Ex No. 2B. IPC Using Message Queue

Objective

Develop a program to make a sender and receiver processes communicate using message queue.

Prepared By S. Rajarajan APIII/CSE

IPC:Message Queues:<sys/msg.h>

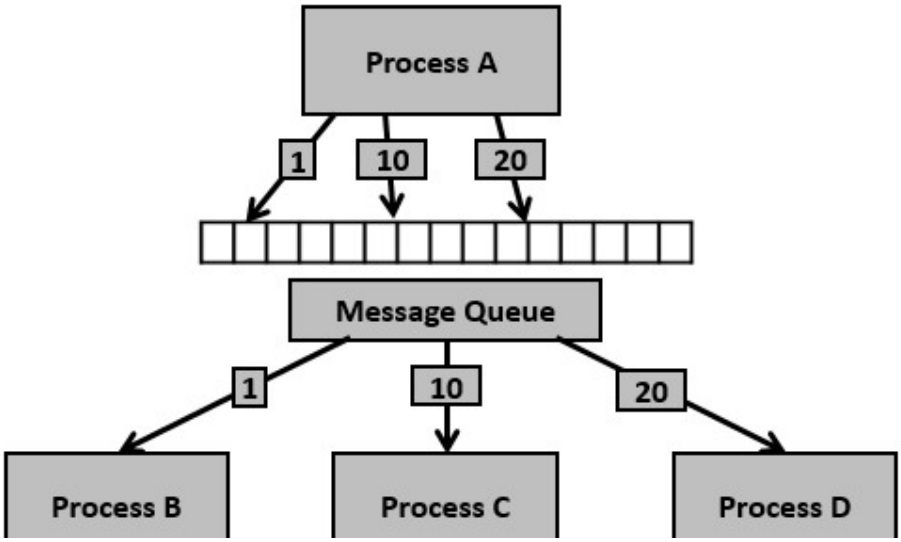
Why do we need message queues when we already have the shared memory? It would be for multiple reasons, let us try to break this into multiple points for simplification –

- Once the message is received by a process it would be no longer available for any other process. Whereas in shared memory, the data is available for multiple processes to access.
- If we want to communicate with small message formats.
- Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.
- If frequency of writing and reading using the shared memory is high, then it would be very complex to implement the functionality.
- What if all the processes do not need to access the shared memory but very few processes only need it, it would be better to implement with message queues.
- If we want to communicate with different data packets, say process A is sending message type 1 to process B, message type 10 to process C, and message type 20 to process D. In this case, it is simpler to implement with message queues

Using Shared Memory or Message Queues depends on the need of the application and how effectively it can be utilized.

Communication using message queues can happen in the following way –

- Writing into the message queue by one process with different data packets and reading from it by multiple processes, i.e., as per message type.



Message queues provide an **asynchronous communications protocol**, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. Message queues have **implicit or explicit limits** on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue.

To perform communication using message queues, following are the steps –

Step 1 – Create a message queue or connect to an already existing message queue (msgget())

Step 2 – Write into message queue (msgsnd())

Step 3 – Read from the message queue (msgrcv())

Step 4 – Perform control operations on the message queue (msgctl())

Before a process can send or receive a message, the queue must be initialized (through the ***msgget*** function see below) Operations to send and receive messages are performed by the ***msgsnd***() and ***msgrcv***() functions, respectively.

When a message is sent, its text is copied to the message queue. The msgsnd() and msgrcv()functions can be performed as either ***blocking or non-blocking operations***. A **Non-blocking** operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been

Initializing the Message Queue

The msgget() function initializes a new message queue:

int msgget(key_t key, int msgflg)

- The first argument, key, recognizes the message queue. The key can be either an arbitrary value or one that can be derived from the library function ftok().
- The second argument, shmflg, specifies the required message queue flag/s such as IPC_CREAT (creating message queue if not exists) or IPC_EXCL (Used with IPC_CREAT to create the message queue and the call fails, if the message queue already exists). Need to pass the permissions as well.

This call would return a valid message queue identifier (used for further calls of message queue) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

Sending and Receiving Messages

The msgsnd() and msgrcv() functions send and receive messages, respectively:

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

This system call sends/appends a message into the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, msgp, is the pointer to the message, sent to the caller in the following form:

  struct msgbuf { long mtype; char mtext[1]; };

  The variable mtype is used for communicating with different message types
- The third argument, msgsz, is the size of message (the message should end with a null character)
- The fourth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in queue or MSG_NOERROR (truncates message text, if more than msgsz bytes)

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

The msqid argument must be the ID of an existing message queue. The msgp argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
    long    mtype;    /* message type */
    char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

The msgsz argument specifies the length of the message in bytes.

The fouth argument, msgtype, indicates the type of message −

- **If msgtype is 0** − Reads the first received message in the queue
- **If msgtype is +ve** − Reads the first message in the queue of type msgtype (if msgtype is 10, then reads only the first message of type 10 even though other types may be in the queue at the beginning)
- **If msgtype is –ve** − Reads the first message of lowest type less than or equal to the absolute value of message type (say, if msgtype is -5, then it reads first message of type less than 5 i.e., message type from 1 to 5)

The argument msgflg specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to msg_qbytes.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (msgflg & IPC_NOWAIT) is non-zero, the message will not be sent and the calling process will return immediately.
- If (msgflg & IPC_NOWAIT) is 0, the calling process will suspend execution until one of the following occurs:
    - The condition responsible for the suspension no longer exists, in which case the message is sent.
    - The message queue identifier msqid is removed from the system; when this occurs, errnois set equal to EIDRM and -1 is returned.
    - The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

Upon successful completion, the following actions are taken with respect to the data structure associated with msqid:

- msg_qnum is incremented by 1.
- msg_lspid is set equal to the process ID of the calling process.
- msg_stime is set equal to the current time.

Controlling message queues

The msgctl() function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using msgctl() Also, any process with permission to do so can use msgctl() for control operations.

The msgctl() function is prototypes as follows:

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

The msqid argument must be the ID of an existing message queue. The cmd argument is one of:

IPC_STAT

-- Place information about the status of the queue in the data structure pointed to by buf. The process must have read permission for this call to succeed.

IPC_SET

-- Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC_RMID

-- Remove the message queue specified by the msqid argument.

Sender Program

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#define MAX 10

// structure for message queue
```

```
struct mesg_buffer {
   long mesg_type;
   char mesg_text[100];
} message;

int main()
{
        key_t key;
        int msgid;
        // ftok to generate unique key
        key = ftok("progfile", 65);
        // msgget creates a message queue
        // and returns identifier
        msgid = msgget(key, 0666 | IPC_CREAT);
        //message.mesg_type = 1;
        printf("Writing Data : ");
        printf("Enter the message:");
        scanf("%s",message.mesg_text);
        do
        {
                printf("Enter the type for message:");
                scanf("%ld",&message.mesg_type);
        // msgsnd to send message
                msgsnd(msgid, &message, sizeof(message), 0);
                // display the message
                //printf("Data send is : %s \n", message.mesg_text);
                printf("Enter the message:");
                scanf("%s",message.mesg_text);
        }while(strcmp(message.mesg_text,"end")!=0);
   return 0;
}
```

Receiver Program

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#define MAX 10

// structure for message queue
struct mesg_buffer {
   long mesg_type;
   char mesg_text[100];
} message;

int main()
```

```
{
        key_t key;
        int msgid,type;
        char choice[10];
        key = ftok("progfile", 65);
        msgid = msgget(key, 0666 | IPC_CREAT);
        printf("Read Data : ");
        do{
                printf("Enter the type of the message: ");
                scanf("%d",&type);
                msgrcv(msgid, &message, sizeof(message),type, 0);
                printf("Message is : %s \n", message.mesg_text);
                printf("Do you want to continue: ");
                scanf("%s",choice);
        }while(strcmp(choice,"no")!=0);
        msgctl(msgid, IPC_RMID, NULL);
        return 0;
}
```