, flag, flags, choice;
    int rtrn, msqid, msgsz, msgflg;
    long mtype, msgtyp;
    struct msqid_ds msqid_ds, *buf;
    buf = &msqid_ds;
    /*Select the desired operation.*/
    printf("Enter the corresponding\n");
    printf("code to send or\n");
    printf("receive a message:\n");
    printf("Send = 1\n");
    printf("Receive = 2\n");
    printf("Entry = ");
    scanf("%d", &choice);
```

```
    if(choice == 1) /*Send a message.*/
    {
      msgp = &sndbuf; /*Point to user send structure.*/
      printf("\nEnter the msqid or\n");
      printf("the message queue to\n");
      printf("handle the message = ");
      scanf("%d", &msqid);
      /*Set the message type.*/
      printf("\nEnter a positive integer\n");
      printf("message type (long) for the\n");
      printf("message = ");
      scanf("%d", &msgtyp);
      msgp->mtype = msgtyp;
      /*Enter the message to send.*/
      printf("\nEnter a message: \n");
      /*A control-d (^d) terminates as
        EOF.*/
      /*Get each character of the message
        and put it in the mtext array.*/
      for(i = 0; ((c = getchar()) != EOF); i++)
        sndbuf.mtext[i] = c;
      /*Determine the message size.*/
      msgsz = i + 1;
      /*Echo the message to send.*/
      for(i = 0; i < msgsz; i++)
        putchar(sndbuf.mtext[i]);
      /*Set the IPC_NOWAIT flag if
        desired.*/
      printf("\nEnter a 1 if you want the\n");
      printf("the IPC_NOWAIT flag set: ");
      scanf("%d", &flag);
      if(flag == 1)
        msgflg |= IPC_NOWAIT;
      else
        msgflg = 0;
      /*Check the msgflg.*/
      printf("\nmsgflg = 0%o\n", msgflg);
      /*Send the message.*/
      rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
      if(rtrn == -1)
      printf("\nMsgsnd failed. Error = %d\n",
        errno);
      else {
        /*Print the value of test which
        should be zero for successful.*/
```

```
            printf("\nValue returned = %d\n", rtrn);
            /*Print the size of the message
            sent.*/
            printf("\nMsgsz = %d\n", msgsz);
            /*Check the data structure update.*/
            msgctl(msqid, IPC_STAT, buf);
            /*Print out the affected members.*/
            /*Print the incremented number of
              messages on the queue.*/
            printf("\nThe msg_qnum = %d\n",
              buf->msg_qnum);
            /*Print the process id of the last sender.*/
            printf("The msg_lspid = %d\n",
              buf->msg_lspid);
            /*Print the last send time.*/
            printf("The msg_stime = %d\n",
              buf->msg_stime);
        }
    }
    if(choice == 2) /*Receive a message.*/
    {
      /*Initialize the message pointer
        to the receive buffer.*/
      msgp = &rcvbuf;
      /*Specify the message queue which contains
        the desired message.*/
      printf("\nEnter the msqid = ");
      scanf("%d", &msqid);
      /*Specify the specific message on the queue
        by using its type.*/
      printf("\nEnter the msgtyp = ");
      scanf("%d", &msgtyp);
      /*Configure the control flags for the
        desired actions.*/
      printf("\nEnter the corresponding codeen");
      printf("to select the desired flags: \n");
      printf("No flags = 0\n");
      printf("MSG_NOERROR = 1\n");
      printf("IPC_NOWAIT = 2\n");
      printf("MSG_NOERROR and IPC_NOWAIT = 3\n");
      printf(" Flags = ");
      scanf("%d", &flags);
      switch(flags) {
        /*Set msgflg by ORing it with the appropriate
        flags (constants).*/
```

```
case 0:
  msgflg = 0;
  break;
case 1:
  msgflg |= MSG_NOERROR;
  break;
case 2:
  msgflg |= IPC_NOWAIT;
  break;
case 3:
  msgflg |= MSG_NOERROR | IPC_NOWAIT;
  break;
}
/*Specify the number of bytes to receive.*/
printf("\nEnter the number of bytes\n");
printf("to receive (msgsz) = ");
scanf("%d", &msgsz);
/*Check the values for the arguments.*/
printf("\nmsqid =%d\n", msqid);
printf("\nmsgtyp = %d\n", msgtyp);
printf("\nmsgsz = %d\n", msgsz);
printf("\nmsgflg = 0%o\n", msgflg);
/*Call msgrcv to receive the message.*/
rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
if(rtrn == -1) {
  printf("\nMsgrcv failed. ");
  printf("Error = %d\n", errno);
}
else {
  printf ("\nMsgctl was successful\n");
  printf("for msqid = %d\n",
    msqid);
  /*Print the number of bytes received,
    it is equal to the return
    value.*/
  printf("Bytes received = %d\n", rtrn);
  /*Print the received message.*/
  for(i = 0; i<=rtrn; i++)
    putchar(rcvbuf.mtext[i]);
}
/*Check the associated data structure.*/
msgctl(msqid, IPC_STAT, buf);
/*Print the decremented number of messages.*/
printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
/*Print the process id of the last receiver.*/
```

```
    printf("The msg_lrpid = %d\n", buf->msg_lrpid);
    /*Print the last message receive time*/
    printf("The msg_rtime = %d\n", buf->msg_rtime);
  }
}
```

System V Semaphores

Semaphore IPC allows processes to communicate through the exchange of data items called semaphores. A single semaphore is represented as a positive integer (0 through 32,767). Semaphores are usually used to manage resources; each semaphore indicates whether or not a specific data item is currently in use (and by how many different processes). Since many applications require the use of more than one semaphore, the UNIX operating system has the ability to create sets or arrays of semaphores at one time. A semaphore set can contain one or more semaphores, up to a limit set by the system administrator. (This limit, a tunable parameter called SEMMSL, has a default value of 25.) To create a set of semaphores, use the **semget()** system call.

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

Provided a process has alter permission, it can manipulate the semaphore set. Each semaphore within a set can be either increased or decreased with the **semop()** system call (see the semop(2) reference page for more information).

To increase a semaphore, pass a positive integer value of the desired magnitude to the **semop()** system call. To decrease a semaphore, pass a negative integer value of the desired magnitude.

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

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

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

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

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

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

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

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

The ability to undo operations requires the UNIX operating 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 attempts to perform its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable *errno* is set accordingly.

Using Semaphores

Before you can use semaphores, you must request a uniquely identified semaphore set and its associated data structure with a system call. The unique identifier is called the semaphore identifier (*semid*); it is used to reference a particular semaphore set and data structure.

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

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

Each semaphore set has an associated data structure. This data structure contains information related to the semaphore set:

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

The definition for the semaphore set is:

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

This definition is located in the */usr/include/sys/sem.h* header file. The structure definition for the associated semaphore data structure is also located in the same header file. Note that the *sem_perm* member of this structure uses *ipc_perm* as a template. The *ipc_perm* data structure is the same for all IPC facilities, and it is located in the */usr/include/sys/ipc.h* header file.

The **semget()** system call performs 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 tuning parameter would be exceeded.

Also, a provision exists for specifying a *key* of value zero, 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 tuning 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 "Controlling Semaphores With semctl()" for more information. 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 you do not want an existing *semid* returned, you can specify IPC_EXCL in the *semflg* argument passed to the system call. The system call fails 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 the **semget()** system call are discussed in "Getting Semaphores With semget()."

Once a uniquelyidentified semaphore set and data structure are created, use semaphore operations (**semop()**) and semaphore control (**semctl()**).

Semaphores can be increased, decreased, or tested for zero. A single system call, **semop()**, is used to perform these operations. Refer to "Operations on Semaphores: semop()" for details on this system call.

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

- to return the value of a semaphore
- to set the value of a semaphore
- to return the process identification (PID) of the last process performing an operation on a semaphore set
- to return the number of processes waiting for a semaphore value to become greater than its current value
- to return the number of processes waiting for a semaphore value to equal zero
- to get all semaphore values in a set and place them in an array in user memory

- to set all semaphore values in a semaphore set from an array of values in user memory
- to place all data structure member values and 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 operating system along with its associated data structure and semaphore set

Refer to “Controlling Semaphores With semctl()” for details of the **semctl()** system call.

Getting Semaphores With semget()

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

The function prototype found in the semget(2) reference page is as follows:

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

Type *key_t* is declared in the *types.h* header file to be an integer.

The integer returned from this system call upon successful completion is the semaphore set identifier (*semid*) discussed above. A new *semid* with an associated semaphore set and data structure is provided when either of the following conditions is true:

- *key* is equal to IPC_PRIVATE.
- *key* is passed a unique hexadecimal integer and *semflg* contains IPC_CREAT.

The value passed to the *semflg* argument specifies the following:

- access permissions to determine the read/alter attributes of the semaphores
- execution modes to determine the user, group, and public attributes
- control fields (commands)

Access permissions and execution modes are collectively referred to as “operation permissions.”

Table 2-4 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 2-4 Operation Permissions Codes

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

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read and alter by others is desired, the code value would be 00406 (00400 plus 00004 plus 00002).

Control commands constants are declared in the *ipc.h* header file. Table 2-5 contains the names of the constants that apply to the **semget()** system call along with their values.

Table 2-5 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for *semflg* is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified using a bitwise OR with the operation permissions.

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

As specified by the semget(2) reference page, success or failure of this system call depends upon the actual argument values for *key*, *nsems*, *semflg*, as limited by system tuning parameters. The system call attempts to return a new *semid* if either of the following conditions is true:

- *key* is equal to IPC_PRIVATE (0).
- *key* does not already have a *semid* associated with it and *semflg* contains IPC_CREAT.

The *key* argument can be set to IPC_PRIVATE as follows:

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

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

The second condition is satisfied if the value for *key* is not already associated with a *semid*, and the bitwise ANDing of *semflg* and IPC_CREAT is "true" (not zero). This means that the *key* is unique (not already in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag has been set (using *semflg* | IPC_CREAT). 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 tuning parameters are exceeded.

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

Example Program

The example program in this section (Example 2-7) is a menu-driven program that exercises all possible combinations of the **semget()** system call.

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

This program begins (lines 4-8) by including the required header files as specified by the semget(2) reference page. Note that the *errno.h* header file is included as opposed to declaring *errno* as an external variable (either method works).

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. The variables declared for this program and their purposes are as follows:

key	The value for the desired key
opperm	The desired operation permissions
flags	The desired control commands (flags)
opperm_flags	The combination from the logical ORing of the *opperm* and *flags* variables; used in the system call to pass the *semflg* argument
semid	The value returned by **semget()**.

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

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in 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 in *nsems*. The system call is made next, and the result is stored in the *semid* variable (lines 60, 61).

Since the *semid* variable now contains a valid semaphore set identifier or the error code (-1), it is tested to see if an error occurred (line 63). If *semid* equals -1, a message indicates that an error resulted and the external *errno* variable is displayed (lines 65, 66).

Remember that the external *errno* variable is set only 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 is shown in Example 2-7.

Example 2-7 semget() System Call Example

```
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
```

```
      the operation permissions.*/
    switch (flags)
    {
    case 0: /*No flags are to be set.*/
      opperm_flags = (opperm | 0);
      break;
    case 1: /*Set the IPC_CREAT flag.*/
      opperm_flags = (opperm | IPC_CREAT);
      break;
    case 2: /*Set the IPC_EXCL flag.*/
      opperm_flags = (opperm | IPC_EXCL);
      break;
    case 3: /*Set the IPC_CREAT and IPC_EXCL
        flags.*/
      opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
    }
    /*Get the number of semaphores for this set.*/
    printf("\nEnter the number of\n");
    printf("desired semaphores for\n");
    printf("this set (25 max) = ");
    scanf("%d", &nsems);
    /*Check the entry.*/
    printf("enNsems = %d\n", nsems);
    /*Call the semget system call.*/
    semid = semget(key, nsems, opperm_flags);
    /*Perform the following if the call is unsuccessful.*/
    if(semid == -1)
    {
      printf("The semget system call failed!\n");
      printf("The error number = %d\n", errno);
    }
    /*Return the semid upon successful completion.*/
    else
      printf("\nThe semid = %d\n", semid);
    exit(0);
  }
```

Controlling Semaphores With semctl()

This section details the **semctl()** system call and provides an example program that exercises all of its capabilities.

The prototype found in the semctl(2) reference page is as follows:

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

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

The *semid* argument must be a nonnegative, integer returned by **semget()**. The *semnum* argument selects a semaphore within a set by its number. The first semaphore in a set is number 0.

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

GETVAL	Return the value of a single semaphore within a semaphore set.
SETVAL	Set the value of a single semaphore within a semaphore set.
GETPID	Return the process identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set.
GETNCNT	Return the number of processes waiting for the value of a particular semaphore to become greater than its current value.
GETZCNT	Return the number of processes waiting for the value of a particular semaphore to be equal to zero.
GETALL	Return the values for all semaphores in a semaphore set.
SETALL	Set all semaphore values in a semaphore set.
IPC_STAT	Return the status information contained in the associated data structure for the specified *semid*, and place it in the data structure pointed to by the **buf* pointer in the user memory area; *arg.buf* is the union member that contains the value of *buf*.
IPC_SET	For the specified semaphore set (*semid*), set the effective user/group identification and operation permissions.

IPC_RMID Remove the specified (*semid*) semaphore set along with its associated data structure.

A process must have an effective user identification of owner/creator of the semaphore set, or of superuser, to perform an IPC_SET or IPC_RMID control command. Read/alter permission is required for other control commands.

Use the *arg* argument to pass the appropriate union member to the system call for the control command to be performed.

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 "Getting Semaphores With semget()"; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Example 2-8 on page 91) is a menu-driven program that exercises all possible combinations of using the **semctl()** system call. From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the semctl(2) reference page. 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. Those declared for this program and their purpose are as follows:

semid_ds The specified semaphore set identifier's data structure when an IPC_STAT control command is performed.

c The input values from the **scanf()** function (line 117) when performing a SETALL control command.

i General counter/index.

length The number of semaphores in a set (lines 97, 115).

uid The IPC_SET value for the effective user identification.

gid The IPC_SET value for the effective group identification.

mode	The IPC_SET value for the operation permissions.
rtrn	The return integer from the system call.
semid	The semaphore set ID.
semnum	The semaphore number to the system call.
cmd	The code for the desired control command.
choice	Member (*uid*, *gid*, *mode*) for the IPC_SET control command that is to be changed.
arg	The union argument to IPC_STAT, IPC_SET, GETALL, and SETALL.

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

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

First, the program prompts for a valid semaphore set identifier, which is stored in 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 in the *cmd* variable. The code is tested to determine the control command for subsequent processing.

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

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored in the *semnum* variable (line 58). Next, a message prompts for the value to which the semaphore is to be set, and it is stored as the *arg.val* member of the union (lines 59, 60). Then, the system call is performed (lines 62). Depending upon 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 upon 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 in 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 upon 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 in 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 upon 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, upon 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* (lines 97-103). The semaphores in the set are displayed on a single line, separated by a space. Depending upon 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 system call is then made (lines 120-122). Depending upon success or failure, the program returns the same messages as for GETVAL above.

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

If the IPC_SET control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 143-146). This is necessary because this example program provides for changing only one member at a time, and the **semctl()** 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 prompt for a code corresponding to the member to be changed (lines 147-153). This code is stored in the *choice* variable (line 154). Now, depending upon the member picked, the program prompts for the new value (lines 155-178). The value is placed in 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* is removed from the UNIX operating system along with its associated data structure and semaphore set. 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 is shown in Example 2-8.

Example 2-8 semctl() System Call Example

```
/*This is a program to illustrate
 *the semaphore control, semctl(),
 *system call capabilities.
 */
  /*Include necessary header files.*/
  #include <stdio.h>
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/sem.h>
  /*Start of main C language program*/
  main()
  {
    extern int errno;
   struct semid_ds semid_ds;
   int c, i, length;
   int uid, gid, mode;
   int retrn, semid, semnum, cmd, choice;
   union semun {
     int val;
     struct semid_ds *buf;
     ushort array[25];
   } arg;
   /*Initialize the data structure pointer.*/
   arg.buf = &semid_ds;
   /*Enter the semaphore ID.*/
   printf("Enter the semid = ");
   scanf("%d", &semid);
   /*Choose the desired command.*/
   printf("\nEnter the number for\n");
   printf("the desired cmd:\n");
   printf("GETVAL = 1\n");
   printf("SETVAL = 2\n");
   printf("GETPID = 3\n");
   printf("GETNCNT = 4\n");
   printf("GETZCNT = 5\n");
   printf("GETALL = 6\n");
   printf("SETALL = 7\n");
   printf("IPC_STAT = 8\n");
   printf("IPC_SET = 9\n");
   printf("IPC_RMID = 10\n");
   printf("Entry = ");
```

```
scanf("%d", &cmd);
/*Check entries.*/
printf ("\nsemid =%d, cmd = %d\n\n",
 semid, cmd);
/*Set the command and do the call.*/
switch (cmd)
{
case 1: /*Get a specified value.*/
  printf("\nEnter the semnum = ");
  scanf("%d", &semnum);
  /*Do the system call.*/
  retrn = semctl(semid, semnum, GETVAL, 0);
  printf("\nThe semval = %d\n", retrn);
  break;
case 2: /*Set a specified value.*/
  printf("\nEnter the semnum = ");
  scanf("%d", &semnum);
  printf("\nEnter the value = ");
  scanf("%d", &arg.val);
  /*Do the system call.*/
  retrn = semctl(semid, semnum, SETVAL, arg.val);
  break;
case 3: /*Get the process ID.*/
  retrn = semctl(semid, 0, GETPID, 0);
  printf("\nThe sempid = %d\n", retrn);
  break;
case 4: /*Get the number of processes
  waiting for the semaphore to
  become greater than its current
  value.*/
  printf("\nEnter the semnum = ");
  scanf("%d", &semnum);
  /*Do the system call.*/
  retrn = semctl(semid, semnum, GETNCNT, 0);
  printf("\nThe semncnt = %d", retrn);
  break;
case 5: /*Get the number of processes
  waiting for the semaphore
  value to become zero.*/
  printf("\nEnter the semnum = ");
  scanf("%d", &semnum);
  /*Do the system call.*/
  retrn = semctl(semid, semnum, GETZCNT, 0);
  printf("\nThe semzcnt = %d", retrn);
  break;
```

```
case 6: /*Get all of the semaphores.*/
  /*Get the number of semaphores in
  the semaphore set.*/
  retrn = semctl(semid, 0, IPC_STAT, arg.buf);
  length = arg.buf->sem_nsems;
  if(retrn == -1)
    goto ERROR;
  /*Get and print all semaphores in the
  specified set.*/
  retrn = semctl(semid, 0, GETALL, arg.array);
  for (i = 0; i < length; i++)
  {
    printf("%d", arg.array[i]);
    /*Seperate each
    semaphore.*/
    printf("%c", ‘ ’);
  }
  break;
case 7: /*Set all semaphores in the set.*/
  /*Get the number of semaphores in
  the set.*/
  retrn = semctl(semid, 0, IPC_STAT, arg.buf);
  length = arg.buf->sem_nsems;
  printf("Length = %den", length);
  if(retrn == -1)
    goto ERROR;
  /*Set the semaphore set values.*/
  printf("\nEnter each value:\n");
  for(i = 0; i < length ; i++)
  {
    scanf("%d", &c);
    arg.array[i] = c;
  }
  /*Do the system call.*/
  retrn = semctl(semid, 0, SETALL, arg.array);
  break;
case 8: /*Get the status for the semaphore set.*/
  /*Get and print the current status values.*/
  retrn = semctl(semid, 0, IPC_STAT, arg.buf);
  printf ("\nThe USER ID = %d\n",
    arg.buf->sem_perm.uid);
  printf ("The GROUP ID = %d\n",
    arg.buf->sem_perm.gid);
  printf ("The operation permissions = 0%o\n",
    arg.buf->sem_perm.mode);
```

```
      printf ("The number of semaphores in set = %d\n",
        arg.buf->sem_nsems);
      printf ("The last semop time = %d\n",
        arg.buf->sem_otime);
      printf ("The last change time = %d\n",
        arg.buf->sem_ctime);
      break;
    case 9: /*Select and change the desired
            member of the data structure.*/
      /*Get the current status values.*/
      retrn = semctl(semid, 0, IPC_STAT, arg.buf);
      if(retrn == -1)
        goto ERROR;
      /*Select the member to change.*/
      printf("\nEnter the number for the\n");
      printf("member to be changed:\n");
      printf("sem_perm.uid = 1\n");
      printf("sem_perm.gid = 2\n");
      printf("sem_perm.mode = 3\n");
      printf("Entry = ");
      scanf("%d", &choice);
      switch(choice){
      case 1: /*Change the user ID.*/
        printf("\nEnter USER ID = ");
        scanf ("%d", &uid);
          arg.buf->sem_perm.uid = uid;
        printf("enUSER ID = %d\n",
          arg.buf->sem_perm.uid);
        break;
      case 2: /*Change the group ID.*/
        printf("\nEnter GROUP ID = ");
        scanf("%d", &gid);
          arg.buf->sem_perm.gid = gid;
        printf("\nGROUP ID = %d\n",
          arg.buf->sem_perm.gid);
        break;
      case 3: /*Change the mode portion of
          the operation
          permissions.*/
        printf("\nEnter MODE = ");
        scanf("%o", &mode);
          arg.buf->sem_perm.mode = mode;
        printf("\nMODE = 0%o\n",
          arg.buf->sem_perm.mode);
        break;
```

```
    }
    /*Do the change.*/
    retrn = semctl(semid, 0, IPC_SET, arg.buf);
    break;
    case 10: /*Remove the semid along with its
      data structure.*/
      retrn = semctl(semid, 0, IPC_RMID, 0);
    }
    /*Perform the following if the call is unsuccessful.*/
    if(retrn == -1)
    {
  ERROR:
      printf ("\n\nThe semctl system call failed!\n");
      printf ("The error number = %d\n", errno);
      exit(0);
    }
    printf ("\n\n semctl system call was successful\n");
      printf ("for semid = %d\n", semid);
    exit (0);
}
```

Operations on Semaphores: semop()

This section details the **semop()** system call and provides an example program that exercises its capabilities.

The prototype found in the semop(2) reference page is as follows:

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

semop() returns 0 on success, or -1 on failure.

The *semid* argument must be a nonnegative integer returned by **semget()**.

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 */usr/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 tuning 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 which the operation is to be performed.

The operation to be performed is determined by the following:

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

The following operation commands (flags) can be used:

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

Example Program

The example program in this section (Example 2-9) is a menu-driven program that exercises all possible combinations of the **semop()** system call.

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

This program begins (lines 5-9) by including the required header files as specified by the semop(2) reference page. 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 purposes are as follows:

sembuf[10] — An array buffer (line 14) to contain a maximum of ten *sembuf* type structures; ten equals SEMOPM, the maximum number of operations on a semaphore set for each **semop()** 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 — The return values from the system call

flags — The code of the IPC_NOWAIT or SEM_UNDO flags for the **semop()** system call (line 60)

i — 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 — Specifies the number of semaphore operations for the system call; must be less than or equal to SEMOPM

semid — The desired semaphore set identifier for the system call

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19-22). The ID is stored in 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 in 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 of 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 upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using **semctl()** with the GETALL control command.

Example 2-9 semop() System Call Example

```
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("ensemid = %d", semid);
25    /*Enter the number of operations.*/
26    printf("\nEnter the number of semaphore\n");
```

```
   printf("operations for this set = ");
   scanf("%d", &nsops);
   printf("ennosops = %d", nsops);
   /*Initialize the array for the
      number of operations to be performed.*/
   for(i = 0; i < nsops; i++, sops++)
   {
     /*This determines the semaphore in
     the semaphore set.*/
     printf("\nEnter the semaphore\n");
     printf("number (sem_num) = ");
     scanf("%d", &sem_num);
     sops->sem_num = sem_num;
     printf("\nThe sem_num = %d", sops->sem_num);
    /*Enter a (-)number to decrement,
      an unsigned number (no +) to increment,
      or zero to test for zero. These values
      are entered into a string and converted
      to integer values.*/
    printf("\nEnter the operation for\n");
    printf("the semaphore (sem_op) = ");
    scanf("%s", string);
    sops->sem_op = atoi(string);
    printf("ensem_op = %d\n", sops->sem_op);
    /*Specify the desired flags.*/
    printf("\nEnter the corresponding\n");
    printf("number for the desired\n");
    printf("flags:\n");
    printf("No flags = 0\n");
    printf("IPC_NOWAIT = 1\n");
    printf("SEM_UNDO = 2\n");
    printf("IPC_NOWAIT and SEM_UNDO = 3\n");
    printf(" Flags = ");
    scanf("%d", &flags);
    switch(flags)
    {
    case 0:
      sops->sem_flg = 0;
      break;
    case 1:
      sops->sem_flg = IPC_NOWAIT;
      break;
    case 2:
      sops->sem_flg = SEM_UNDO;
      break;
```

```
    case 3:
      sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
      break;
    }
    printf("\nFlags = 0%o\n", sops->sem_flg);
  }
  /*Print out each structure in the array.*/
  for(i = 0; i < nsops; i++)
  {
      printf("\nsem_num = %d\n", sembuf[i].sem_num);
      printf("sem_op = %d\n", sembuf[i].sem_op);
      printf("sem_flg = %o\n", sembuf[i].sem_flg);
      printf("%c", ' ');
    }
    sops = sembuf; /*Reset the pointer to
        sembuf[0].*/
    /*Do the semop system call.*/
    retrn = semop(semid, sops, nsops);
    if(retrn == -1) {
      printf("\nSemop failed. ");
      printf("Error = %d\n", errno);
    }
    else {
      printf ("\nSemop was successful\n");
      printf("for semid = %d\n", semid);
      printf("Value returned = %d\n", retrn);
  }
}
```

System V Shared Memory

Shared memory IPC allows two or more executing processes to share memory and consequently the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which depends on memory management hardware.

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. Arguments to this function set 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) upon 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 perform other functions on the shared memory segment using the **shmctl()** system call.

System calls, documented in the IRIX reference pages, make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call attempts to perform its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (-1) is returned to the process, and the external variable *errno* is set accordingly.

Using Shared Memory

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

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

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

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

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

Note that the *shm_perm* member of this structure uses *ipc_perm* as a template. The *ipc_perm* data structure is the same for all IPC facilities, and is located in the */usr/include/sys/ipc.h* header file.

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

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

The task performed is determined by the value of the *key* argument passed to the **shmget()** 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 tuning parameters would be exceeded.

A provision exists for specifying a *key* of value zero, 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 tuning 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 you do not want to have an existing *shmid* returned, a control command (IPC_EXCL) can be set in the *shmflg* argument passed to the system call. The details of using this system call are discussed in "Getting Shared Memory Segments With shmget()."

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 "Controlling Shared Memory: shmctl()." The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely identified shared memory segment data structure is created, shared memory segment operations and control can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are **shmat()** and **shmdt()**. Refer to “Operations for Shared Memory: shmat() and shmdt()” 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 operating system along with its associated shared memory segment data structure
- to lock a shared memory segment in memory
- to unlock a shared memory segment

Refer to “Controlling Shared Memory: shmctl()” for details of the **shmctl()** system call.

Getting Shared Memory Segments With shmget()

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

The prototype found in the shmget(2) reference page is as follows:

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

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

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

- *key* is equal to IPC_PRIVATE.
- *key* is passed a unique hexadecimal integer, and *shmflg* contains IPC_CREAT.

The value passed to the *shmflg* argument specifies access permissions, execution modes, and command options.

Access permissions determine the read/write attributes, and execution modes determine the user/group/other attributes of the *shmflg* argument. They are collectively referred to as "operation permissions." Table 2-6 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 2-6 Operation Permissions Codes

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

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others are desired, the code value would be 00406 (00400 plus 00004 plus 00002).

Constants located in the *shm.h* header file can be used for the owner. They include:

```
SHM_R 0400
SHM_W 0200
```

Control command constants are predefined in the *ipc.h* header file. Table 2-7 contains the names of the constants that apply to the **shmget()** system call along with their values.

Table 2-7 Control Commands (Flags)

Control	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for *shmflg* is a combination of operation permissions and control commands, usually an expression using bitwise OR (|) with the operation permissions.

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

As specified by the shmget(2) reference page, success or failure of this system call depends upon the argument values for *key, size*, and *shmflg*, and on system tuning parameters. The system call attempts to return a new *shmid* if one of the following conditions is true:

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

The 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 not zero. Exceeding the SHMMNI system tuning parameter in either case causes a failure. The SHMMNI system tuning parameter determines the maximum number of unique shared memory segments (*shmids*) in the UNIX operating system.

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

The system call fails 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(2) reference 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 (Example 2-10) is a menu-driven program that exercises all possible combinations of the **shmget()** system call. The program begins (lines 4-7) by including the required header files as specified by the shmget(2) reference page. Note that the *errno.h* header file is included as opposed to declaring *errno* as an external variable; either method works.

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	The desired semaphore key
opperm	The desired operation permissions
flags	The desired control commands (flags)
opperm_flags	The combination from the logical ORing of the *opperm* and *flags* variables, used in the system call to pass the *shmflg* argument
shmid	The message queue identification number from a successful system call or the error code (-1) for an unsuccessful one
size	The shared memory segment size

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and finally for the control command combinations (flags), which are selected from a menu (lines 14-31). All possible combinations are allowed even though they might not be viable. Thus you can observe the errors for illegal combinations.

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

A display then prompts for the size of the shared memory segment, which is stored in the *size* variable (lines 51-54).

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

Since the *shmid* variable now contains a valid message queue identifier or the error code (-1), it is tested to see if an error occurred (line 58). If *shmid* equals -1, a message indicates that an error resulted and the external *errno* variable is displayed (lines 60, 61). If no error occurred, the returned shared memory segment identifier is displayed (line 65).

Example 2-10 shmget() System Call Example

```
/*This is a program to illustrate
 *the shared memory get, shmget(),
 *system call capabilities.*/
 #include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/shm.h>
 #include <errno.h>
 /*Start of main C language program*/
 main()
 {
    key_t key; /*declare as long integer*/
    int opperm, flags;
    int shmid, size, opperm_flags;
    /*Enter the desired key*/
    printf("Enter the desired key in hex = ");
    scanf("%x", &key);
    /*Enter the desired octal operation
      permissions.*/
    printf("\nEnter the operation\n");
    printf("permissions in octal = ");
    scanf("%o", &opperm);
    /*Set the desired flags.*/
    printf("\nEnter corresponding number to\n");
    printf("set the desired flags:\n");
    printf("No flags = 0\n");
    printf("IPC_CREAT = 1\n");
    printf("IPC_EXCL = 2\n");
    printf("IPC_CREAT and IPC_EXCL = 3\n");
    printf(" Flags = ");
    /*Get the flag(s) to be set.*/
    scanf("%d", &flags);
    /*Check the values.*/
    printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
      key, opperm, flags);
    /*Incorporate the control fields (flags) with
      the operation permissions*/
    switch (flags)
    {
    case 0: /*No flags are to be set.*/
      opperm_flags = (opperm | 0);
      break;
    case 1: /*Set the IPC_CREAT flag.*/
      opperm_flags = (opperm | IPC_CREAT);
```

```
      break;
    case 2: /*Set the IPC_EXCL flag.*/
      opperm_flags = (opperm | IPC_EXCL);
      break;
    case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
      opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
    }
    /*Get the size of the segment in bytes.*/
    printf ("\nEnter the segment");
    printf ("\nsize in bytes = ");
    scanf ("%d", &size);
    /*Call the shmget system call.*/
    shmid = shmget (key, size, opperm_flags);
    /*Perform the following if the call is unsuccessful.*/
    if(shmid == -1)
    {
      printf ("\nThe shmget system call failed!\n");
      printf ("The error number = %d\n", errno);
    }
    /*Return the shmid upon successful completion.*/
    else
      printf ("\nThe shmid = %d\n", shmid);
    exit(0);
  }
```

Controlling Shared Memory: shmctl()

This section details using the **shmctl()** system call and provides an example program that exercises all of its capabilities.

The prototype found in the shmctl(2) reference page is as follows:

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

The **shmctl()** system call requires two or three arguments to be passed to it, and it returns an integer value. Upon successful completion, a zero value is returned; when unsuccessful, **shmctl()** returns a -1.

The *shmid* variable must be a nonnegative integer returned by **shmget()**. 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 the effective user identification of the owner/creator of the segment, or of superuser, to perform an IPC_SET or IPC_RMID control command. Only the superuser can perform a SHM_LOCK or SHM_UNLOCK control command. A process must have read permission to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read “Getting Shared Memory Segments With shmget()”; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Example 2-11) is a menu-driven program that exercises all possible combinations of the **shmctl()** system call. From studying this program, you can observe the method of passing arguments and receiving return values. The user-written program requirements are pointed out.

This program begins (lines 5-9) by including the required header files as specified by the shmctl(2) reference page. 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 The IPC_SET value for the effective user identification.

gid The IPC_SET value for the effective group identification.

mode The IPC_SET value for the operation permissions.

rtrn The return integer value from the system call.

shmid The shared memory segment identifier to the system call.

command The code for the desired control command so that subsequent processing can be performed on it.

choice Which member for the IPC_SET control command is to be changed.

shmid_ds The data structure returned from an IPC_STAT control command.

buf Address of the data structure in the user memory area where IPC_STAT 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).

First, the program prompts for a valid shared memory segment identifier that is stored in 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 in the *command* variable. The code is tested to determine the control command for subsequent processing.

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

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 90-92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93-98). This code is stored in the *choice* variable (line 99). Now, depending upon the member picked, the program prompts for the new value (lines 105-127). The value is placed in the appropriate member in the user memory area data structure, and the system call is made (lines 128-130). Depending upon 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 operating system. Note that the **buf* pointer is not required as an argument to perform this control command and its value can be zero or NULL. Depending upon 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 upon 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 upon the success or failure, the program returns the same messages as for the other control commands.

Example 2-11 shmctl() System Call Example

```
/*This is a program to illustrate
*the shared memory control, shmctl(),
*system call capabilities.
*/
/*Include necessary header files.*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
 /*Start of main C language program*/
 main()
 {
    extern int errno;
    int uid, gid, mode;
    int rtrn, shmid, command, choice;
    struct shmid_ds shmid_ds, *buf;
    buf = &shmid_ds;
    /*Get the shmid, and command.*/
    printf("Enter the shmid = ");
    scanf("%d", &shmid);
    printf("\nEnter the number for\n");
    printf("the desired command:\n");
    printf("IPC_STAT = 1\n");
    printf("IPC_SET = 2\n");
    printf("IPC_RMID = 3\n");
    printf("SHM_LOCK = 4\n");
    printf("SHM_UNLOCK = 5\n");
    printf("Entry = ");
    scanf("%d", &command);
    /*Check the values.*/
    printf ("\nshmid =%d, command = %d\n",
    shmid, command);
    switch (command)
    {
    case 1: /*Use shmctl() to duplicate
      the data structure for
        shmid in the shmid_ds area pointed
        to by buf and then print it out.*/
      rtrn = shmctl(shmid, IPC_STAT,
        buf);
      printf ("\nThe USER ID = %d\n",
        buf->shm_perm.uid);
      printf ("The GROUP ID = %d\n",
```

```
        buf->shm_perm.gid);
      printf ("The creator's ID = %d\n",
        buf->shm_perm.cuid);
      printf ("The creator's group ID = %d\n",
        buf->shm_perm.cgid);
      printf ("The operation permissions = 0%o\n",
        buf->shm_perm.mode);
      printf ("The slot usage sequenceen");
      printf ("number = 0%x\n",
        buf->shm_perm.seq);
      printf ("The key= 0%x\n",
        buf->shm_perm.key);
      printf ("The segment size = %d\n",
        buf->shm_segsz);
      printf ("The pid of last shmop = %d\n",
        buf->shm_lpid);
      printf ("The pid of creator = %d\n",
        buf->shm_cpid);
      printf ("The current # attached = %d\n",
        buf->shm_nattch);
      printf("The in memory # attached = %d\n",
        buf->shm_cnattach);
      printf("The last shmat time = %d\n",
        buf->shm_atime);
      printf("The last shmdt time = %d\n",
        buf->shm_dtime);
      printf("The last change time = %d\n",
        buf->shm_ctime);
      break;
      /* Lines 73 - 87 deleted */
    case 2: /*Select and change the desired
      member(s) of the data structure.*/
      /*Get the original data for this shmid
         data structure first.*/
      rtrn = shmctl(shmid, IPC_STAT, buf);
      printf("\nEnter the number for the\n");
      printf("member to be changed:\n");
      printf("shm_perm.uid = 1\n");
      printf("shm_perm.gid = 2\n");
      printf("shm_perm.mode = 3\n");
      printf("Entry = ");
      scanf("%d", &choice);
      /*Only one choice is allowed per
        pass as an illegal entry will
        cause repetitive failures until
```

```
     shmid_ds is updated with
     IPC_STAT.*/
  switch(choice){
  case 1:
    printf("\nEnter USER ID = ");
    scanf ("%d", &uid);
    buf->shm_perm.uid = uid;
    printf("\nUSER ID = %d\n",
    buf->shm_perm.uid);
    break;
  case 2:
    printf("\nEnter GROUP ID = ");
    scanf("%d", &gid);
    buf->shm_perm.gid = gid;
    printf("\nGROUP ID = %d\n",
    buf->shm_perm.gid);
    break;
  case 3:
    printf("\nEnter MODE = ");
    scanf("%o", &mode);
    buf->shm_perm.mode = mode;
    printf("\nMODE = 0%o\n",
    buf->shm_perm.mode);
    break;
}
/*Do the change.*/
rtrn = shmctl(shmid, IPC_SET,
buf);
break;
case 3: /*Remove the shmid along with its
  associated
  data structure.*/
  rtrn = shmctl(shmid, IPC_RMID, NULL);
  break;
case 4: /*Lock the shared memory segment*/
  rtrn = shmctl(shmid, SHM_LOCK, NULL);
  break;
case 5: /*Unlock the shared memory
  segment.*/
  rtrn = shmctl(shmid, SHM_UNLOCK, NULL);
  break;
}
/*Perform the following if the call fails.*/
if(rtrn == -1)
{
```

```
      printf ("\nThe shmctl system call failed!\n");
      printf ("The error number = %d\n", errno);
    }
    /*Return the shmid upon successful completion.*/
    else
      printf ("\nShmctl was successful for shmid = %d\n",
      shmid);
    exit (0);
}
```

Operations for Shared Memory: shmat() and shmdt()

This section details the **shmat()** and **shmdt()** system calls, and presents an example program that exercises all of their capabilities.

The prototypes found in the shmop(2) reference page, which includes both **shmat()** and **shmaddr()**, are as follows:

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

Attaching to a Shared Memory Segment

The **shmat()** system call takes three arguments and it returns a character pointer value. The returned value can be cast to an integer value. Upon successful completion, this value is the address in memory where the process is attached to the shared memory segment, and when unsuccessful, it is -1.

The *shmid* argument must be a nonnegative integer returned from **shmget()**. The *shmaddr* argument can be 0 or nonzero. When it is 0 , IRIX picks the address of the shared segment. When *shmaddr* is nonzero it must be a valid address that IRIX would pick. It is wise to let the operating system pick addresses so as to improve portability.

The *shmflg* argument passes the SHM_RND and SHM_RDONLY flags to **shmat()**.

Detaching Shared Memory Segments

The **shmdt()** system call takes one argument and returns an integer value, 0 if it completes successfully, otherwise -1. Further details of this system call are discussed in the example program.

Example Program

The example program in Example 2-12 is a menu-driven program that exercises all possible combinations of the **shmat()** and **shmdt()** system calls.

This program begins (lines 5-9) by including the required header files as specified by the shmop(2) reference page. 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 were chosen to be as close as possible to those in the synopsis. Their declarations are self-explanatory. The names make the program more readable; this is legal since they are local to the program. The variables declared for this program and their purposes include:

flags	The codes of SHM_RND or SHM_RDONLY for the **shmat()** system call
addr	The address of the shared memory segment for the **shmat()** and **shmdt()** system calls
i	A loop counter for attaching and detaching
attach	The desired number of attach operations
shmid	The desired shared memory segment identifier
shmflg	The value of flags to the **shmat()** system call
rtrn	The return values from both system calls
detach	The desired number 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.

Example Use of shmat()

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

A loop is entered using the attach variable and the *i* counter (lines 23-70) to perform the specified number of attachments. In this loop, the program prompts for a shared memory segment identifier (lines 24-27) and it is stored in the *shmid* variable (line 28). Next, the program prompts for the address where the segment is to be attached (lines 30-34), and it is stored in 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 in 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 the call succeeds, a message stating so is displayed along with the attach address (lines 66-68). If the call is unsuccessful, a message stating so is displayed and the error code is displayed (lines 62, 63). The loop then continues.

Example Use of 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 in the *detach* variable (line 76).

A loop is entered using the detach variable and the *i* counter (lines 78-95) to perform the specified number of detachments. In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79-83), and it is stored in the *addr* variable (line 84). Then, the **shmdt()** system call is performed (line 87). If the call succeeds, a message stating so is displayed along with the address that the segment was detached from (lines 92,93). When the call is unsuccessful, the error number is displayed (line 89). The loop continues.

Example 2-12 shmat() and shmdt() System Call Example

```
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>
```

```
/*Start of main C language program*/
main()
{
   extern int errno;
   int flags, addr, i, attach;
   int shmid, shmflg, retrn, detach;
   /*Loop for attachments by this process.*/
   printf("Enter the number ofen");
   printf("attachments for this\n");
   printf("process (1-4).\n");
   printf(" Attachments = ");
   scanf("%d", &attach);
   printf("Number of attaches = %d\n", attach);
   for(i = 1; i <= attach; i++) {
     /*Enter the shared memory ID.*/
     printf("\nEnter the shmid ofen");
     printf("the shared memory segment to\n");
     printf("be operated on = ");
     scanf("%d", &shmid);
     printf("\nshmid = %d\n", shmid);
     /*Enter the value for shmaddr.*/
     printf("\nEnter the value for\n");
     printf("the shared memory address\n");
     printf("in hexadecimal:\n");
     printf(" Shmaddr = ");
     scanf("%x", &addr);
     printf("The desired address = 0x%x\n", addr);
     /*Specify the desired flags.*/
     printf("\nEnter the corresponding\n");
     printf("number for the desireden");
     printf("flags:\n");
     printf("SHM_RND = 1\n");
     printf("SHM_RDONLY = 2\n");
     printf("SHM_RND and SHM_RDONLY = 3\n");
     printf(" Flags = ");
     scanf("%d", &flags);
     switch(flags)
     {
     case 1:
       shmflg = SHM_RND;
       break;
     case 2:
       shmflg = SHM_RDONLY;
       break;
     case 3:
```

```
        shmflg = SHM_RND | SHM_RDONLY;
        break;
      }
      printf("\nFlags = 0%o\n", shmflg);
      /*Do the shmat system call.*/
      retrn = (int)shmat(shmid, addr, shmflg);
      if(retrn == -1) {
        printf("\nShmat failed. ");
        printf("Error = %d\n", errno);
      }
      else {
        printf ("\nShmat was successful\n");
        printf("for shmid = %d\n", shmid);
        printf("The address = 0x%x\n", retrn);
      }
  }
  /*Loop for detachments by this process.*/
  printf("Enter the number ofen");
  printf("detachments for this\n");
  printf("process (1-4).\n");
  printf(" Detachments = ");
  scanf("%d", &detach);
  printf("Number of attaches = %d\n", detach);
  for(i = 1; i <= detach; i++) {
    /*Enter the value for shmaddr.*/
    printf("\nEnter the value for\n");
    printf("the shared memory address\n");
    printf("in hexadecimal:\n");
    printf(" Shmaddr = ");
    scanf("%x", &addr);
    printf("The desired address = 0x%x\n", addr);
    /*Do the shmdt system call.*/
    retrn = (int)shmdt(addr);
    if(retrn == -1) {
      printf("Error = %d\n", errno);
    }
    else {
      printf ("\nShmdt was successful\n");
      printf("for address = 0%x\n", addr);
    }
  }
}
```