UNIT 8.

Interprocess Communication Mechanisms

Stucture

8.0 Introduction

Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-Process Communication (IPC) mechanisms. Signals and pipes are two of them but Linux also supports the System V IPC mechanisms named after the Unix ™ release in which they first appeared. With System V, AT&T introduced three new forms of IPC facilities (message queues, semaphores, and shared memory).

8.1 Objectives

At the end of this unit, you would be able to

- Undestand the concept of Inter Process Communication Mechanisms
- Know about different IPC structures
- Distinguish among different types of communication between processes
- Describe the usage and manipulations of Message Queues
- Examine and illustrate Semaphores
- Elaborate the purpose of Shared Memory

8.1.1 System V IPC Mechanisms

Linux supports three types of interprocess communication mechanisms that first appeared in Unix ™ System V (1983). These are

i) message queues
ii) semaphores
iii) shared memory.

These System V IPC mechanisms all share common authentication methods. Processes may access these resources only by passing a unique reference identifier to the kernel via system calls. Access to these System V IPC objects is checked using access permissions, much like accesses to files are checked. The access rights to the System V IPC object is set by the creator of the object via system calls. The object's reference identifier is used by each mechanism as an index into a table of resources. It is not a straight forward index but requires some manipulation to generate the index.

All Linux data structures representing System V IPC objects in the system include an ipc_perm structure which contains the owner and creator process's user and group identifiers. The access mode for this object (owner, group and other) and the IPC object's key. The key is used as a way of locating the System V IPC object's reference identifier. Two sets of keys are supported: public and private. If the key is public then any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object. System V IPC objects can never be referenced with a key, only by their reference identifier.

8.1.2 IPC Identifiers

Each IPC *object* has a unique IPC identifier associated with it. When we say “IPC object", we are speaking of a single message queue, semaphore set, or shared memory segment. This identifier is used within the kernel to uniquely identify an IPC object. For example, to access a particular shared memory segment, the only item you need is the unique ID value which has been assigned to that segment.

The uniqueness of an identifier is relevant to the *type* of object in question. To illustrate this, assume a numeric identifier of ”12345". While there can never be two message queues with this same identifier, there exists the distinct possibility of a message queue and, say, a shared memory segment, which have the same numeric identifier.

8.1.3 IPC Keys

To obtain a unique ID, a *key* must be used. The key must be mutually agreed upon by both client and server processes. This represents the first step in constructing a client/server framework for an application.

When we use a telephone to call someone, we must know their number. In addition, the phone company must know how to relay your outgoing call to its final destination. Once the other party responds by answering the telephone call, the connection is made.

In the case of System V IPC facilities, the “telephone" correllates directly with the type of object being used. The “phone company", or routing method, can be directly associated with an IPC key.

The key can be the same value every time, by hardcoding a key value into an application. This has the disadvantage of the key possibly being in use already. Often, the ftok() function is used to generate key values for both the client and the server.

8.1.4 The ipcs Command

The ipcs command can be used to obtain the status of all System V IPC objects.

bash$ **ipcs -q**: Show only message queues bash$ **ipcs -s**: Show only semaphores bash$ **ipcs -m**: Show only shared memory bash$ **ipcs --help**: Additional arguments

By default, all three categories of objects are shown. Consider the following sample output of ipcs:

```
------ Shared Memory Segments --------
shmid     owner     perms     bytes     nattch    status

------ Semaphore Arrays --------
semid     owner     perms     nsems     status

------ Message Queues --------
msqid     owner     perms     used-bytes  messages
0          root       660        5              1
```

Here we see a single message queue which has an identifier of “0". It is owned by the user *root*, and has octal permissions of 660, or -rw-rw--. There is one message in the queue, and that message has a total size of 5 bytes.

The ipcs command is a very powerful tool which provides a peek into the kernel's storage mechanisms for IPC objects.

8.1.5 The ipcrm Command

The ipcrm command can be used to remove an IPC object from the kernel. While IPC objects can be removed via system calls in user code, the need often arises, especially under development environments, to remove IPC objects manually. Its usage is simple:

```
bash$  ipcrm <msg | sem | shm>  <IPC ID>
```

Simply specify whether the object to be deleted is a message queue (*msg*), a semaphore set (*sem*), or a shared memory segment (*shm*). The IPC ID can be obtained by the ipcs command. You have to specify the type of object, since identifiers are unique among the same type.

8.2 Message Queues

8.2.1 Basic Concepts

Message queues can be best described as an internal linked list within the kernel's addressing space. Messages can be sent to the queue in order and retrieved from the queue in several different ways. Each message queue (of course) is uniquely identified by an IPC identifier.

Message queues allow one or more processes to write messages, which will be read by one or more reading processes. Linux maintains a list of message queues, the msgque vector; each element of which points to a msqid_ds data structure that fully describes the message queue. When message queues are created a new msqid_ds data structure is allocated from system memory and inserted into the vector.

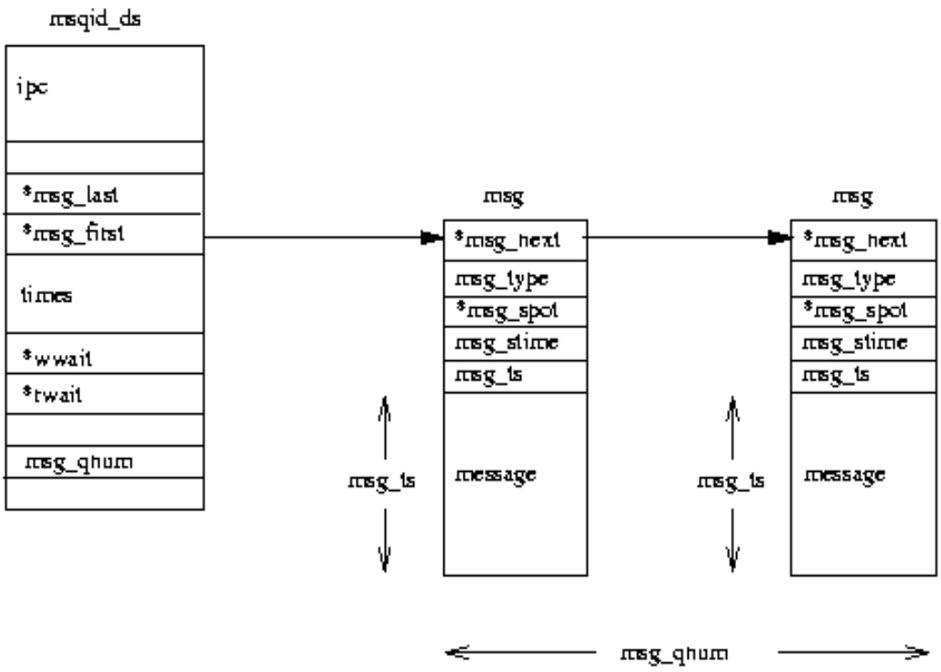

Figure 8.2: System V IPC Message Queues

Each msqid_ds data structure contains an ipc_perm data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. The msqid_ds also contains two wait queues; one for the writers to the queue and one for the readers of the message queue.

Each time a process attempts to write a message to the write queue its effective user and group identifiers are compared with the mode in this queue's ipc_perm data structure. If the process can write to the queue then the message may be copied from the

process's address space into a msg data structure and put at the end of this message queue. Each message is tagged with an application specific type, agreed between the cooperating processes. However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be woken up when one or more messages have been read from this message queue.

Reading from the queue is a similar process. Again, the processes access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match this criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be woken up and run again.

8.2.2 SYSTEM CALL: msgget()

In order to create a new message queue, or access an existing queue, the `msgget()` system call is used.

```
  SYSTEM CALL: msgget();

  PROTOTYPE: int msgget ( key_t key, int msgflg );
    RETURNS: message queue identifier on success
             -1 on error: errno = EACCESS (permission denied)
                                  EEXIST (Queue exists, cannot create)
                                  EIDRM (Queue is marked for deletion)
                                  ENOENT (Queue does not exist)
                                  ENOMEM (Not enough memory to create
queue)
                                  ENOSPC (Maximum queue limit exceeded)
```

The first argument to `msgget()` is the key value (in our case returned by a call to `ftok()`). This key value is then compared to existing key values that exist within the kernel for other message queues. At that point, the open or access operation is dependent upon the contents of the `msgflg` argument.

IPC_CREAT

Create the queue if it doesn't already exist in the kernel.

IPC_EXCL

When used with IPC_CREAT, fail if queue already exists.

If `IPC_CREAT` is used alone, `msgget()` either returns the message queue identifier for a newly created message queue, or returns the identifier for a queue which exists with the same key value. If `IPC_EXCL` is used along with `IPC_CREAT`, then either a new queue is created, or if the queue exists, the call fails with -1. `IPC_EXCL` is useless by itself, but when combined with `IPC_CREAT`, it can be used as a facility to guarantee that no existing queue is opened for access.

8.2.3 SYSTEM CALL: msgsnd()

Once we have the queue identifier, we can begin performing operations on it. To deliver a message to a queue, you use the `msgsnd` system call:

```
  SYSTEM CALL: msgsnd();

  PROTOTYPE: int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int
msgflg );
    RETURNS: 0 on success
-1 on error: errno = EAGAIN (queue is full, and IPC_NOWAIT was asserted)
                     EACCES (permission denied, no write permission)
                     EFAULT (msgp address isn't accessable - invalid)
                     EIDRM  (The message queue has been removed)
                     EINTR  (Received a signal while waiting to write)
                     EINVAL   (Invalid   message   queue   identifier,
nonpositive message type, or invalid message size)
                     ENOMEM (Not enough memory to copy message buffer)
```

The first argument to `msgsnd` is our queue identifier, returned by a previous call to `msgget`. The second argument, `msgp`, is a pointer to our redeclared and loaded message buffer. The `msgsz` argument contains the size of the message in bytes, excluding the length of the message type (4 byte long).

The `msgflg` argument can be set to 0 (ignored), or:

IPC_NOWAIT

If the message queue is full, then the message is not written to the queue, and control is returned to the calling process. If not specified, then the calling process will suspend (block) until the message can be written.

8.2.4 SYSTEM CALL: msgctl()

Through the development of the wrapper functions presented earlier, you now have a simple, somewhat elegant approach to creating and utilizing message queues in your applications. Now, we will turn the discussion to directly manipulating the internal structures associated with a given message queue.

To perform control operations on a message queue, you use the `msgctl()` system call.

```
  SYSTEM CALL: msgctl();
  PROTOTYPE: int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
    RETURNS: 0 on success
   -1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
                        EFAULT (Address pointed to by buf is invalid
with IPC_SET and IPC_STAT commands)
                        EIDRM  (Queue was removed during retrieval)
                        EINVAL (msgqid invalid, or msgsz less than 0)
                        EPERM  (IPC_SET or IPC_RMID command was issued,
but calling process does not have write (alter) access to the queue)
By using msgctl() with a selective set of commands, you have the ability
to manipulate those items which are less likely to cause grief. Let's
look at these commands:
```

IPC_STAT

Retrieves the msqid_ds structure for a queue, and stores it in the address of the buf argument.

IPC_SET

Sets the value of the ipc_perm member of the msqid_ds structure for a queue. Takes the values from the buf argument.

IPC_RMID

Removes the queue from the kernel.

8.3 Semaphores

8.3.1 Basic Concepts

Semaphores can best be described as counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it. Semaphores are often dubbed the most difficult to grasp of the three types of System V IPC objects. In order to fully understand semaphores, we'll discuss them briefly before engaging any system calls and operational theory.

In its simplest form a semaphore is a location in memory whose value can be tested and set by more than one process. The test and set operation is, so far as each process is concerned, uninterruptible or atomic; once started nothing can stop it. The result of the test and set operation is the addition of the current value of the semaphore and the set value, which can be positive or negative. Depending on the result of the test and set operation one process may have to sleep until the semphore's value is changed by another process. Semaphores can be used to implement *critical regions*, areas of critical code that only one process at a time should be executing.

Say you had many cooperating processes reading records from and writing records to a single data file. You would want that file access to be strictly coordinated. You could use a semaphore with an initial value of 1 and, around the file operating code, put two semaphore operations, the first to test and decrement the semaphore's value and the second to test and increment it. The first process to access the file would try to decrement the semaphore's value and it would succeed, the semaphore's value now being 0. This process can now go ahead and use the data file but if another process wishing to use it now tries to decrement the semaphore's value it would fail as the result would be -1. That process will be suspended until the first process has finished with the data file. When the first process has finished with the data file it will increment the semaphore's value, making it 1 again. Now the waiting process can be woken and this time its attempt to increment the semaphore will succeed.

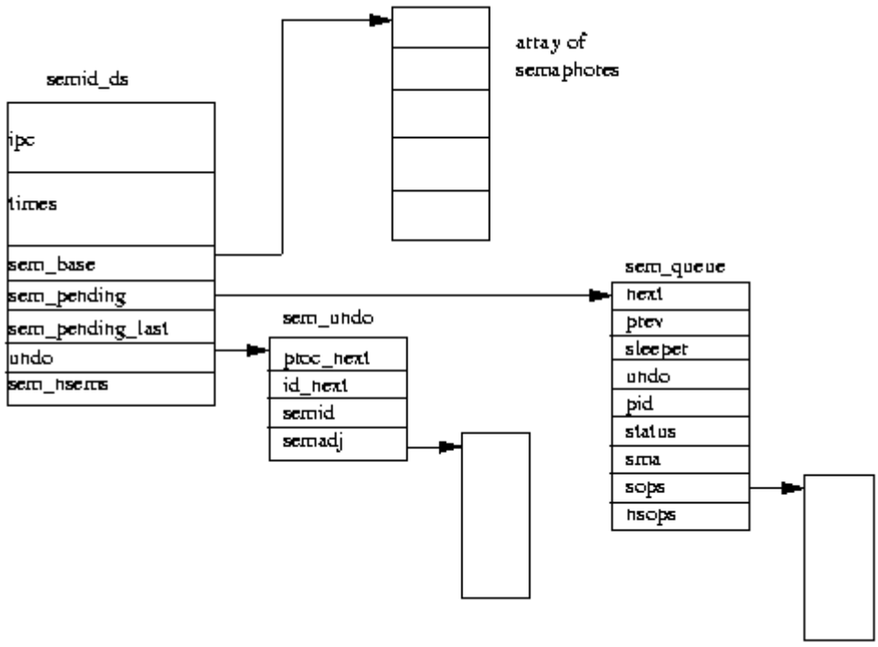

Figure 8.3: System V IPC Semaphores

System V IPC semaphore objects each describe a semaphore array and Linux uses the semid_ds data structure to represent this. All of the semid_ds data structures in the system are pointed at by the semary, a vector of pointers. There are sem_nsems in each semaphore array, each one described by a sem data structure pointed at by sem_base. All of the processes that are allowed to manipulate the semaphore array of a System V IPC semaphore object may make system calls that perform operations on them. The system call can specify many operations and each operation is described by three inputs; the semaphore index, the operation value and a set of flags. The semaphore index is an index into the semaphore array and the operation value is a numerical value that will be added to the current value of the semaphore. First Linux tests whether or not all of the operations would succeed. An operation will succeed if the operation value added to the semaphore's current value would be greater than zero or if both the operation value and the semaphore's current value are zero. If any of the semaphore operations would fail Linux may suspend the process but only if the operation flags have not requested that the system call is non-blocking. If the process is to be suspended then Linux must save the state of the semaphore operations to be performed and put the current process onto a wait queue. It does this by building a sem_queue data structure on the stack and filling it out. The new sem_queue data structure is put at the end of this semaphore object's wait queue (using the sem_pending and sem_pending_last pointers). The current process is put on the wait queue in the sem_queue data structure (sleeper) and the scheduler called to choose another process to run.

8.3.2 SYSTEM CALL: semget()

In order to create a new semaphore set, or access an existing set, the `semget()` system call is used.

```
  SYSTEM CALL: semget();

  PROTOTYPE: int semget ( key_t key, int nsems, int semflg );
    RETURNS: semaphore set IPC identifier on success
    -1 on error: errno = EACCESS (permission denied)
                         EEXIST (set exists, cannot create (IPC_EXCL))
                         EIDRM (set is marked for deletion)
                         ENOENT (set does not exist, no IPC_CREAT was
used)
                         ENOMEM (Not enough memory to create new set)
                         ENOSPC (Maximum set limit exceeded)
```

The first argument to `semget()` is the key value (in our case returned by a call to `ftok()`). This key value is then compared to existing key values that exist within the kernel for other semaphore sets. At that point, the open or access operation is dependent upon the contents of the `semflg` argument.

IPC_CREAT

Create the semaphore set if it doesn't already exist in the kernel.

IPC_EXCL

When used with IPC_CREAT, fail if semaphore set already exists.

8.3.3 SYSTEM CALL: semop()

```
  SYSTEM CALL: semop();
  PROTOTYPE: int semop ( int semid, struct sembuf *sops, unsigned
nsops);
    RETURNS: 0 on success (all operations performed)
    -1 on error: errno = E2BIG (nsops greater than max number of ops
allowed atomically)
                         EACCESS (permission denied)
                         EAGAIN (IPC_NOWAIT asserted, operation could
not go through)
                         EFAULT (invalid address pointed to by sops
argument)
                         EIDRM (semaphore set was removed)
                         EINTR (Signal received while sleeping)
                         EINVAL (set doesn't exist, or semid is invalid)
                         ENOMEM (SEM_UNDO asserted, not enough memory to
create the undo structure necessary)
                         ERANGE (semaphore value out of range)
```

The first argument to `semget()` is the key value (in our case returned by a call to `semget`). The second argument (`sops`) is a pointer to an array of *operations* to be performed on the semaphore set, while the third argument (`nsops`) is the number of operations in that array.

8.3.4 SYSTEM CALL: semctl()

```
      It Performs control operations on a semaphore set.

  SYSTEM CALL: semctl();
  PROTOTYPE: int semctl ( int semid, int semnum, int cmd, union semun
arg );
    RETURNS: positive integer on success
    -1 on error: errno = EACCESS (permission denied)
                         EFAULT  (invalid  address  pointed  to  by  arg
argument)
                         EIDRM (semaphore set was removed)
                         EINVAL (set doesn't exist, or semid is invalid)
                         EPERM (EUID has no privileges for cmd in arg)
                         ERANGE (semaphore value out of range)
```

The *semctl* system call is used to perform control operations on a semaphore set. This call is analogous to the *msgctl* system call which is used for operations on message queues. If you compare the argument lists of the two system calls, you will notice that the list for *semctl* varies slightly from that of *msgctl*. Recall that semaphores are actually implemented as sets, rather than as single entities. With semaphore operations, not only does the IPC key need to be passed, but the target semaphore within the set as well.

8.4 Shared Memory

8.4.1 Basic Concepts

Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process. This is by far the fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue, etc). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process, and subsequently written to and read from by any number of processes.

Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. The pages of the virtual memory is referenced by page table entries in each of the sharing processes' page tables. It does not have to be at the same address in all of the processes' virtual memory. As with all System V IPC objects, access to shared memory areas is controlled via keys and access rights checking. Once the memory is being shared, there are no checks on how the processes are using it. They must rely on other mechanisms, for example System V semaphores, to synchronize access to the memory.

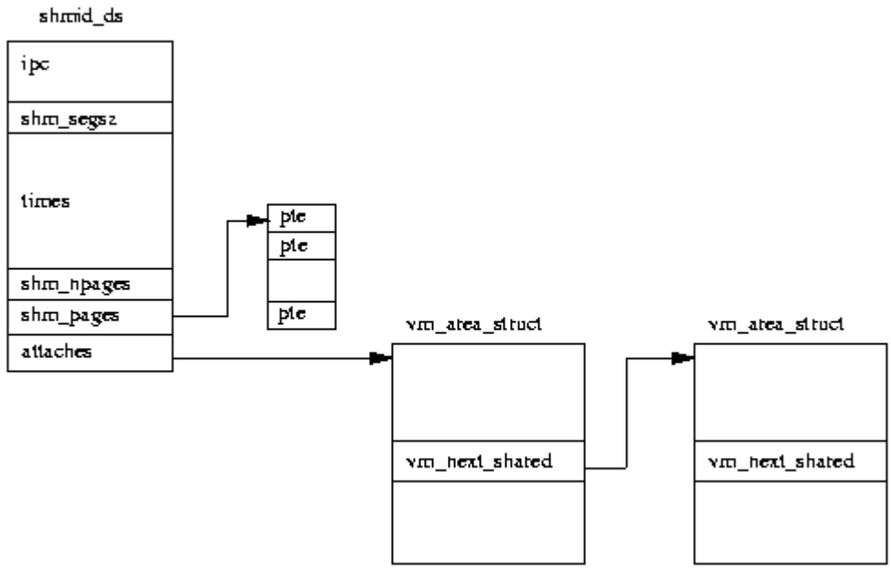

Figure 8.4: System V IPC Shared Memory

Each newly created shared memory area is represented by a shmid_ds data structure. These are kept in the shm_segs vector.

The shmid_ds data structure decribes how big the area of shared memory is, how many processes are using it and information about how that shared memory is mapped into their address spaces. It is the creator of the shared memory that controls the access permissions to that memory and whether its key is public or private. If it has enough access rights it may also lock the shared memory into physical memory.

Each process that wishes to share the memory must attach to that virtual memory via a system call. This creates a new vm_area_struct data structure describing the shared memory for this process. The process can choose where in its virtual address space the shared memory goes or it can let Linux choose a free area large enough. The new vm_area_struct structure is put into the list of vm_area_struct pointed at by the shmid_ds. The vm_next_shared and vm_prev_shared pointers are used to link them together. The virtual memory is not actually created during the attach; it happens when the first process attempts to access it.

When processes no longer wish to share the virtual memory, they detach from it. So long as other processes are still using the memory the detach only affects the current process. Its vm_area_struct is removed from the shmid_ds data structure and deallocated. The current process's page tables are updated to invalidate the area of virtual memory that it used to share. When the last process sharing the memory detaches from it, the pages of the shared memory current in physical memory are freed, as is the shmid_ds data structure for this shared memory.

8.4.2 SYSTEM CALL: shmget()

In order to create a new message queue, or access an existing queue, the `shmget()` system call is used.

```
SYSTEM CALL: shmget();
```

```
  PROTOTYPE: int shmget ( key_t key, int size, int shmflg );
    RETURNS: shared memory segment identifier on success
    -1 on error: errno = EINVAL (Invalid segment size specified)
                         EEXIST (Segment exists, cannot create)
                         EIDRM (Segment is marked for deletion, or was
removed)
                         ENOENT (Segment does not exist)
                         EACCES (Permission denied)
                         ENOMEM (Not enough memory to create segment)
```

This particular call should almost seem like old news at this point. It is strikingly similar to the corresponding `get` calls for message queues and semaphore sets.

The first argument to `shmget()` is the key value (in our case returned by a call to `ftok()`). This key value is then compared to existing key values that exist within the kernel for other shared memory segments. At that point, the open or access operation is dependent upon the contents of the `shmflg` argument.

Once a process has a valid IPC identifier for a given segment, the next step is for the process to `attach` or map the segment into its own addressing space.

8.4.3 SYSTEM CALL: shmat()

```
  SYSTEM CALL: shmat();

  PROTOTYPE: int shmat ( int shmid, char *shmaddr, int shmflg);
    RETURNS: address at which segment was attached to the process, or
    -1 on error: errno = EINVAL (Invalid IPC ID value or attach address
passed)
                         ENOMEM (Not enough memory to attach segment)
                         EACCES (Permission denied)
```

If the addr argument is zero (0), the kernel tries to find an unmapped region. This is the recommended method. An address can be specified, but is typically only used to facilitate proprietary hardware or to resolve conflicts with other apps. The SHM_RND flag can be OR'd into the flag argument to force a passed address to be page aligned (rounds down to the nearest page size).

8.4.4 SYSTEM CALL: shmctl()

```
  SYSTEM CALL: shmctl();
  PROTOTYPE: int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
    RETURNS: 0 on success
    -1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
                         EFAULT (Address pointed to by buf is invalid
with IPC_SET and IPC_STAT commands)
                         EIDRM  (Segment was removed during retrieval)
                         EINVAL (shmqid invalid)
                         EPERM  (IPC_SET or IPC_RMID command was issued,
but calling process does not have write (alter) access ro the segment)
```

This particular call is modeled directly after the *msgctl* call for message queues.

To properly detach a shared memory segment, a process calls the *shmdt* system call.

8.4.5 SYSTEM CALL: shmdt()

```
SYSTEM CALL: shmdt();

PROTOTYPE: int shmdt ( char *shmaddr );
  RETURNS: -1 on error: errno = EINVAL (Invalid attach address passed)
```

After a shared memory segment is no longer needed by a process, it should be detached by calling this system call. As mentioned earlier, this is not the same as removing the segment from the kernel! After a detach is successful, the shm_nattch member of the associates shmid_ds structure is decremented by one. When this value reaches zero (0), the kernel will physically remove the segment.

8.5 Summary

When processes wish to communicate, they must first establish communication. The stream mechanisms introduced in the Eighth Edition Unix system, which have now become part of AT&T's Unix System V, provide a flexible way for processes to conduct an already-begun conversation with devices and with each other: an existing stream connection is named by a file descriptor, and the usual read, write, and I/O control requests apply. Processing modules may be inserted dynamically into a stream connection, so network protocols, terminal processing, and device drivers separate cleanly. However, these mechanisms, by themselves, do not provide a general way to create channels between processes.

Simple extensions provide new ways of establishing communication. In our system, the traditional Unix IPC mechanism, the pipe, is a cross-connected stream.

A generalisation of file-system mounting associates a stream with a named file. When the file is opened, operations on the file are operations on the stream. Open files may be passed from one process to another over a pipe. These low-level mechanisms allow construction of flexible and general routines for connecting local and remote processes.

8.6 Check Your Progress

I. Choose the correct answer

1. Linux supporting Inter-Process Communication mechanisms are _____.

   a. Signals

   b. pipes

   c. both a & b

d. none of the above

2. If msgget() uses `IPC_CREAT` alone,then it returns _____.

a. the message queue identifier for a newly created message queue

b. the identifier for a queue which exists with the same key value

c. both a & b

d either a or b

3. msgctl() system call is used for _____.

a. controlling opeartions on a message queue

b. to deliver a message to a queue

c. to access an existing queue

d. none of the above

4. _____ is the mechanism used to prevent processes from accessing a particular resource while another process is performing operations on it.

a Shared Memory

b Semaphores

c Multi-processor

d none of the above

5. semget() system call is used _____.

a. to access an existing semaphore set

b. to perform control operations on a semaphore set

c. both a & b

d. none of the above

6. SYSTEM CALLS belongs to Shared Memory are _____.

a. shmget() & semget()

b. semctl() & shmctl()

c. shmget() & shmctl()

d. none of the above

II. Say True or False

1. ipc_perm structure contains the owner and creator process's user and group identifiers True/False
2. ipcs command can be used to obtain the status of all System V IPC objects. True/False

3. ipcrm command can be used to add an IPC object to the kernel.
   True/False
4. int msgsnd(int msqid,struct msgbuf *msgp, int msgsz, int msgflg)
   return: 1 on success . True/False
5 semaphore is a location in memory whose value can be tested and set by more than one process. True/False

III.Essay Type Questions

1. What is IPC? Explain different types of system V IPC mechanisms?
2. Explain the concept of Message Queues with neat diagram?
3. Write a note on the following SYSTEM CALLS:-
   i. semget()
   ii. msgsnd()
   iii msgctl()
4. How to control the access to shared resources by multiple processes.
5. What is meant by Shared Memory? Elaborate the different SYSTEM CALLS comes across it.

IV. Further readings and other activities

1. Get more information using man or info or any help command for
   ipcs, ipcrm, msgget, msgctl, semop, semctl, shmget, shmctl
   neconfig,userconf, ipcalc, fdisk, sfdisk.
   EX: 1. $ man ipcs 2. $ apropos ipc

2. Try various ipc programs on Unix or Linux platforms. Apreciate the secured way of communication between processes.
3. For further readings you can refer the following books
   1. Title: *Linux in a Nutshell*
      Author: Jessica Perry Hekman and the staff of O'Reilly & Associates;
      ISBN 1-56592-167-4; 1997. A complete reference for Linux.
   2. Title: Linux Kernel Internals
      Author: M. Beck etal
      Publication: Addison-Wesley, Second Edition
   3. Title: Unix System Programming
      Author: Richard Stevens

Reference e-mails: raomvp@yahoo.com roopasindhe@lycos.com
URL / Web Site: http://www.raomvp.bravepages.com

Good-Luck