Message Passing Between Processes Through A Message Queue

Message queues overcome some inherent limitations of FIFO. With the help of examples, we look at some basic operations concerning message queues in this write up.

Although interprocess communication using FIFO is simple, it has some disadvantages. We cannot store any information inside the FIFO, since it has a zero buffering capacity. The retrieval of any specific information is also not possible using FIFO. Also, both the read and write communication ends must be kept open in the FIFO approach. Otherwise, since read and write are blocking calls, it is impossible to establish a connection. A message queue overcomes these limitations quite well.

Understanding message queues

The concept of a message queue is required to help create a queue at the system level. Each message is then identified by a message queue identifier, which is created by passing some unique key values and suitable IPC flag(s). When a process wants to access an existing message queue, it should have the access permission and must use the correct message queue ID value.

This article begins with explaining the procedure of creating a message queue, sending a message to the queue and receiving a message from the message queue. Finally, it explains the controlling commands—which are used to modify the message queue parameters. Many examples are given to illustrate these concepts clearly. The internals of the message queue related system calls are also explained with the help of kernel source code. The complete source code is not taken up for discussion, because of its sheer volume and complexity. Only some important functions and instructions are discussed to explain the internals of a message queue.

Creating a message queue

Since we have already discussed the ftok function in some detail in a previous LFY article, here we shall directly go into the msgget function, which is used for creating a message queue. The syntax of the msgget function is as shown below:

```
int msgget (key_t key, int msgflg);
```

The first argument key can be passed from the return value of the ftok function or made IPC_PRIVATE. To create a message queue, IPC_CREAT ORed with access permission is set for the msgflg argument. Once a message queue is created, if any process wants to connect to this existing queue, the same key value and 0 for msgflg should be passed to the msgget function. Passing IPC_CREAT with the access permission instead of 0 will also work. But, if IPC_CREAT ORed with IPC_EXCL is passed, the msgget function will fail. If we pass IPC_PRIVATE as a key, then we won't be able to connect to the existing message queue since every execution of the msgget function, along with the IPC_PRIVATE key, will create a new message queue. Instead, we need to pass a message queue ID directly to message queue related functions like msgsnd, msgrcv or msgctl. A program to create a message queue is included on the LFY CD (ipcprog1). If the queue has already been created, the mesgget function returns an error. If the queue is created successfully, then it will print the allocated key and message queue identifier values. After the execution of the program is complete, the status of the message queue can be checked by executing the *ipcs –q* command at the shell prompt.

From the message queue source code (/usr/src/linux-2.6.x/ipc/msg.c), we can understand the actual functionality of the msgget function. The corresponding function for the creation of a message queue is sys_msgget. First, it checks the key value:

```
if (key = = IPC_PRIVATE)
    ret = newque(key, msgflg);
```

The newque function allocates the memory for a new message queue descriptor and initialises some of the message queue data structures. Subsequently, a new queue ID is returned to the caller.

If the key value is already given to an existing queue, it will try to find out whether the key value is valid or not, by calling the ipc_findkey function:

```
id = ipc_findkey(&msg_ids, key)
```

If this fails (the key doesn't exist), then you need to check msgflg using IPC_CREATE and return with a new ID by making use of the following function:

```
if (!(msgflg & IPC_CREAT))
   ret = -ENOENT;
else
   ret = newque(key, msgflg);
```

If the ipc_findkey returns successful (the key exists), then check msgflg by using:

```
if (msgflg & IPC_CREAT && msgflg & IPC_EXCL)
  ret = -EEXIST;
```

If the flag is IPC_CREAT | IPC_EXCL, then the above returns with an error; otherwise, it will check the permission through:

```
if (ipcperms(&msq->q_perm, msgflg))
    ret = -EACCES;
```

If a process has suitable permissions, the message queue ID is returned, else an error is returned. Most of these operations are synchronised using a suitable global lock.

Allocation of resources

When a new message queue is created using the msgget system call, the kernel creates the relevant data structures and initialises their value. Two very important structures are:

```
struct ipc_perm
struct msqid_ds
```

The structure details are listed on the LFY CD (ipcprog2 and ipcprog3). During the creation of a message queue, the members of the structures are initialised. cuid, uid, gid and cgid are set to the effective uid and the effective gid of the calling processes respectively. The number of messages currently on the queue, the process ID of the sender and receiver, and the time of the sender and the receiver are all set to zero. The creation time is set to the current time and the maximum number of bytes allowed on the queue is set to the system limit. ipcprog4 on the LFY CD prints the relevant information regarding the two structures. The system limitations are stored in the msginfo structure. ipcprog5 on the LFY CD shows the msginfo structure and ipcprog6 shows how we may print the members of the msginfo structure using a program.

Sending a message to the queue

Each message is composed of a structure with a minimum of two fields, namely the message type and the message text. However, depending on the user requirement, more members can be added to the structure. The first member should always be a long int. The msgsnd function is called to send a message to an existing queue. The syntax of the function is:

```
int msgsnd (int msqid, structu msgbuf *msgp, size_t msgsz,
int msgflg);
```

The first argument is the message queue ID and the second argument is the address of the structure. So, before calling the msgsnd function, we need to fill the structure. The third argument is the size of the message text and the fourth argument is the message flag. The flag value may either be 0 or IPC_NOWAIT. The message queue is visible at the system level and the created messages are stored in a kernel buffer. Since kernel space is limited, if there is no space available to store a message, the system reaches its maximum limit of the number of messages that can be stored in the queue. The process can then specify whether it can wait till more space is available to store a message (if msglfg is 0) or returns with an error without waiting (if msgflg is set to IPC_NOWAIT). When it is waiting for space to store a message, if the queue is removed or if the process receives a signal, the system call will fail. On the success of msgsnd, some of the data structures are modified—for example, the process ID is set to msg_lspid and msg_qunum is incremented by 1. Further, msg_stime is set to the current time. The file ipcprog7 on the LFY CD shows a program to send a message to the queue.

msgsnd is explained with sys_msgsnd functions in the kernel source code. It checks the message size, the message queue ID, and the message type.

```
if (msgsz > msg_ctlmax || (long) msgsz < 0 || msqid < 0)
      return -EINVAL;
if (get_user(mtype, &msgp->mtype))
     return -EFAULT;
if (mtype < 1)
    return -EINVAL;
```

Then, the user message is loaded by executing the load_msg function. Later, the type and size of the message are initialised.

```
msg = load_msg(msgp->mtext, msgsz);
.....................
msg->m_type = mtype;
msg->m_ts = msgsz;
```

Now, the message queue and permission are validated by calling the msg_checkid and ipcperms functions:

```
msg_checkid(msq,msqid);

ipcperms(&msq->q_perm, S_IWUGO);
```

We may now check the available space in the message queue and compare it with the message size. This is evaluated by using:

```
if(msgsz + msq->q_cbytes <= msq->q_qbytes && 1 + msq->q_qnum <=
msq->q_qbytes)
```

If there is enough space, the kernel updates the sending process ID and time, using:

```
msq->q_lspid = current->tgid;
msq->q_stime = get_seconds();
```

Now, the calls pipelined_send function is called using:

```
if(!pipelined_send(msq,msg)) {
        list_add_tail(&msg->m_list,&msq->q_messages);
        msq->q_cbytes += msgsz;
        msq->q_qnum++;
        atomic_add(msgsz,&msg_bytes);
        atomic_inc(&msg_hdrs);
 }
```

If any receiver is waiting for the message, the pipelined_send function directly sends the message to the receiver instead of storing it in the message queue. To find out the first receiver who is waiting for the message, the testmsg function is used. If no receiver is waiting, then the message is added at the tail end of the queue and the relevant data structures are updated.

If enough space is not available in the queue, then the kernel checks msgflg. If it is IPC_NOWAIT, then msgsnd returns with an error.

```
if(msgflg&IPC_NOWAIT)
err = - EAGAIN;
```

Otherwise, the current process is put into the sender wait queue. When the process is awakened, the kernel checks whether the message queue ID and permission is still valid or not. Then it checks if any signal is pending. If there is a signal pending, then the kernel frees the message buffer and the system call returns with an error. Otherwise, the function again checks the availability of space.

Receiving a message

To receive any message, the msgrcv function is called. The syntax of the function is:

```
ssize_t msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz,
long msgtype, int msgflg);
```

The arguments are the same as those in the msgsnd function, except for the fourth argument, which is used to retrieve a particular message by specifying a message type. If the given message size is less than the actual message size and the msgflg is not set to MSG_NOERROR, then the system call returns with an error. However, if the msgflg is set to MSG_NOERROR, then the message will be received by the process, but (actual size–given message size) the text will be lost. If the message type is 0, the process reads the message in the FIFO order. If it is any positive value, the exact value of the message type is received by the process. If the value is negative, then the process receives the first message on the queue, whose type value is less than or equal to the absolute value. If the msgflag is 0 and the requested message type is not in the message queue, the process will wait till some sender sends the message. But if the msgflag is set to IPC_NOWAIT, the system call will not wait for a message—it will simply return with an error. When the receiver process is waiting for the message, if the queue is removed or the process receives any signal, then the system call returns with an error. If successful, msgrcv returns with the number of bytes actually copied into the message text array, updates msg_lrpid as the calling pid, and decrements msg_qnum by 1. msg_time is set to the current time. The file ipcprog8 on the LFY CD lists a program to receive a message without waiting for an unavailable message in the queue—even if the given message size is less then the actual size.

The msgrcv system call's internals can be explained using the sys_msgrcv function, which is in the kernel source code. First, it checks the message queue ID and the message size, and then calls the convert_mode function:

```
if (msqid < 0 || (long) msgsz < 0)
                return -EINVAL;
        mode = convert_mode(&msgtyp,msgflg);
```

The convert mode function returns in the search mode to the calling process, based on its argument's message type and message flag. The actual implementation of finding a desired message to retrieve is done by this function.

```
static inline int convert_mode(long* msgtyp, int msgflg)
{
        if(*msgtyp = = 0)
                return SEARCH_ANY;
        if(*msgtyp < 0) {
                *msgtyp = - (*msgtyp);
                return SEARCH_LESSEQUAL;
        }
        if(msgflg & MSG_EXCEPT)
                return SEARCH_NOTEQUAL;
        return SEARCH_EQUAL;
}
```

Then, the kernel checks the access permission of the process and verifies the message size. Next, it removes the message from the message queue and updates the message queue parameters. It then wakes up all the processes waiting in the 'sender waiting' queue:

```
list_del(&msg->m_list);
 msq->q_qnum –;
 msq->q_rtime = get_seconds();
 msq->q_lrpid = current->tgid;
 msq->q_cbytes -= msg->m_ts;
 atomic_sub(msg->m_ts,&msg_bytes);
 atomic_dec(&msg_hdrs);
 ss_wakeup(&msq->q_senders,0);
```

If a given message type does not match with the available messages in the queue, then the kernel checks the msgflg argument. If it is IPC_NOWAIT, the system call returns with an error message; otherwise, the process will be put in the waiting list of the receivers:

```
if (msgflg & IPC_NOWAIT)
  msg = ERR_PTR(-ENOMSG);
.....................................................
list_add_tail(&msr_d.r_list,&msq->q_receivers);
```

If msgflg is set to MSG_NOERROR, then the message size is set to the given size. Once the message is available, the kernel fills the user message structure and frees the memory where the message was previously stored in the message queue. The put_user function copies the message type and the store_msg function copies the message text into the given address of the structure, in the msgrcv function.

```
if (put_user (msg->m_type, &msgp->mtype) ||
            store_msg(msgp->mtext, msg, msgsz))
                    msgsz = -EFAULT;

        free_msg(msg);
```

Finally, the message is removed from the queue by executing the free_msg function.

Controlling a message queue

So far, we have seen how to create a message queue, and then how to send and receive a message to the created queue. If we want to know the status of the message queue, desire to modify some of the parameters of the existing message queue, or need to remove the message queue itself, then we should call the function msgctl. The syntax of the function is:

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

This function carries out operations specified by the command on the given message queue. There are three ways in which it can perform operations—reading the status of the message queue parameters, modifying any parameters that are of interest, or removing the message queue itself. IPC_STAT, IPC_SET and IPC_RMID are used to retrieve the status, set parameters and remove a message queue respectively. ipcprog9 on the LFY CD shows how to modify a user and access permission of an existing message queue. ipcprog10 is a program to remove the message queue.

The three commands are implemented through the switch case statements. Depending on the command specified in the msgctl system call, the corresponding case statements in the message queue's kernel source code will be executed. In all these cases, the kernel first checks the message queue ID and the command, and then verifies the access permission of the calling process. In the IPC_STAT case, it first declares the temporary buffer of the message queue data structure. After filling the members of the temporary structures from the message queue information, it is copied to the user specified address. In the case of IPC_SET, the user data is copied into the kernel buffer and the message queue parameters are updated according to that. In the case of IPC_RMID, the freeque( ) function is called. This frees the resources of the specified message queue and removes the message queue.

Limitations

The problem with message queues is that when you send a message to a queue, the message structure is copied from the user buffer to the kernel buffer, and during the message retrieval it is again copied from the kernel buffer to the user buffer. To overcome this and other such overheads, we can keep our data in a memory segment—probably physical memory—and can give access permission, and the address of the segment, to the relevant processes. This approach uses the concept of a shared memory. END

By: Dr B. Thangaraju. *The author is working as a technical lead with Talent Transformation, Wipro Technologies, Bangalore. He can be reached at [balat.raju@wipro.com.]*