MESSAGE QUEUE

Message Queues give you another way to send information from one process to another. There is overlap in Message Queue capabilities and Named Pipes. However, there are some substantial differences:

1. named pipe (**FiFo**) is byte oriented, **message q** is packet oriented.

2.Message Queues allow you to take packets **out of order** in some cases.

3. Each message has a **message type** associated with it. A Message Queue reader can specify which type of message that it will read. Or it can say that it will read all messages in order.

4.It is quite possible to have **any number of Msg Queue readers, or writers**. In fact the same process can be both a writer and a reader.

- One real **bad characteristic**: Msg Queues are **based on a system buffer resource**. It is possible for one process to let its messages pile up. This can result in all processes on a given system to be hung up because the system is out of resources. This is bad news when it happens.
- There is a **limit to the size of each packet and there is a limit to the total number of bytes** that can show up in any given Message Queue.

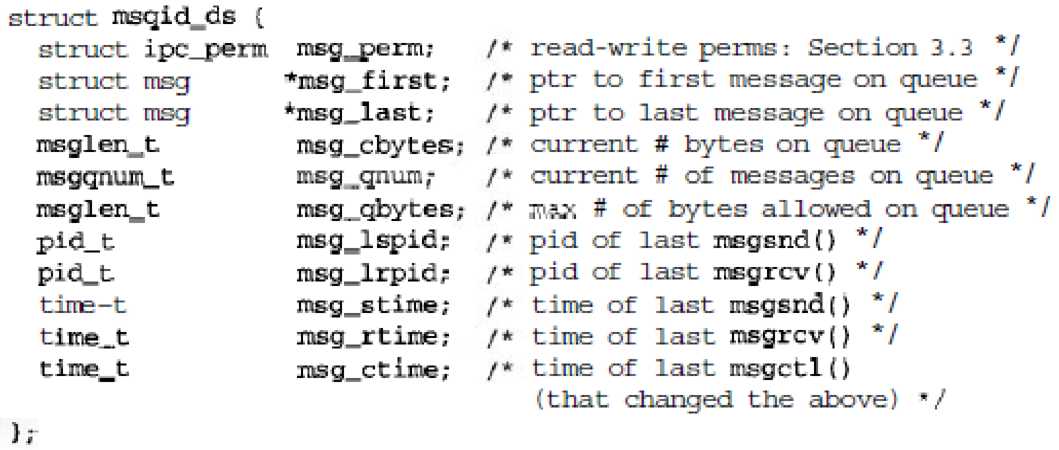
acer@ubuntu**:~$ ipcs**

gives all Inter Process Communication current objects are displayed as below

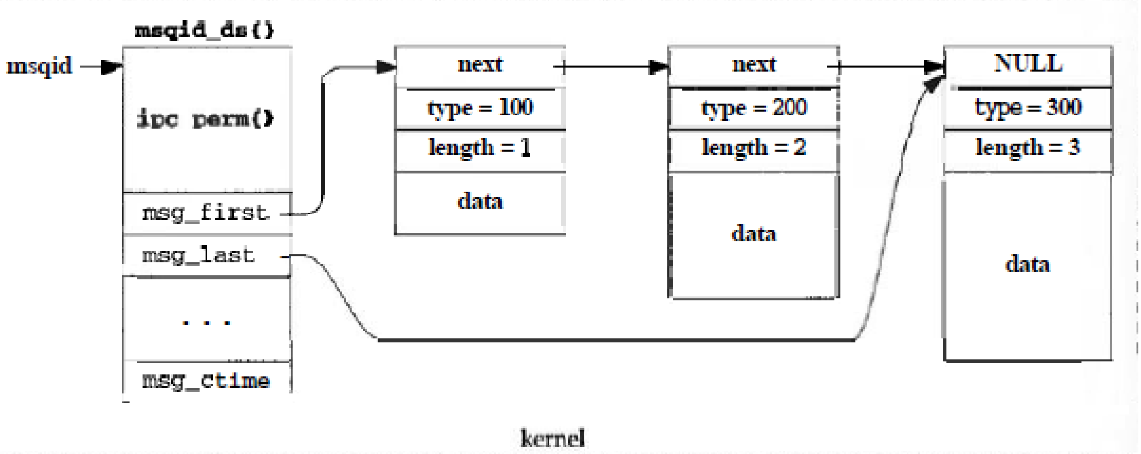
acer@ubuntu**:~$ ipcs -q**

gives all message queues are displayed.

For every message queue in the system, the **kernel maintains the following structure of information**, defined by including *<sys* **/msg . h>:**



```
struct msqid_ds {
  struct ipc_perm  msg_perm;    /* read-write perms: Section 3.3  */
  struct msg      *msg_first;   /* ptr to first message on queue */
  struct msg      *msg_last;    /* ptr to last message on queue */
  msglen_t         msg_cbytes;  /* current # bytes on queue */
  msgqnum_t        msg_qnum;    /* current # of messages on queue */
  msglen_t         msg_qbytes;  /* max # of bytes allowed on queue */
  pid_t            msg_lspid;   /* pid of last msgsnd() */
  pid_t            msg_lrpid;   /* pid of last msgrcv() */
  time-t           msg_stime;   /* time of last msgsnd() */
  time_t           msg_rtime;   /* time of last msgrcv() */
  time_t           msg_ctime;   /* time of last msgctl()
                                   (that changed the above) */
};
```



Creating Message Queue - msgget

NAME

msgget - get a message queue identifier

SYNOPSIS

#include <sys/msg.h>

int msgget(key_t key, int msgflg);

DESCRIPTION

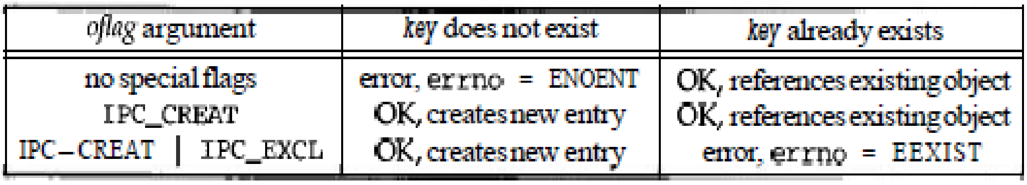
The msgget() system call returns the message queue identifier associated with the value of the *key* argument. The **msgget** function is used either to **create a new message queue or to locate an existing queue based on a key**. Message queues are implemented by UNIX System Services, and allow messages of multiple types to be queued. UNIX System Services message queues support multiple senders and multiple receivers and do not require the senders and receivers to be running simultaneously. Message queues, once created using **msgget** , remain in existence until they are explicitly destroyed with a call to **msgctl** .

The **key** argument is an integral value which identifies the message queue desired. A **key** value of **IPC_PRIVATE** requests **a new message queue without an associated key** , and which can be accessed only by the process in which **queue id returned by msgget** .

The **flags** argument specifies zero or more option flags specifying whether or not the queue already exists, and how access to the queue should be regulated. The argument should be specified as **0** for no flags, or as one or more of the following symbolic constants, combined using the or operator (|):

□ **IPC_CREAT** - Specifies that if a queue with the requested **key does not exist**, **it should be created**. This flag is ignored if **IPC_PRIVATE** is specified.

□ **IPC_EXCL** - Specifies that a queue with the requested **key** must not already exist. This flag is ignored if **IPC_PRIVATE** is specified, or if **IPC_CREAT** is not specified.



oflag argument	*key* does not exist	*key* already exists
no special flags	error, errno = ENOENT	OK, references existing object
IPC_CREAT	OK, creates new entry	OK, references existing object
IPC–CREAT | IPC_EXCL	OK, creates new entry	error, errno = EEXIST

The other argument, *msgflg* tells **msgget()** what to do with queue in question. To create a queue, this field must be set equal to **IPC_CREAT bit-wise OR'd with the permissions** for this queue. (The queue permissions are the same as standard file permissions—queues take on the user-id and group-id of the program that created them.)

i.e

The flags include permissions and can **optionally contain IPC_CREAT and IPC_EXCL**. IPC_CREAT creates a message queue if it doesn't already exist. If **IPC_EXCL is included with IPC_CREAT** then it is considered a failure if the message queue does already exist. Some valid flag combinations are:

0660

0660 | IPC_CREAT

0600 | IPC_CREAT | IPC_EXCL

If ***msgflg*** specifies both IPC_CREAT and IPC_EXCL and a message queue already exists for key, then msgget() fails with errno set to EEXIST.

RETURN VALUE

If **successful**, the return value will be the **message queue identifier** (a nonnegative integer), **otherwise -1 with *errno*** indicating the error.

Example: key =1234 (some key value hard coded)

```
msqid = msgget(key, 0666 | IPC_CREAT);
```

Sending and Receiving message MSGSND and MSGRCV

NAME

msgrcv, msgsnd - message operations

SYNOPSIS

```
#include <sys/msg.h>

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

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

The **msgsnd()** and **msgrcv()** system calls are used, respectively, to send messages to, and receive messages from, a message queue. The calling process(**sender**) must have **write permission on the message queue** in order to send a message, and **read permission (receiver process) to receive a message**.

1st Argument:

msqid – message queue id of message queue created (using *msgget()*)

2nd Argument:

msgp – is a **pointer to caller-defined structure** of the following general form:

```
struct msgbuf {
    long mtype;      /* message type, must be > 0 */
    char mtext[1];   /* message data */
 };
```

If you want to send data - RegNo , Name through message queue then **msgbuf** declaration can be as below-

```
struct msgbuf {
    long mtype;      /* message type, must be > 0 */
    int  RegNo;
    char Name[18];   /* message data */
 };
```

mtext field is an array (or other structure) whose size is specified by msgsz **(see below)**, a nonnegative integer value. Messages of **zero** length (i.e., no mtext field) are permitted.
mtype- The mtype field must have a **strictly positive integer value**. This value can be used by the **receiving process for message selection**.

2^{nd} Argument in msgsnd () and msgrcv() is pointer to the above msgbuf structure.

mtype in msgbuf can be as below-

3rd Argument

msgsz- specifies size of the message to be put on message queue. To get the size of the data to send, just **subtract** the **sizeof(long)** (the *mtype*) from the **sizeof()** the **whole message** buffer structure:

Exampe:

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

4th Argument

msgflag- The msgflg argument is a bit mask **constructed by ORing together zero or more of the following flags**:

IPC_NOWAIT

In case of msgsnd() , returns error if message queue is full.

In case of msgrcv(), Return immediately if **no message of the requested type** is in the queue. The system call fails with errno set to ENOMSG.

MSG_EXCEPT

Used with **msgtyp greater than 0** to **read the first message** in the queue with **message type that differs from msgtyp**.

MSG_NOERROR

To **truncate** the message text if **longer than msgsz** bytes.

When msgflg=0

If **no message** of the **requested type** is available and **IPC_NOWAIT** isn't specified in msgflg, *(i.e. if msgflg set as 0 , then called process* ***runs in blocking mode****)* the calling process is **blocked until** one of the following conditions occurs:

* A message of the desired type is placed in the queue.

* The message queue is removed from the system. In this case the system call fails with errno set to EIDRM.

* The calling process **catches a signal**. In this case the system call fails with errno set to **EINTR**. (msgrcv() is never automatically restarted after being interrupted by a signal handler, regardless of the setting of the SA_RESTART flag when establishing a signal handler.)

Upon successful completion the message queue data structure is updated as follows:

msg_lrpid is set to the process ID of the calling process.

msg_qnum is decremented by 1.

msg_rtime is set to the current time.

5th Argument

msgtyp Argument in msgrcv()

Actually, the behavior of **msgrcv()** can be modified drastically by choosing a *msgtyp* that is **positive, negative, or zero**:

msgtyp	**Effect on msgrcv()**
Zero	Retrieve the next message on the queue(FIFO order), regardless of its *mtype*.
Positive	Get the next message with an ***mtype*** ***equal to*** **the specified *msgtyp***.
Negative	Retrieve the first message on the queue whose *mtype* field is less than or equal to the absolute value of the *msgtyp* argument.

RETURN VALUE

On failure **both** functions **return -1 with errn**o indicating the error, otherwise **msgsnd()** ***returns 0*** and **msgrcv()** returns the **number of bytes** actually copied into the mtext array.

Example:

```
/*
IPC Message Queue Implementation in C
A simple implementation of IPC Message Queues.
IPC_msgq_send.c adds the message on the message queue .
IPC_msgq_rcv.c removes the message from the message queue.

To use this program first compile and run IPC_msgq_send.c to add a message to the message queue.
To see the Message Queue type ipcs -q on your Unix/Linux Terminal.

Now compile and run IPC_msgq_rcv.c to read the message from the Message Queue.
To see that you have read the message again use ipcs -q
*/
//IPC_msgq_send.c

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXSIZE     128

void die(char *s)
{
  perror(s);
  exit(1);
}

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

main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    struct msgbuf sbuf;
    size_t buflen;

    key = 1234;

    if ((msqid = msgget(key, msgflg )) < 0)   //Get the message queue ID for the given key
      die("msgget");

    //Message Type
    sbuf.mtype = 1;

    printf("Enter a message to add to message queue : ");
    scanf("%[^\n]",sbuf.mtext);
    getchar();

    buflen = strlen(sbuf.mtext) + 1 ;
// syntax:   int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

    if (msgsnd(msqid, &sbuf, buflen, IPC_NOWAIT) < 0)
    {
        printf ("%d, %ld, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buflen);
        die("msgsnd");
    }

    else
        printf("Message Sent\n");

    exit(0);
}
```

Message receiver

```
//IPC_msgq_rcv.c

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE     128

void die(char *s)
{
  perror(s);
  exit(1);
}

typedef struct msgbuf
{
   long    mtype;
   char    mtext[MAXSIZE];
} ;


main()
{
   int msqid;
   key_t key;
   struct msgbuf rcvbuffer;

   key = 1234;

   if ((msqid = msgget(key, 0666)) < 0)
     die("msgget()");


    //Receive an answer of message type 1.
/* Syntax : ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int
msgflg);  */
   if (msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0) < 0)
     die("msgrcv");

   printf("%s\n", rcvbuffer.mtext);
   exit(0);
}
```

FTOK

NAME

ftok - convert a pathname and a project identifier to a System V IPC key

SYNOPSIS

```
#include <sys/ipc.h>
```

key_t ftok(const char *pathname, int proj_id);

DESCRIPTION

The ftok() function uses the identity of the file named by the given pathname (which must refer to an existing, accessible file) and the least significant 8 bits of proj_id (which must be nonzero) to generate a key_t type System V IPC key, suitable for use with msgget()

MSGCTL(2)   Linux Programmer's Manual   MSGCTL(2)

NAME

msgctl - message control operations

SYNOPSIS

```
#include <sys/msg.h>

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

msqid- message queue of id.
cmd- Valid values for **cmd** are:

IPC_RMID

Immediately remove the message queue, awakening all waiting reader and writer processes (with an error return and errno set to EIDRM). The calling process must have appropriate privileges or its effective user ID must be either that of the creator or owner of the message queue.

IPC_STAT

Copy information from the kernel data structure associated with msqid into the msqid_ds structure pointed to by buf. The caller must have read permission on the message queue.

IPC_SET

Write the values of some members of the msqid_ds structure pointed to by buf to the kernel data structure associated with this message queue, updating also its msg_ctime member, msg_qbytes, msg_perm.uid, msg_perm.gid etc.

msqid_ds -The **msqid_ds** data structure is defined in <sys/msg.h> as follows:
For every message queue created the system maintains following informations in the form of following structure.

```
struct msqid_ds {
    struct ipc_perm msg_perm;     /* Ownership and permissions */
    time_t          msg_stime;    /* Time of last msgsnd(2) */
    time_t          msg_rtime;    /* Time of last msgrcv(2) */
    time_t          msg_ctime;    /* Time of last change */
    unsigned long   __msg_cbytes; /* Current number of bytes in
                                queue (nonstandard) */
    msgqnum_t       msg_qnum;     /* Current number of messages
                                in queue */
    msglen_t        msg_qbytes;   /* Maximum number of bytes
                                       allowed in queue */
    pid_t           msg_lspid;    /* PID of last msgsnd(2) */
    pid_t           msg_lrpid;    /* PID of last msgrcv(2) */
};
```

ipc_perm- The **ipc_perm s**tructure is defined in <sys/ipc.h> as follows (the highlighted fields are settable using IPC_SET):

```
struct ipc_perm {
        key_t          __key;       /* Key supplied to msgget(2) */
        uid_t          uid;         /* Effective UID of owner */
        gid_t          gid;         /* Effective GID of owner */
        uid_t          cuid;        /* Effective UID of creator */
        gid_t          cgid;        /* Effective GID of creator */
        unsigned short mode;        /* Permissions */
        unsigned short __seq;       /* Sequence number */
      };
```

*** Example-** ***msgctl(msqid, IPC_RMID,0)***

This will kill message queue if IPC_RMID is used , 3rd parameter always set to 0 when IPC_RMID is used.

msgctl(msqid, IPC_STAT,&qstatus)

qstatus must be of type structure msqid_ds.

This will copy the content of msqid_ds (generated by kernel for the message queue created with id msqid) into user defined structure qstatus. We can use qstatus information in the program.

/* This program gives the use of msgctl() to collect queue status information from msgqid_ds structure. */

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


int main()
/*      argc,argvint argc;
        char* argv[];*/
{
        int qid;
        struct msqid_ds qstatus;

        //qid=msgget((key_t)atoi(argv[1]),IPC_CREAT);

                qid=msgget(1120,0666|IPC_CREAT);
        if(qid==-1)
        {
                perror("Message Q creation failed \n");
                exit(1);
        }
        else
        {
                if(msgctl(qid,IPC_STAT,&qstatus)<0)
                {
                        perror("MSGCTL failed \n");
                        exit(1);
                }
                else
                {
                        printf(" User ID of creator %d\n", qstatus.msg_perm.cuid);
                        printf(" Group ID of creator %d\n", qstatus.msg_perm.cgid);
                        printf(" Effective user-ID of creator %d\n", qstatus.msg_perm.uid);
                        printf(" Effective Group-ID of creator %d\n", qstatus.msg_perm.gid);
                        printf(" Permissions %o\n",qstatus.msg_perm.mode);
                        printf(" Message Queue ID %d\n",qstatus.msg_lspid);
                }
        }
}
```