Interprocess Communications

- The basic IPC facilities discussed so far have been, for the most part, available in UNIX for many years. Now we will discuss some more sophisticated and more powerful IPC facilities.

- With System V, AT&T introduced three new forms of IPC facilities (**message queues**, **semaphores**, and **shared memory**).

- While the POSIX committee has not yet completed its standardization of these facilities, most implementations do support these. In addition, Berkeley (BSD) uses sockets as its primary form of IPC, rather than the System V elements.

- Linux has the ability to use both forms of IPC (BSD and System V); we will discuss sockets later.

- Generically, these new features are described as IPC facilities. This single descriptive term emphasizes similarities in structure and usage, although there are three distinct types of facility gathered under this heading, namely:

  1. **Message passing**

     This facility allows a process to send and receive messages; a message being in essence an arbitrary sequence of bytes or characters.

  2. **Semaphores**

     Compared with message passing, semaphores provide a rather low-level means for process synchronization, not suited to the transmission of large amounts of information.

  3. **Shared memory**

     This final IPC facility allows two or more processes to share the data contained in specific memory segments. It is probably the fastest of the IPC mechanisms, but requires some hardware support.

IPC Facility Keys

- The programming interface for all three IPC facilities has been made as similar as possible, reflecting a similarity of implementation within the kernel.

- The most important common feature is the **IPC facility key**. Keys are numbers used to identify an IPC object on a UNIX system in much the same way as a filename identifies a file.

- In other words, a key allows an IPC resource to be easily shared between several processes. The object identified can be a message queue, a set of semaphores or a shared memory segment.

- The actual data type of a key is determined by the implementation dependent type **key_t**, which is defined in the system header file **types.h**; **key_t** is equivalent to a long integer.

- Of course keys are not file names and carry less meaning. They should be chosen carefully to avoid clashes between programs, perhaps on the basis of “project numbers” assigned to different development projects.

- Some versions of UNIX provide a simple library function that maps a file's path name into a key. The routine is called ***ftok***:

```
key_t   mykey;
mykey = ftok("/tmp/foobar", 'a');
```

- In the above snippet, the directory **/tmp/foobar** is combined with the one letter identifier of **'a'**. Another common example is to use the current directory:

```
key_t   mykey;
mykey = ftok(".", 'a');
```

The *ipcs* Command

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

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

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

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

```
------ Semaphore Arrays --------
key           semid      owner      perms     nsems
0x00000000    1441792    apache     600       1
0x00000000    1474561    apache     600       1
0x00000000    1507330    apache     600       1


------ Shared Memory Segments --------
key           shmid     owner    perms    bytes     nattch    status
0x00000001    32768     root     600      655360    2

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

- In the message queue output we see a single message queue that 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, which has a total size of 5 bytes.

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:

  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 (recall our discussion of this earlier).

IPC get Operations

- A program uses a key to either create an IPC object or gain access to an existing one. Both options are called out by an IPC get operation.

- The result of a get operation is an integer IPC facility identifier, which can be used in calls to other IPC routines.

- If we pursue the file analogy further, the get operation is like a call to either **creat** or **open**, where the IPC facility identifier acts rather like a file descriptor.

- As an example the following statement uses the IPC call ***msgget*** to create a new message queue:

  msg_id = msgget ((key_t)0100, 0644|IPC_CREAT|IPC_EXCL);

- Here, the first argument to **msgget** is the message queue key. If successful, the routine will return a non-negative value in **msg_qid**, which acts as the message queue identifier.

Status Data Structures

- When an IPC object is created, the system also creates an IPC facility status structure that contains all administrative information associated with the object.

- This permission structure, is named **ipc_perm** and includes the following members:

```
struct ipc_perm
{
    key_t  key;
    ushort uid;          /* owner effective user-id and effective group-id */
    ushort gid;
    ushort cuid;         /* creator effective user-id and effective group-id */
    ushort cgid;
    ushort mode;         /* access modes */
    ushort seq;          /* slot usage sequence number */
};
```

- All of the above are fairly self-explanatory. Stored along with the IPC key of the object is information about both the creator and owner of the object (they may be different).

- The octal access modes are also stored here, as an **unsigned short**.

- This decides whether a user can 'read' and IPC object (that is obtain information on the object) or 'write' to it (which means manipulate it). The permissions are constructed in exactly the same way as with files.

- So the value 0644 for the **umode** member means that the owner can read and write the associated object, while other users can only read it.

- Finally, the **slot usage sequence** number is stored at the end. Each time an IPC object is closed via a system call (destroyed), this value gets incremented by the maximum number of IPC objects that can reside in a system.

Message Passing

- In essence, a message is simply a sequence of characters or bytes.

- Messages are passed between processes by means of message queues, which are created or accessed via the ***msgget*** primitive.

- Once a queue is established, a process may, given appropriate queue permissions, place a message onto it with ***msgsnd***.

- Another process can then read this message with ***msgrcv***, which also removes it from the queue.

- One can easily see from the above discussion that message passing in the IPC sense is similar to what can be achieved using read and write calls with pipes.

- The initialization function ***msgget*** is called as follows:

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

int msg_qid, permflags;
key_t key;
.
.
.
msg_qid = msgget (key, permflags);
```

- If the call is successful, and a new queue is created or an existing one accessed, **msg_qid** will hold an integer message queue identifier.

- The **permflags** parameter determines the exact action performed by ***msgget***. Two constants are of relevance here, both defined in the file **ipc.h**; they can be used alone, or bitwise ORed together:

- **IPC_CREAT**

    - This tells ***msgget*** to create a message queue for the value key if one does not already exist. If the IPC_CREAT flag isn't set, then a message queue identifier is only returned by ***msgget*** if the queue already exists.

- **IPC_EXCL**

    - If this and IPC_CREAT are both set, then the call is intended only to create a message queue. Setting this without setting the IPC_CREAT has no meaning.

- When a queue for key already exists, ***msgget*** will fail and return -1.

- When a message queue is created, the low-order 9 bits of **permflags** are used to give the permissions for the message queue, rather like a file mode.

- These are stored in the **ipc_perm** structure, which is created along with the queue itself. Look at our previous example:

```
msgq_id = msgget ((key_t)0100, 0644|IPC_CREAT|IPC_EXCL);
```

- This call is intended to create (and only create) a message queue for the key value **(key_t)100**. If the call is successful, the queue will have permissions 0644.

- If necessary, ***msgctl*** can be used later to alter the permissions and ownerships associated with the queue.

- The following is a wrapper function for opening or creating message queue:

```
int open_queue (key_t keyval)
{
        int     qid;

        if( (qid = msgget (keyval, IPC_CREAT | 0660 ) ) == -1 )
        {
                return (-1);
        }

        return (qid);
}
```

- The kernel stores each message in the queue within the framework of the **msg** structure. It is defined for us in **linux/msg.h** as follows:

```
/* one msg structure for each message */
struct msg {
     struct msg *msg_next;          /* next message on queue */
     long  msg_type;
     char *msg_spot;                /* message text address */
     short msg_ts;                  /* message text size */
};
```

- **msg_next**
    - This is a pointer to the next message in the queue. They are stored as a singly linked list within kernel addressing space.

- **msg_type**
    - This is the message type, as assigned in the user structure **msgbuf**.

- **msg_spot**
    - A pointer to the beginning of the message body.

- **msg_ts**
    - The length of the message text, or body.

The msgop Routines: msgsend and msgrcv

- Once a queue has been created, there are two **msgop** primitives which can be used to manipulate it.

The first of these is ***msgsnd***:

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

int msg_qid, size, flags, retval;
struct my_msg
{
        long mtype;
        char mtext[SOMEVALUE];
}message;
.
.
retval = msgsnd (msg_qid, &message, size, flags);
```

- ***msgsnd*** is used to send a message to the queue denoted by **msg_qid**, the value of which will normally have been obtained from ***msgget***.

- The message itself is contained in the structure message. The **mtype** member can be used by the programmer to categorize messages; each possible value representing a different potential category.

- The length of the message to be actually sent is given by the size parameter to ***msgsnd***, and this can range from zero to the smaller of SOMEVALUE or a system determined maximum.

- The flags parameter for ***msgsnd*** can take just one meaningful value: **IPC_NOWAIT**. If **IPC_NOWAIT** is not set, the calling process will sleep if there are insufficient system resources to send a message.

- If **IPC_NOWAIT** is set, the call will return immediately if the message can't be sent. Its return value will then be –1.

- ***msgsnd*** can also fail because of the permissions associated with the message queue. If, for example, the user has neither the effective user-id nor the effective group-id associated with the queue, and the queue permissions are 0060, then a call to ***msgsnd*** for that queue will fail.

- The following is a useful wrapper function for sending messages:

```
int send_message( int msg_qid, struct msgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof (struct msgbuf) – sizeof(long);

        if( (result = msgsnd ( msg_qid, qbuf, length, 0) ) == –1)
        {
                return (–1);
        }

        return (result);
}
```

- The following example utilizes the two wrapper functions we have developed so far:

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

int main (void)
{
        int    qid;
        key_t  msgkey;

        struct msgbuf {
                long   mtype;           // Message type
                int   command;          // Message function
                double   data_size;     // Number of octets
        } msg;

        // Generate our IPC key value
        msgkey = ftok (".", 'm');

        // Open/create the queue
        if ((qid = open_queue (msgkey)) == –1)
        {
                perror("open_queue");
                exit(1);
        }

        // Load the message structure with arbitrary test data
        msg.mtype   = 1;                // Message type must be a positive number
        msg.command = 10;               // Command #10
        msg.data_size  = 1024;          // 1024 octets

        // Send it
        if((send_message( qid, &msg )) == -1)
        {
                perror("send_message");
                exit(1);
        }
}
```

- The other routine that comes under the ***msgop*** heading is ***msgrcv***.

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

                int msg_qid, size, flags, retval;
                struct my_msg
                {
                        long mtype;
                        char mtext[SOMEVALUE];
                }message;
                long msg_type;
                .
                .
                .
                retval = msgrcv (msg_qid, &message, size, msg_type, flags);
```

- ***msgrcv*** is used to read a message off the queue identified by **msg_qid**, providing the queue's permissions allow the process to do this. The act of reading a message causes it to be removed from the queue.

- **message** is used to hold a received message, and the **size** parameter gives the maximum length that can be held within the structure.

- The third argument (**size**) represents the size of the message buffer structure, excluding the length of the **mtype** member. This can be calculated as:

```
        size = sizeof (struct msgbuf) - sizeof(long);
```

- If the call is successful, then **retval** will contain the length of the received message.

- The **msg_type** parameter decides exactly what message is actually received. It selects according to the value of a message's **mtype** field.

- If **msg_type** is zero, the first message on the queue, i.e. the earliest sent, is read. If **msg_type** has a non-zero, positive value then the first message with that value is read.

- For example, if the queue contains messages with **mtype** values of 999, 5 and 1, and ***msgrcv*** is called with **msg_type** set to **5**, then the message of **type 5** is read.

- Finally, if **msg_type** has a non-zero, **negative value**, the first message with the lowest **mtype** number that is less than or equal to **msg_type**'s absolute value is received.

- The last parameter flags again contains control information. Two values, **IPC_NOWAIT** and **MSG_NOERROR**, can be set, alone or OR'ed together.

- **IPC_NOWAIT** means much as it did before; if it isn't set then the process sleep is there isn't a suitable message on the queue, returning when a message of the appropriate type arrives. If it is set, the call will return immediately, whatever the circumstances.

- If **MSG_NOERROR** is set, a message will be truncated if it is no longer than size bytes, otherwise the call to ***msgrcv*** would fail. There is unfortunately no way of knowing when truncation has occurred.

- The following is a wrapper function that can be used to retrieve a message from our queue:

```
int read_message (int qid, long type, struct msgbuf *qbuf )
{
        int     result, length;

        // The length is essentially the size of the structure minus sizeof(mtype)
        length = sizeof(struct msgbuf) – sizeof(long);

        if ( (result = msgrcv ( qid, qbuf, length, type,  0) ) == –1)
        {
                return(–1);
        }

        return(result);
}
```

The *msgctl* System Call

- The ***msgctl*** routine serves three purposes: it allows a process to get status information about a message queue, to change some of the limits associated with a message queue, or to delete a queue from the system altogether.

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

int msg_qid, command, retval;
struct msqid_ds msq_stat;
.
.
.
retval = msgctl (msg_qid, command, &msq_stat);
```

- **msg_qid** is, of course, a valid message queue identifier. The parameter **&msq_stat** is the address of a **msqid_ds** structure.

- Each of the three types of IPC objects has an internal data structure that is maintained by the kernel.

- For message queues, this is the **msqid_ds** structure. The kernel creates, stores, and maintains an instance of this structure for every message queue created on the system. It is defined in **linux/msg.h** as follows:

```
// one msqid structure for each queue on the system
struct msqid_ds {
   struct ipc_perm msg_perm;
   struct msg *msg_first;        // first message on queue
   struct msg *msg_last;         // last message in queue
   time_t msg_stime;             // last msgsnd time
   time_t msg_rtime;             // last msgrcv time
   time_t msg_ctime;             // last change time
   struct wait_queue *wwait;
   struct wait_queue *rwait;
   ushort msg_cbytes;
   ushort msg_qnum;
   ushort msg_qbytes;            // max number of bytes on queue
   ushort msg_lspid;             // pid of last msgsnd
   ushort msg_lrpid;             // last receive pid
};
```

- **msg_perm**
    - An instance of the **ipc_perm** structure, which is defined for us in linux/ipc.h. This holds the permission information for the message queue, including the access permissions, and information about the creator of the queue (uid, etc).
- **msg_first**
    - Link to the first message in the queue (the head of the list).
- **msg_last**
    - Link to the last message in the queue (the tail of the list).
- **msg_stime**
    - Timestamp (time_t) of the last message that was sent to the queue.
- **msg_rtime**
    - Timestamp of the last message retrieved from the queue.
- **msg_ctime**
    - Timestamp of the last "change" made to the queue.
- **wwait** and **rwait**
    - Pointers into the kernel's **wait queue**. They are used when an operation on a message queue deems the process go into a sleep state (i.e. queue is full and the process is waiting for an opening).
- **msg_cbytes**
    - Total number of bytes residing on the queue (sum of the sizes of all messages).
- **msg_qnum**
    - Number of messages currently in the queue.
- **msg_qbytes**
    - Maximum number of bytes on the queue.
- **msg_lspid**
    - The PID of the process that sent the last message.
- **msg_lrpid**
    - The PID of the process that retrieved the last message.

- The types ushort and time_t are system dependent and are defined in **types.h**. Typically they will reduce to unsigned short and long respectively.

- The command parameter to ***msgctl*** tells the system what operation is to be performed. There are three options available here, all of which apply to all three IPC facilities.

- They are defined by constants defined in **ipc.h**.

- **IPC_STAT**

    - Tells the system to place status information about the structure into **msq_stat**.

- **IPC_SET**

    - Used to set the values of control variables for the message queue, according to information held within **msq_stat**. Only the following items can be changed:

      msg_stat.msg_perm.uid
      msg_stat.msg_perm.gid
      msg_stat.msg_perm.mode
      msg_stat.msg_qbytes

- An **IPC_SET** operation will succeed only if executed by superuser or the current owner of the queue as indicated by **msq_stat.msg_perm.uid**.

- In addition, only the superuser can increase the **msg_qbytes** limit, which gives the maximum number of characters that may exist on the queue at any one time.

- **IPC_RMID**

    - This removes the message queue from the system. Again, this can be done only by superuser or the queue owner.

- The following is a wrapper function that will retrieve the internal structure and copy it into a passed address:

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
        if( msgctl( qid, IPC_STAT, qbuf) == -1)
        {
                return (-1);
        }

        return (0);
}
```

- IPC objects remain in the system unless explicitly removed, or the system is rebooted. Therefore, our message queue still exists within the kernel, available for use long after a single message disappears.

- Good programming practice dictates that they should be removed with a call to ***msgctl(),*** using the **IPC_RMID** command.

- The following wrapper function returns 0 if the queue was removed successfully, or else a value of –1:

```
int remove_queue( int qid )
{
        if( msgctl (qid, IPC_RMID, 0) == -1)
        {
                return (-1);
        }

        return (0);
}
```

- The following example ***show_msg*** prints out some of the status information associated with a message queue. It is intended to be invoked as:

```
$ show_msg keyvalue
```

- **show_msg** uses the **ctime** library routine to convert **time_t** values to readable form.