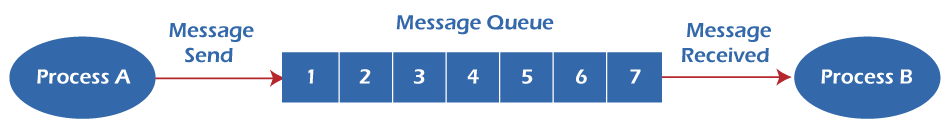
IPC using Message Queues

A **message queue** is an inter-process communication (IPC) mechanism that allows processes to exchange data in the form of messages between two processes. It allows processes to communicate asynchronously by sending messages to each other where the messages are stored in a queue, waiting to be processed, and are deleted after being processed.



The message queue is a buffer that is used in non-shared memory environments, where tasks communicate by passing messages to each other rather than by accessing shared variables. Tasks share a common buffer pool. The message queue is an unbounded FIFO queue that is protected from concurrent access by different threads.

Events are asynchronous. When a class sends an event to another class, rather than sending it directly to the target reactive class, it passes the event to the operating system message queue. The target class retrieves the event from the head of the message queue when it is ready to process it. Synchronous events can be passed using triggered operations instead.

Many tasks can write messages into the queue, but only one can read messages from the queue at a time. The reader waits on the message queue until there is a message to process. Messages can be of any size.

Functions of Message Queue

There are four important functions that we will use in the programs to achieve IPC using message queues.

**1. int msgget (key\_t key, int msgflg);**

We use the msgget function to create and access a message queue. It takes two parameters.

* The first parameter is a key that names a message queue in the system.
* The second parameter is used to assign permission to the message queue and is ORed with IPC\_CREAT to create the queue if it doesn't already exist. If the queue already exists, then IPC\_CREAT is ignored. On success, the msgget function returns a positive number which is the queue identifier, while on failure, it returns -1.

**2. int msgsnd (int msqid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);**

This function allows us to add a message to the message queue.

* The first parameter (msgid) is the message queue identifier returned by the msgget function.
* The second parameter is the pointer to the message to be sent, which must start with a long int type.
* The third parameter is the size of the message. It must not include the long int message type.
* The fourth and final parameter controls what happens if the message queue is full or the system limit on queued messages is reached. The function on success returns 0 and place the copy of message data on the message queue. On failure, it returns -1.

There are two constraints related to the structure of the message. First, it must be smaller than the system limit and, second, it must start with a long int. This long int is used as a message type in the receive function. The best structure of the message is shown below.

|  |
| --- |
| struct my\_message  {  **long** **int** message\_type;  /\* The data you wish to transfer \*/  } |

Since the message\_type is used in message reception, you can't simply ignore it. You must declare your data structure to include it, and it's also wise to initialize it to contain a known value.

**3. int msgrcv (int msqid, void \*msg\_ptr, size\_t msg\_sz, long int msgtype, int msgflg);**

This function retrieves messages from a message queue.

* The first parameter (msgid) is the message queue identifier returned by the msgget function.
* As explained above, the second parameter is the pointer to the message to be received, which must start with a long int type.
* The third parameter is the size of the message.
* The fourth parameter allows implementing priority. If the value is 0, the first available message in the queue is retrieved. But if the value is greater than 0, then the first message with the same message type is retrieved. If the value is less than 0, then the first message having the type value same as the absolute value of msgtype is retrieved. In simple words, 0 value means to receive the messages in the order in which they were sent, and non zero means receive the message with a specific message type.
* The final parameter controls what happens if the message queue is full or the system limit on queued messages is reached. The function on success returns 0 and place the copy of message data on the message queue. On failure, it returns -1.

**4. int msgctl (int msqid, int command, struct msqid\_ds \*buf);**

The final function is msgctl, which is the control function.

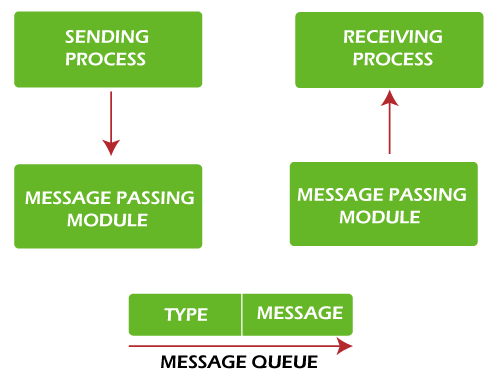
* The first parameter is the identifier returned by the msgget function.
* The second parameter can have one out of the below three values.

|  |  |
| --- | --- |
| **Command** | **Description** |
| IPC\_STAT | It sets the data in the msqid\_ds structure to reflect the values associated with the message queue. |
| IPC\_SET | If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid\_ds data structure. |
| IPC\_RMID | It deletes the message queue. |

The msgctl function returns 0 on success and -1 on error. The send or receive function will fail if a message queue is deleted while a process is waiting in an msgsnd or msgrcv function.

Steps to Perform IPC using Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. Below are the following steps to perform communication using message queues.



1. A new queue is created or an existing queue opened by **msgget()**.
2. New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all specified to msgsnd() when the message is added to a queue.
3. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field. All processes can exchange information through access to a common system message queue. The sending process places a message onto a queue that another process can read. Each message is given an identification or type so that processes can select the appropriate message. The process must share a common key to gain access to the queue in the first place.
4. Perform control operations on the message queue **msgctl()**.

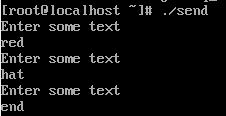
Examples

**Program 1:** Let's write a program for IPC using Message Queues to send data to a message queue.

|  |
| --- |
| #include<stdlib.h>   #include<stdio.h>   #include<string.h>   #include<unistd.h>   #include<sys/types.h>   #include<sys/ipc.h>   #include<sys/msg.h>   #define MAX\_TEXT 512   //maximum length of the message that can be sent allowed   struct my\_msg{  **long** **int** msg\_type;  **char** some\_text[MAX\_TEXT];   };  **int** main()   {  **int** running=1;  **int** msgid;           struct my\_msg some\_data;  **char** buffer[50]; //array to store user input           msgid=msgget((key\_t)14534,0666|IPC\_CREAT);  **if** (msgid == -1) // -1 means the message queue is not created           {                   printf("Error in creating queue\n");                   exit(0);           }    **while**(running)           {                   printf("Enter some text:\n");                   fgets(buffer,50,stdin);                   some\_data.msg\_type=1;                   strcpy(some\_data.some\_text,buffer);  **if**(msgsnd(msgid,(**void** \*)&some\_data, MAX\_TEXT,0)==-1) // msgsnd returns -1 if the message is not sent                   {                           printf("Msg not sent\n");                   }  **if**(strncmp(buffer,"end",3)==0)                   {                           running=0;                   }           }   } |

**Output**

The above program gives this output.



**How does it Work?**

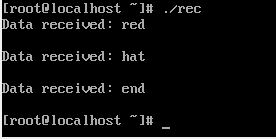
The structure my\_msg declares the long int variable and the char array to store the data sent to the message queue. Then the message queue is created using the msgget() function. Next, read data from the user into the buffer using fgets() and then copy it into the variable some\_text of the structure some\_data. Finally, send the data to the queue using the msgsnd() function. The strcmp function is used to stop sending the data by comparing the first three characters of the data. If the data starts with "end", it means no more data is to be sent.

**Program 2:** Now, let's write a program for IPC using Message Queues to receive or read message from the above created message queue.

|  |
| --- |
| #include<stdlib.h>   #include<stdio.h>   #include<string.h>   #include<unistd.h>   #include<sys/types.h>   #include<sys/ipc.h>   #include<sys/msg.h>   struct my\_msg{  **long** **int** msg\_type;  **char** some\_text[BUFSIZ];   };  **int** main()   {  **int** running=1;  **int** msgid;           struct my\_msg some\_data;  **long** **int** msg\_to\_rec=0;           msgid=msgget((key\_t)12345,0666|IPC\_CREAT);  **while**(running)           {                   msgrcv(msgid,(**void** \*)&some\_data,BUFSIZ,msg\_to\_rec,0);                   printf("Data received: %s\n",some\_data.some\_text);  **if**(strncmp(some\_data.some\_text,"end",3)==0)                   {                           running=0;                   }           }            msgctl(msgid,IPC\_RMID,0);   } |

**Output**

The above code gives the following output.



**How does it work?**

The msg\_to\_rec variable is set to 0 so that the data is received in the same order as sent. The while is used to continuously receive the data using the mgrcv() function until the text received is "end", which we check using the strcmp function. The data is read using the structure my\_msg.