**Name:Rohan Nivas Chavan**

**PRN:2019BTEIT00084**

**UOS LAB**

**7. IPC: Message Queue**

**7.2** Write 2 programs that will both send and messages and constructthe following dialog between them

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

**Theory:**

Two (or more) processes can exchange information via access to a common system message queue. The sending process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure 24.1). Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized

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. 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 transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

1.The call succeeds.

2.The process receives a signal.

3.The queue is removed.

**1. Initialising the Message Queue**

The msgget() function initializes a new message queue:

int msgget(key\_t key, int msgflg)

It can also return the message queue ID (msqid) of the queue corresponding to the key argument. The value passed as the msgflg argument must be an octal integer with settings for the queue's permissions and control flags.

**2. 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 )

**3. 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);

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 structure member msgtype is the received message's type as specified by the sending process.

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, errno is 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:

1.msg\_qnum is incremented by 1.

2.msg\_lspid is set equal to the process ID of the calling process.

3.msg\_stime is set equal to the current time.

**Data Dictionary:**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **TYPE** | **USE** |
| **Sr.** | **NAME OF VARIABLE** |  |  |
| **No.** | **/FUNCTION/STRUCT** |  |  |
|  |  |  |  |
| **1** | Msqid | int | FOR SOCKET TUPLE |
|  |  |  |  |
| **2** | Msgflg | int | FOR SEMAPHORE |
|  |  |  |  |
| 3 | Key | Key\_t | Semaphore id |
|  |  |  |  |
| 4 | Sbuf | Struct |  |
|  |  | msgbuf |  |
|  |  |  |  |
| 5 | buf\_length | size\_t |  |
|  |  |  |  |
|  |  |  |  |

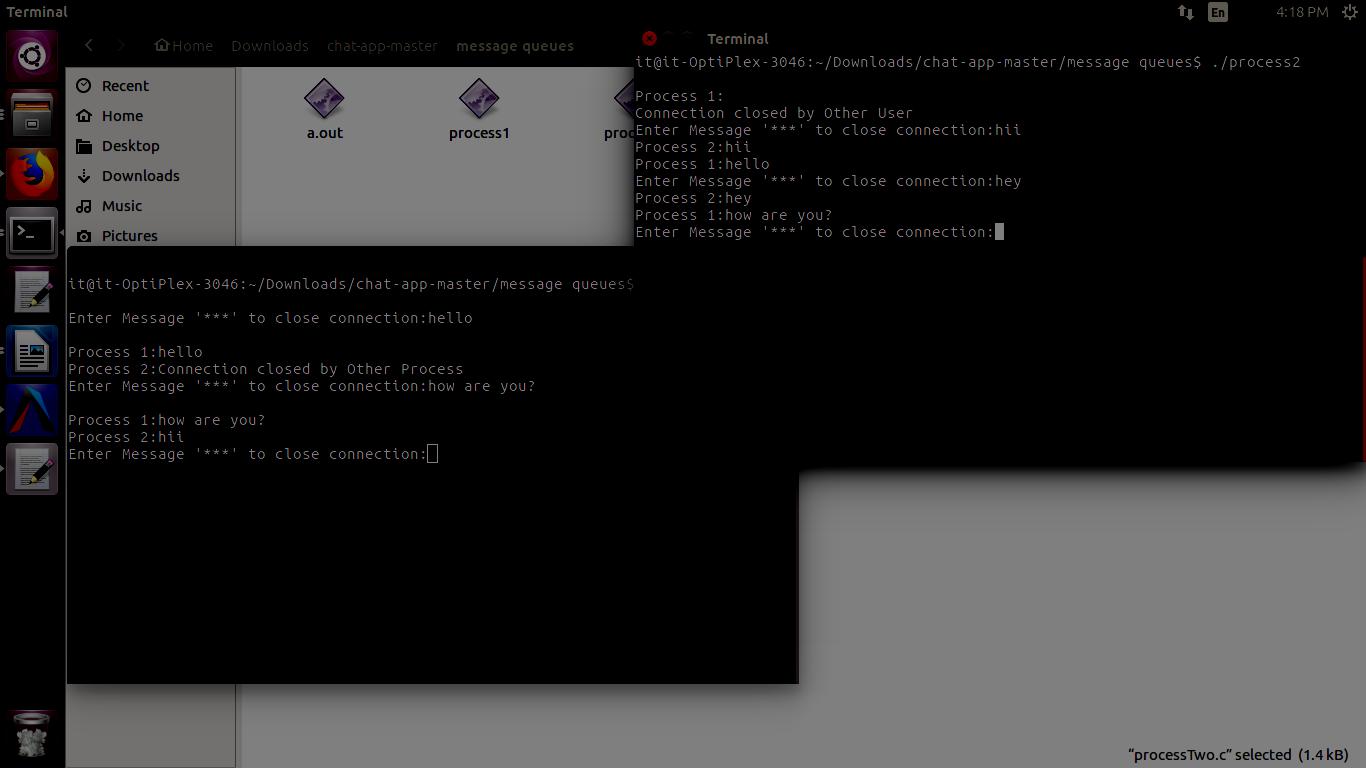
Table 7.2.1 Data Dictonary

**Data/Variables dictionary for Receiver:**

|  |  |  |  |
| --- | --- | --- | --- |
| SN. | **NAME OF VARIABLE/ FUNCTION/** | **TYPE** | **USE** |
|  | **STRUCT** |  |  |
|  |  |  |  |
| 1 | msqid | int | FOR SOCKET TUPLE |
|  |  |  |  |
| 3 | Key | key\_t | Semaphore id |
|  |  |  |  |
| **4** | rbuf | Struct | To store the message |
|  |  | msgbuf |  |
|  |  |  |  |
| 5 | buf\_length | size\_t | Length of message to be |
|  |  |  | sent |
|  |  |  |  |
|  |  |  |  |

Table 7.2.2 Data Dictonary

**Output:**



**Conclusion:**

1.Use of message queue functions like msgget, msgsend

2. Msgrecv to implement message passing mechanism between server

3. Client studied and implemented it to intoduce concept of chatting.

**References:**

[1] Dave’s Programming in C Tutorials.