14

Semaphores, Shared Memory, and Message Queues

In this chapter, we discuss a set of inter-process communication facilities that were originally introduced in the AT&T System V.2 release of UNIX. Because all these facilities appeared in the same release and have a similar programmatic interface, they are often referred to as the IPC (Inter-Process Communication) facilities, or more commonly System V IPC. As you've already seen, they are by no means the only way of communicating between processes, but the expression System V IPC is usually used to refer to these specific facilities.

We cover the following topics in this chapter:

- Semaphores, for managing access to resources
- Shared memory, for highly efficient data sharing between programs
- Messaging, for an easy way of passing data between programs

Semaphores

When you write programs that use threads operating in multiuser systems, multiprocessing systems, or a combination of the two, you may often discover that you have *critical sections* of code, where you need to ensure that a single process (or a single thread of execution) has exclusive access to a resource.

Semaphores have a complex programming interface. Fortunately, you can easily provide a much-simplified interface that is sufficient for most semaphore-programming problems.

In the first example application in Chapter 7 — using `dbm` to access a database — the data could be corrupted if multiple programs tried to update the database at exactly the same time. There's

no trouble with two different programs asking different users to enter data for the database; the only potential problem is in the parts of the code that update the database. These sections of code, which actually perform data updates and need to execute exclusively, are called *critical sections*. Frequently they are just a few lines of code from much larger programs.

To prevent problems caused by more than one program simultaneously accessing a shared resource, you need a way of generating and using a token that grants access to only one thread of execution in a critical section at a time. You saw briefly in Chapter 12 some thread-specific ways you could use a mutex or semaphores to control access to critical sections in a threaded program. In this chapter, we return to the topic of semaphores, but look more generally at how they are used between different processes.

> *The semaphore functions used with threads that you saw in Chapter 12 are not the more general ones we discuss in this chapter, so be careful not to confuse the two types.*

It's surprisingly difficult to write general-purpose code that ensures that one program has exclusive access to a particular resource, although there's a solution known as Dekker's Algorithm. Unfortunately, this algorithm relies on a "busy wait," or "spin lock," where a process runs continuously, waiting for a memory location to be changed. In a multitasking environment such as Linux, this is an undesirable waste of CPU resources. The situation is much easier if hardware support, generally in the form of specific CPU instructions, is available to support exclusive access. An example of hardware support would be an instruction to access and increment a register in an atomic way, such that no other instruction (not even an interrupt) could occur between the read/increment/write operations.

One possible solution that you've already seen is to create files using the `O_EXCL` flag with the `open` function, which provides atomic file creation. This allows a single process to succeed in obtaining a token: the newly created file. This method is fine for simple problems, but rather messy and very inefficient for more complex examples.

An important step forward in this area of concurrent programming occurred when Edsger Dijkstra, a Dutch computer scientist, introduced the concept of the semaphore. As briefly mentioned in Chapter 12, a semaphore is a special variable that takes only whole positive numbers and upon which programs can only act atomically. In this chapter we expand on that earlier simplified definition. We show in more detail how semaphores function, and how the more general-purpose functions can be used between separate processes, rather than the special case of multi-threaded programs you saw in Chapter 12.

A more formal definition of a semaphore is a special variable on which only two operations are allowed; these operations are officially termed *wait* and *signal*. Because "wait" and "signal" already have special meanings in Linux programming, we'll use the original notation:

- `P(semaphore variable)` for wait
- `V(semaphore variable)` for signal

These letters come from the Dutch words for wait (*passeren*: to pass, as in a checkpoint before the critical section) and signal (*vrijgeven*: to give or release, as in giving up control of the critical section). You may also come across the terms "up" and "down" used in relation to semaphores, taken from the use of signaling flags.

Semaphore Definition

The simplest semaphore is a variable that can take only the values 0 and 1, a *binary semaphore.* This is the most common form. Semaphores that can take many positive values are called *general semaphores*. For the remainder of this chapter, we concentrate on binary semaphores.

The definitions of `P` and `V` are surprisingly simple. Suppose you have a semaphore variable `sv`. The two operations are then defined as follows:

`P(sv)`	If `sv` is greater than zero, decrement `sv`. If `sv` is zero, suspend execution of this process.
`V(sv)`	If some other process has been suspended waiting for `sv`, make it resume execution. If no process is suspended waiting for `sv`, increment `sv`.

Another way of thinking about semaphores is that the semaphore variable, `sv`, is `true` when the critical section is available, is decremented by `P(sv)` so it's `false` when the critical section is busy, and is incremented by `V(sv)` when the critical section is again available. Be aware that simply having a normal variable that you decrement and increment is not good enough, because you can't express in C, C++, C#, or almost any conventional programming language the need to make a single, atomic operation of the test to see whether the variable is `true`, and if so change the variable to make it `false`. This is what makes the semaphore operations special.

A Theoretical Example

You can see how this works with a simple theoretical example. Suppose you have two processes `proc1` and `proc2`, both of which need exclusive access to a database at some point in their execution. You define a single binary semaphore, `sv`, which starts with the value 1 and can be accessed by both processes. Both processes then need to perform the same processing to access the critical section of code; indeed, the two processes could simply be different invocations of the same program.

The two processes share the `sv` semaphore variable. Once one process has executed `P(sv)`, it has obtained the semaphore and can enter the critical section. The second process is prevented from entering the critical section because when it attempts to execute `P(sv)`, it's made to wait until the first process has left the critical section and executed `V(sv)` to release the semaphore.

The required pseudocode is identical for both processes:

```
semaphore sv = 1;

loop forever {
    P(sv);
    critical code section;
    V(sv);
    noncritical code section;
}
```

The code is surprisingly simple because the definition of the `P` and `V` operations is very powerful. Figure 14-1 shows a diagram showing how the `P` and `V` operations act as a gate into critical sections of code.

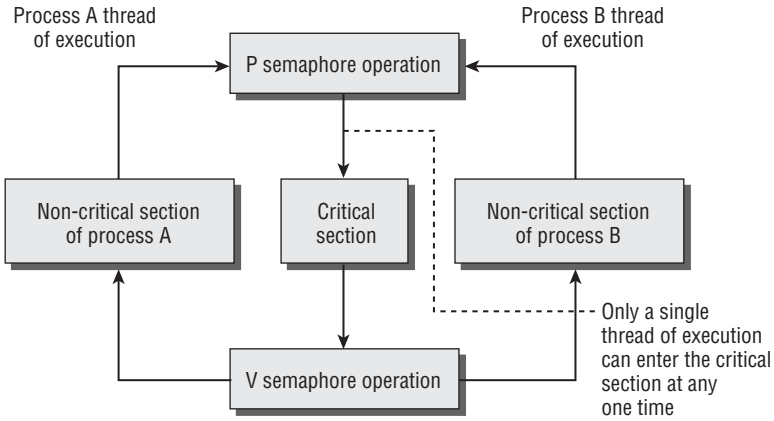

Figure 14-1

Linux Semaphore Facilities

Now that you've seen what semaphores are and how they work in theory, you can look at how the features are implemented in Linux. The interface is rather elaborate and offers far more facilities than are generally required. All the Linux semaphore functions operate on arrays of general semaphores rather than a single binary semaphore. At first sight, this just seems to make things more complicated, but in complex cases where a process needs to lock multiple resources, the ability to operate on an array of semaphores is a big advantage. In this chapter, we concentrate on using single semaphores, because in most cases that's all you will need to use.

The semaphore function definitions are

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

int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

The header file `sys/sem.h` *usually relies on two other header files,* `sys/types.h` *and* `sys/ipc.h`*. Normally they are automatically included by* `sys/sem.h` *and you do not need to explicitly add a* `#include` *for them.*

As you work through each function in turn, remember that these functions were designed to work for arrays of semaphore values, which makes their operation significantly more complex than would have been required for a single semaphore.

Notice that `key` acts very much like a filename in that it represents a resource that programs may use and cooperate in using if they agree on a common name. Similarly, the identifier returned by `semget` and used by the other shared memory functions is very much like the `FILE *` file stream returned by `fopen` in that it's a value used by the process to access the shared file. Just as with files different processes will have different semaphore identifiers, though they refer to the same semaphore. This use of a key and identifiers is common to all of the IPC facilities discussed here, although each facility uses independent keys and identifiers.

semget

The `semget` function creates a new semaphore or obtains the semaphore key of an existing semaphore:

```
int semget(key_t key, int num_sems, int sem_flags);
```

The first parameter, `key`, is an integral value used to allow unrelated processes to access the same semaphore. All semaphores are accessed indirectly by the program supplying a key, for which the system then generates a semaphore identifier. The semaphore key is used only with `semget`. All other semaphore functions use the semaphore identifier returned from `semget`.

There is a special semaphore `key` value, `IPC_PRIVATE`, that is intended to create a semaphore that only the creating process could access, but this rarely has any useful purpose. You should provide a unique, non-zero integer value for `key` when you want to create a new semaphore.

The `num_sems` parameter is the number of semaphores required. This is almost always 1.

The `sem_flags` parameter is a set of flags, very much like the flags to the `open` function. The lower nine bits are the permissions for the semaphore, which behave like file permissions. In addition, these can be bitwise ORed with the value `IPC_CREAT` to create a new semaphore. It's not an error to have the `IPC_CREAT` flag set and give the key of an existing semaphore. The `IPC_CREAT` flag is silently ignored if it is not required. You can use `IPC_CREAT` and `IPC_EXCL` together to ensure that you obtain a new, unique semaphore. It will return an error if the semaphore already exists.

The `semget` function returns a positive (nonzero) value on success; this is the semaphore identifier used in the other semaphore functions. On error, it returns –1.

semop

The function `semop` is used for changing the value of the semaphore:

```
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

The first parameter, `sem_id`, is the semaphore identifier, as returned from `semget`. The second parameter, `sem_ops`, is a pointer to an array of structures, each of which will have at least the following members:

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

The first member, `sem_num`, is the semaphore number, usually 0 unless you're working with an array of semaphores. The `sem_op` member is the value by which the semaphore should be changed. (You can change a semaphore by amounts other than 1.) In general, only two values are used, –1, which is your `P` operation to wait for a semaphore to become available, and +1, which is your `V` operation to signal that a semaphore is now available.

The final member, `sem_flg`, is usually set to `SEM_UNDO`. This causes the operating system to track the changes made to the semaphore by the current process and, if the process terminates without releasing the semaphore, allows the operating system to automatically release the semaphore if it was held by this

process. It's good practice to set `sem_flg` to `SEM_UNDO`, unless you specifically require different behavior. If you do decide you need a value other than `SEM_UNDO`, it's important to be consistent, or you can get very confused as to whether the kernel is attempting to "tidy up" your semaphores when your process exits.

All actions called for by `semop` are taken together to avoid a race condition implied by the use of multiple semaphores. You can find full details of the processing of `semop` in the manual pages.

semctl

The `semctl` function allows direct control of semaphore information:

```
int semctl(int sem_id, int sem_num, int command, ...);
```

The first parameter, `sem_id`, is a semaphore identifier, obtained from `semget`. The `sem_num` parameter is the semaphore number. You use this when you're working with arrays of semaphores. Usually, this is 0, the first and only semaphore. The `command` parameter is the action to take, and a fourth parameter, if present, is a `union semun`, which according to the X/OPEN specification must have at least the following members:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
}
```

Most versions of Linux have a definition of the `semun` union in a header file (usually `sem.h`), though X/Open does say that you have to declare your own. If you do find that you need to declare your own, check the manual pages for `semctl` to see if there is a definition given. If there is, we suggest you use exactly the definition given in your manual, even if it differs from that given here.

There are many different possible values of `command` allowed for `semctl`. Only the two that we describe here are commonly used. For full details of the `semctl` function, you should consult the manual page.

The two common values of `command` are:

- `SETVAL`: Used for initializing a semaphore to a known value. The value required is passed as the `val` member of the union `semun`. This is required to set the semaphore up before it's used for the first time.
- `IPC_RMID`: Used for deleting a semaphore identifier when it's no longer required.

The `semctl` function returns different values depending on the `command` parameter. For `SETVAL` and `IPC_RMID` it returns 0 for success and –1 on error.

Using Semaphores

As you can see from the previous section's descriptions, semaphore operations can be rather complex. This is most unfortunate, because programming multiple processes or threads with critical sections is quite a difficult problem on its own and having a complex programming interface simply adds to the intellectual burden.

Fortunately you can solve most problems that require semaphores using only a single binary semaphore, the simplest type. In the following example, you use the full programming interface to create a much simpler `P` and `V` type interface for a binary semaphore. You then use this much simpler interface to demonstrate how semaphores function.

To experiment with semaphores, you use a single program, `sem1.c`, that you can invoke several times. You use an optional parameter to specify whether the program is responsible for creating and destroying the semaphore.

You use the output of two different characters to indicate entering and leaving the critical section. The program invoked with a parameter prints an `X` on entering and exiting its critical section. Other invocations of the program print an `O` on entering and exiting their critical sections. Because only one process should be able to enter its critical section at any given time, all `X` and `O` characters should appear in pairs.

Try It Out Semaphores

1. After the system `#includes`, you include a file `semun.h`. This defines the union `semun`, as required by X/OPEN, if the system include `sys/sem.h` doesn't already define it. Then come the function prototypes, and the global variable, before you come to the `main` function. There the semaphore is created with a call to `semget`, which returns the semaphore ID. If the program is the first to be called (that is, it's called with a parameter and `argc > 1`), a call is made to `set_semvalue` to initialize the semaphore and `op_char` is set to `X`:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>


#include <sys/sem.h>

#include "semun.h"


static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);

static int sem_id;


int main(int argc, char *argv[])
{
    int i;
    int pause_time;
    char op_char = 'O';

    srand((unsigned int)getpid());

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);

    if (argc > 1) {
```

```
            if (!set_semvalue()) {
                fprintf(stderr, "Failed to initialize semaphore\n");
                exit(EXIT_FAILURE);
            }
            op_char = 'X';
            sleep(2);
        }
```

2. Then you have a loop that enters and leaves the critical section 10 times. There you first make a call to `semaphore_p`, which sets the semaphore to wait as this program is about to enter the critical section:

```
        for(i = 0; i < 10; i++) {

            if (!semaphore_p()) exit(EXIT_FAILURE);
            printf("%c", op_char);fflush(stdout);
            pause_time = rand() % 3;
            sleep(pause_time);
            printf("%c", op_char);fflush(stdout);
```

3. After the critical section, you call `semaphore_v`, setting the semaphore as available, before going through the `for` loop again after a random wait. After the loop, the call to `del_semvalue` is made to clean up the code:

```
            if (!semaphore_v()) exit(EXIT_FAILURE);

            pause_time = rand() % 2;
            sleep(pause_time);
        }

        printf("\n%d - finished\n", getpid());

        if (argc > 1) {
            sleep(10);
            del_semvalue();
        }

        exit(EXIT_SUCCESS);
    }
```

4. The function `set_semvalue` initializes the semaphore using the `SETVAL` command in a `semctl` call. You need to do this before you can use the semaphore:

```
    static int set_semvalue(void)
    {
        union semun sem_union;

        sem_union.val = 1;
        if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
        return(1);
    }
```

5. The `del_semvalue` function has almost the same form, except that the call to `semctl` uses the command `IPC_RMID` to remove the semaphore's ID:

```
static void del_semvalue(void)
{
    union semun sem_union;

    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}
```

6. `semaphore_p` changes the semaphore by –1. This is the "wait" operation:

```
static int semaphore_p(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = -1; /* P() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}
```

7. `semaphore_v` is similar except for setting the `sem_op` part of the `sembuf` structure to 1. This is the "release" operation, so that the semaphore becomes available:

```
static int semaphore_v(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = 1; /* V() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
    }
    return(1);
}
```

Notice that this simple program allows only a single binary semaphore per program, although you could extend it to pass the semaphore variable if you need more semaphores. Normally, a single binary semaphore is sufficient.

You can test your program by invoking it several times. The first time, you pass a parameter to tell the program that it's responsible for creating and deleting the semaphore. The other invocations have no parameter.

Here's some sample output, with two invocations of the program.

```
$ cc sem1.c -o sem1
$ ./sem1 1 &
[1] 1082
$ ./sem1
OOXXOOXXOOXXOOXXOOXXOOOOXXOOXXOOXXOOXXXX
1083 - finished
1082 - finished
$
```

Remember that "O" represents the first invocation of the program, and "X" the second invocation of the program. Because each program prints a character as it enters and again as it leaves the critical section, each character should only appear as part of a pair. As you can see, the `O`s and `X`s are indeed properly paired, indicating that the critical section is being correctly processed. If this doesn't work on your particular system, you may have to use the command `stty -tostop` before invoking the program to ensure that the background program generating `tty` output does not cause a signal to be generated.

How It Works

The program starts by obtaining a semaphore identity from the (arbitrary) key that you've chosen using the `semget` function. The `IPC_CREAT` flag causes the semaphore to be created if one is required.

If the program has a parameter, it's responsible for initializing the semaphore, which it does with the function `set_semvalue`, a simplified interface to the more general `semctl` function. It also uses the presence of the parameter to determine which character it should print out. The `sleep` simply allows you some time to invoke other copies of the program before this copy gets to execute too many times around its loop. You use `srand` and `rand` to introduce some pseudo-random timing into the program.

The program then loops 10 times, with pseudo-random waits in its critical and noncritical sections. The critical section is guarded by calls to your `semaphore_p` and `semaphore_v` functions, which are simplified interfaces to the more general `semop` function.

Before it deletes the semaphore, the program that was invoked with a parameter then waits to allow other invocations to complete. If the semaphore isn't deleted, it will continue to exist in the system even though no programs are using it. In real programs, it's very important to ensure you don't unintentionally leave semaphores around after execution. It may cause problems next time you run the program, and semaphores are a limited resource that you must conserve.

Shared Memory

Shared memory is the second of the three IPC facilities. It allows two unrelated processes to access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes. Although the X/Open standard doesn't require it, it's probable that most implementations of shared memory arrange for the memory being shared between different processes to be the same physical memory.

Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process. Other processes can then "attach" the same shared memory segment into their own address space. All processes can access the memory locations just as if the memory had been allocated by `malloc`. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

Shared memory provides an efficient way of sharing and passing data between multiple processes. By itself, shared memory doesn't provide any synchronization facilities. Because it provides no synchronization facilities, you usually need to use some other mechanism to synchronize access to the shared memory. Typically, you might use shared memory to provide efficient access to large areas of memory and pass small messages to synchronize access to that memory.

There are no automatic facilities to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It's the responsibility of the programmer to synchronize access. Figure 14-2 shows an illustration of how shared memory works.

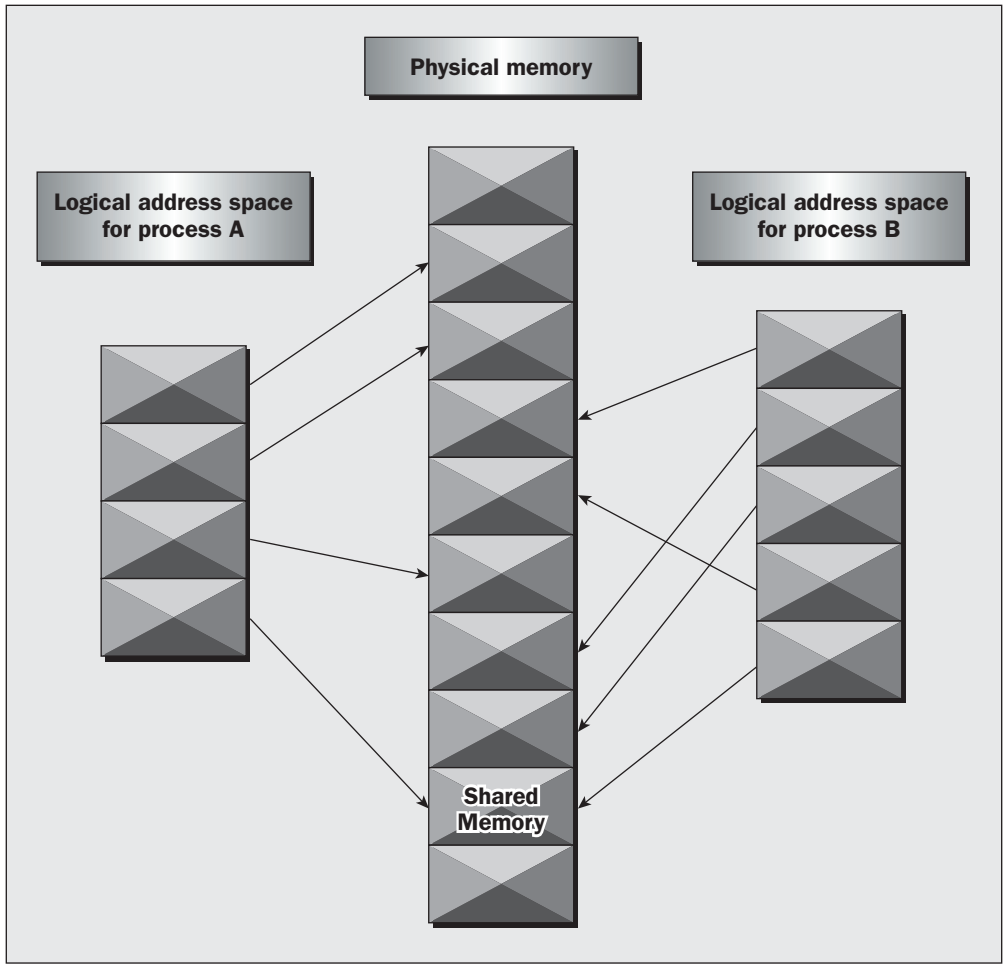

Figure 14-2

The arrows show the mapping of the logical address space of each process to the physical memory available. In practice, the situation is more complex because the available memory actually consists of a mix of physical memory and memory pages that have been swapped out to disk.

The functions for shared memory resemble those for semaphores:

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

void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);
```

As with semaphores, the `include` files `sys/types.h` and `sys/ipc.h` are normally automatically included by `shm.h`.

shmget

You create shared memory using the `shmget` function:

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

As with semaphores, the program provides `key`, which effectively names the shared memory segment, and the `shmget` function returns a shared memory identifier that is used in subsequent shared memory functions. There's a special `key` value, `IPC_PRIVATE`, that creates shared memory private to the process. You wouldn't normally use this value, and you may find the private shared memory is not actually private on some Linux systems.

The second parameter, `size`, specifies the amount of memory required in bytes.

The third parameter, `shmflg`, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by `IPC_CREAT` must be bitwise ORed with the permissions to create a new shared memory segment. It's not an error to have the `IPC_CREAT` flag set and pass the key of an existing shared memory segment. The `IPC_CREAT` flag is silently ignored if it is not required.

The permission flags are very useful with shared memory because they allow a process to create shared memory that can be written by processes owned by the creator of the shared memory, but only read by processes that other users have created. You can use this to provide efficient read-only access to data by placing it in shared memory without the risk of its being changed by other users.

If the shared memory is successfully created, `shmget` returns a nonnegative integer, the shared memory identifier. On failure, it returns –1.

shmat

When you first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, you must attach it to the address space of a process. You do this with the `shmat` function:

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

The first parameter, `shm_id`, is the shared memory identifier returned from `shmget`.

The second parameter, `shm_addr`, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears.

The third parameter, `shmflg`, is a set of bitwise flags. The two possible values are `SHM_RND`, which, in conjunction with `shm_addr`, controls the address at which the shared memory is attached, and `SHM_RDONLY`, which makes the attached memory read-only. It's very rare to need to control the address at which shared memory is attached; you should normally allow the system to choose an address for you, because doing otherwise will make the application highly hardware-dependent.

If the `shmat` call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.

The shared memory will have read or write access depending on the owner (the creator of the shared memory), the permissions, and the owner of the current process. Permissions on shared memory are similar to the permissions on files.

An exception to this rule arises if `shmflg & SHM_RDONLY` is `true`. Then the shared memory won't be writable, even if permissions would have allowed write access.

shmdt

The `shmdt` function detaches the shared memory from the current process. It takes a pointer to the address returned by `shmat`. On success, it returns 0, on error –1. Note that detaching the shared memory doesn't delete it; it just makes that memory unavailable to the current process.

shmctl

The control functions for shared memory are (thankfully) somewhat simpler than the more complex ones for semaphores:

```
int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

The `shmid_ds` structure has at least the following members:

```
struct shmid_ds {
    uid_t shm_perm.uid;
    uid_t shm_perm.gid;
    mode_t shm_perm.mode;
}
```

The first parameter, `shm_id`, is the identifier returned from `shmget`.

The second parameter, `command`, is the action to take. It can take three values, shown in the following table.

Command	Description
IPC_STAT	Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
IPC_SET	Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.
IPC_RMID	Deletes the shared memory segment.

The third parameter, buf, is a pointer to the structure containing the modes and permissions for the shared memory.

On success, it returns 0, on failure, –1. X/Open doesn't specify what happens if you attempt to delete a shared memory segment while it's attached. Generally, a shared memory segment that is attached but deleted continues to function until it has been detached from the last process. However, because this behavior isn't specified, it's best not to rely on it.

Try It Out Shared Memory

Now that you've seen the shared memory functions, you can write some code to use them. In this Try It Out, you write a pair of programs, shm1.c and shm2.c. The first (the consumer) will create a shared memory segment and then display any data that is written into it. The second (the producer) will attach to an existing shared memory segment and allow you to enter data into that segment.

1. First create a common header file to describe the shared memory you want to pass around. Call this shm_com.h:

```
#define TEXT_SZ 2048

struct shared_use_st {
    int written_by_you;
    char some_text[TEXT_SZ];
};
```

   This defines a structure to use in both the consumer and producer programs. You use an int flag written_by_you to tell the consumer when data has been written to the rest of the structure and arbitrarily decide that you need to transfer up to 2k of text.

2. The first program, shm1.c, is the consumer. After the headers, the shared memory segment (the size of your shared memory structure) is created with a call to shmget, with the IPC_CREAT bit specified:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


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

```
#include "shm_com.h"

int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    int shmid;

    srand((unsigned int)getpid());

    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }
```

3. You now make the shared memory accessible to the program:

```
    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }

    printf("Memory attached at %X\n", (int)shared_memory);
```

4. The next portion of the program assigns the `shared_memory` segment to `shared_stuff`, which then prints out any text in `written_by_you`. The loop continues until `end` is found in `written_by_you`. The call to `sleep` forces the consumer to sit in its critical section, which makes the producer wait:

```
    shared_stuff = (struct shared_use_st *)shared_memory;
    shared_stuff->written_by_you = 0;
    while(running) {
        if (shared_stuff->written_by_you) {
            printf("You wrote: %s", shared_stuff->some_text);
            sleep( rand() % 4 ); /* make the other process wait for us ! */
            shared_stuff->written_by_you = 0;
            if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
                running = 0;
            }
        }
    }
```

5. Finally, the shared memory is detached and then deleted:

```
    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }

    if (shmctl(shmid, IPC_RMID, 0) == -1) {
```

```
            fprintf(stderr, "shmctl(IPC_RMID) failed\n");
            exit(EXIT_FAILURE);
        }

        exit(EXIT_SUCCESS);
    }
```

6. The second program, `shm2.c`, is the producer; it allows you to enter data for consumers. It's very similar to `shm1.c` and looks like this:

```
    #include <unistd.h>
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>


    #include <sys/shm.h>

    #include "shm_com.h"

    int main()
    {
        int running = 1;
        void *shared_memory = (void *)0;
        struct shared_use_st *shared_stuff;
        char buffer[BUFSIZ];
        int shmid;

        shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

        if (shmid == -1) {
            fprintf(stderr, "shmget failed\n");
            exit(EXIT_FAILURE);
        }
        shared_memory = shmat(shmid, (void *)0, 0);
        if (shared_memory == (void *)-1) {
            fprintf(stderr, "shmat failed\n");
            exit(EXIT_FAILURE);
        }

        printf("Memory attached at %X\n", (int)shared_memory);

        shared_stuff = (struct shared_use_st *)shared_memory;
        while(running) {
            while(shared_stuff->written_by_you == 1) {
                sleep(1);
                printf("waiting for client...\n");
            }
            printf("Enter some text: ");
            fgets(buffer, BUFSIZ, stdin);

            strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
            shared_stuff->written_by_you = 1;
```

```
            if (strncmp(buffer, "end", 3) == 0) {
                    running = 0;
            }
        }

        if (shmdt(shared_memory) == -1) {
            fprintf(stderr, "shmdt failed\n");
            exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
    }
```

When you run these programs, you get sample output such as this:

```
$ ./shm1 &
[1] 294
Memory attached at 40017000
$ ./shm2
Memory attached at 40017000
Enter some text: hello
You wrote: hello
waiting for client...
waiting for client...
Enter some text: Linux!
You wrote: Linux!
waiting for client...
waiting for client...
waiting for client...
Enter some text: end
You wrote: end
$
```

How It Works

The first program, `shm1`, creates the shared memory segment and then attaches it to its address space. You impose a structure, `shared_use_st` on the first part of the shared memory. This has a flag, `written_by_you`, which is set when data is available. When this flag is set, the program reads the text, prints it out, and clears the flag to show it has read the data. Use the special string, `end`, to allow a clean exit from the loop. The program then detaches the shared memory segment and deletes it.

The second program, `shm2`, gets and attaches the same shared memory segment, because it uses the same key, `1234`. It then prompts the user to enter some text. If the flag `written_by_you` is set, `shm2` knows that the client process hasn't yet read the previous data and waits for it. When the other process clears this flag, `shm2` writes the new data and sets the flag. It also uses the magic string `end` to terminate and detach the shared memory segment.

Notice that you had to provide your own, rather crude synchronization flag, `written_by_you`, which involves a very inefficient busy wait (by continuously looping). This keeps the example simple, however in real programs you would have used a semaphore, or perhaps passed a message, either using a pipe or IPC messages (which we discuss in the next section), or generated a signal (as shown in Chapter 11) to provide a more efficient synchronization mechanism between the reading and writing parts of the application.