System V Inter Process Communications
An Overview

This article introduces the basics of advanced interprocess communication mechanisms, like message queues, shared memory and semaphores that developers can use for their IPC implementations.

So far, we have seen the basics of inter process communications like pipes and FIFO in the earlier articles on IPC in LFY. In this article, we shall understand the rudiments of advanced interprocess communication mechanisms like message queues, shared memory and semaphores. All these mechanisms have two different implementations—Sys V IPC and POSIX IPC. We shall begin with Sys V IPC mechanisms.

The System V IPC package consists of

three mechanisms, namely message queues, shared memory and semaphores. These mechanisms can be used in between independent processes. Let us take a brief look at these mechanisms.

Message queues

Once a message queue is created along with an identifier, many messages in the form of formatted data streams can be stored in it. These messages can be accessed by an arbitrary process, if the message queue identifier is known and has the required access permissions. The messages will remain in the queue till any other processes retrieve them, or explicitly delete the message queue or reboot the system.

Shared memory

This memory is attached to the main memory with a specified segment size. Other processes that want to access the shared memory can also attach the shared memory segment into their own address space and then access that memory space just as they would access some memory-space after allocating through *malloc*. Since many processes can access the shared memory, any modification done by one process in the address space is visible to all other processes. Since the address space is a shared resource, the developer should implement a proper locking mechanism to prevent the race condition in the shared memory. Shared memory is one of the fastest mechanisms among all IPC mechanisms.

Semaphore

A semaphore is actually not a message passing or data sharing mechanism—it is a synchronisation tool. A semaphore function creates a set of semaphores, which can be used in complex situations—where a process needs to protect many critical sections from concurrent access.

Since Sys V IPC is implemented as a single unit, the mechanisms can share some of the common resources. For example, the kernel maintains a table to store entries of all instances of each mechanism. When a user calls IPC functions to create a resource, an IPC data structure is created dynamically. The ipc_perm structure is also included with the data structure and it has a key field for the corresponding resource. For each resource, the kernel uses an ipc_perm structure to verify access permissions to perform the operations, since it has stored the user and group IDs and read-write permissions whenever the IPC resources are created. The created resources are not known to the file system—so we need to use a different set of specific system calls to exploit IPC resources.

The System V IPC objects are created at the kernel level and the resources persist till explicitly deleted by the owner or rebooted off the system. The three mechanisms are similar in their interface to the user and system level implementations. But the interface differs for each mechanism for any specific issues. Using a standard set of

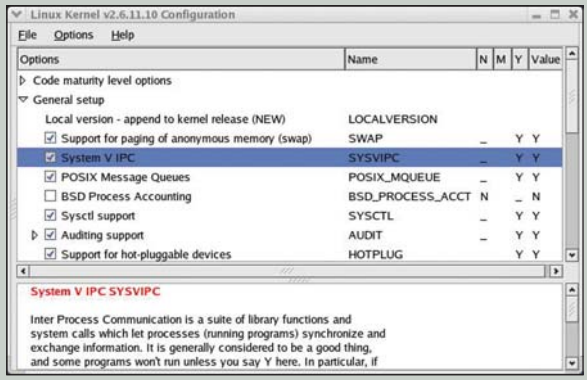

Figure 1: Kernel configuration to enable System V IPC

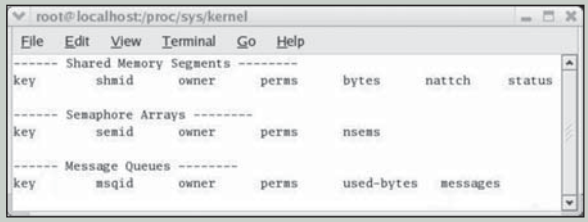

Figure 2: Template for knowing the status of IPC objects

system calls, a user can access these mechanisms. For this, the user should check whether the System V IPC feature is enabled in the kernel configuration or not. By default, it is enabled. If it is not enabled, the user can enable the option during the time of kernel configuration, as shown in Figure 1 and build the kernel to use the IPC mechanisms.

Linux uses different sets of functions to create and access IPC objects. First, it has to get a unique key value for creating a new identifier for a given object. After creating an object using some resources, specific functions will be used to access it. When we want to control objects, a control function with a set of commands is used. Some of the shell commands are also available to print the status and limitations of the IPC mechanisms.

The *key* function

If we wish to communicate between different processes using an IPC resource, the first step is to create a shared unique identifier. The simplest form of the identifier is a number—the system generates this number dynamically for a given mechanism by using the *ftok* library function. But apart from the creator, other processes that want to communicate with the creator process should agree to the key value.

The syntax of the *ftok* function is:

```
key_t ftok (const char *filename, int id);
```

The first argument file name should exist before you specify it into the *ftok*. If we specify the same filename and integer ID for different instances of the *ftok* function, we shall get the same key value, which is utilised to connect to

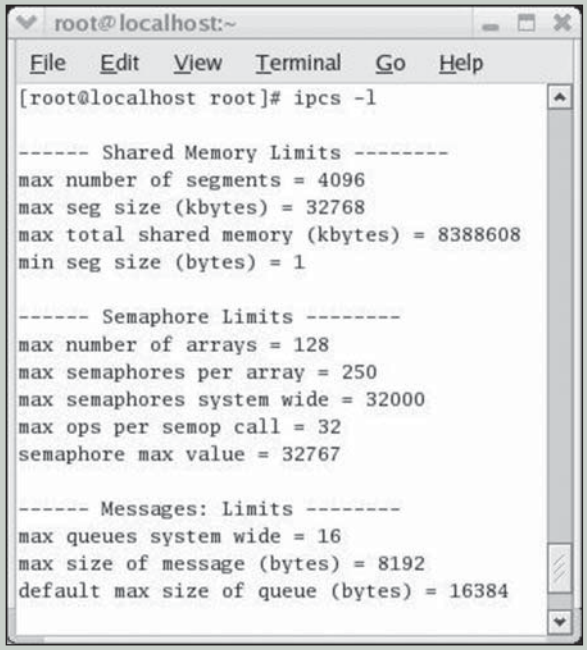

Figure 3: The system's in-built limit on IPC objects

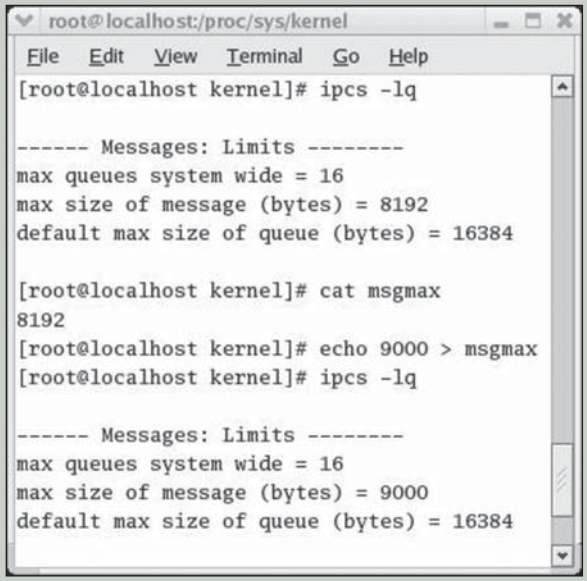

Figure 4: Modifying the system limitation to the maximum size of the message

a server from a client process. The return value datatype key_t is defined as a long integer in the <sys/types.h> file. If the function fails due to a wrong file name, it returns -1.

The *get* function

The generated key value is used to pass an argument to the *get* function, which is used to create an IPC resource or, if it has already been created, it is used to connect to the server. The *get* function takes two arguments—the key and the IPC flag. The key can be generated in three different ways—from the *ftok* library function, by choosing some static positive integer value and by using the IPC_PRIVATE macro. If the IPC_PRIVATE macro is specified as a key, the system guarantees that the process creates a new IPC structure. So if we want to connect to any existing IPC objects, we can't use IPC_PRIVATE as a key.

The syntax for the *get* function is:

```
int xxxget (key_t key, int xxxflg);   (xxx may be msg or shm
or sem)
```

If successful, this function returns to the identifier; otherwise it returns -1.

The flags commonly used with this function are IPC_CREAT and IPC_EXCL. The IPC_CREAT flag is used to create a resource if it doesn't exist. If IPC_EXCL is ORed with IPC_CREAT, the *get* function returns an error if the resource in question already exists. If the flag value is NULL, the kernel searches the same key value with appropriate access permissions for an existing resource. Commonly, the NULL value is specified in the client program since the server has already created the resource.

When a *get* function is called, internally it will call an ipc( ) system call. The syntax for the system call is:

```
int ipc(unsigned int call, int first, int second, int third,
void *ptr, long fifth);
```

The arguments depend on the calling function—for example, if a *semget* function is called, internally the arguments of the IPC function would be filled with *semget*, *key*, the number of the semaphore, semaphore flag and so on. The IPC system call is a common entry point for any of the IPC mechanisms. Depending on the arguments in the *get* function, the kernel invokes the requested service. At the user level this understanding of the *get* function is sufficient. Kernel hackers and library function implementers, however, should know the intricacies of the ipc( ) system call.

The *control* function

Each IPC mechanism has a control function to set some value of a resource or print the status of an entry; or to remove the resource itself.

The syntax for the *control* function is:

```
int xxxctl (int xxxid, int cmd, struct xxxid_ds *buffer);
(xxx may be msg or shm or sem);
```

The command argument may be IPC_STAT—for example, if the function is *msgctl* then the kernel copies the information from the message data structure associated with *msg_ds* into the structure pointed to the buffer. IPC_SET writes the changes of some members in the *msgid_ds* structure pointed to by the buffer to the message data structures. IPC_RMID removes the object

(continued on page 75)

with its associated data structures. A semaphore has some additional commands to get and set values of individual semaphores in the set. If successful, the *xxxctl* function returns zero, otherwise it returns -1.

Shell commands

The detailed man page for the IPC is "man 5 ipc"—it refers to the Linux implementation of message queues, semaphore sets and shared memory segments. The status of the IPC objects can be seen by executing "ipcs" from the shell, which will list all the mechanisms with their key value, ID, owner, permission and some other resource specific information (refer to Figure 2). The system stores this information into the */proc/sysvipc* directory. This directory contains three files, namely msg, sem and shm to store the status of the message queues, semaphores and shared memory respectively. The ipc objects have built-in limits, which can be displayed by using the "ipcs –l" command. "info ipcs" or "man 8 ipcs" will give more information about the *ipcs* commands together with the options you can use with it. Figure 3 shows the system limitation of the system V IPC mechanisms. The */proc/sys/kernel* directory contains the system's limitation value of each IPC object. If we want to change any particular value, it can be done by issuing *echo [new value]→filename.* Figure 4 illustrates this.

After using any created IPC object, it should be removed from the system. For this, we can either use a *xxxctl* function in the program itself or use the "ipcrm" command from the shell. To identify a particular IPC resource, the ipcrm command takes a key or identifier of a specified mechanism, which is to be removed along with its associated data structures from the system. Message queues and semaphores will be deleted immediately after issuing the ipcrm command, but shared memory is only removed after all currently attached processes have detached the object from their virtual address space. For more information about the different options that the ipcrm command accepts, issue "man 8 ipcrm" at the command-line.

In the next article in this series, we shall explore message queues along with their usage and internals—practically, and with the help of a lot of example programs.

By: Dr B. Thangaraju. *The author is a tech leader in Talent Transformation, Wipro Technologies, Bangalore. He can be reached at balat.raju@wipro.com*

Acknowledgement: *The author would like to thank Minjal Shah for her help during the preparation of this article.*