### IPC: Message Queue

Subject:- Unix Operating System System Lab Class :- TYIT

**Name: - SHRENIK JADHAV**

**OM GHARGE**

**PRN: -2020BTEIT00011,2020BTEIT00041**

**Assignment No 7b**

**Title-**Write a 2 programs that will both send and messages and construct the following dialog between them.

**Objective:**

1. To learn about IPC through message queue.
2. Use of system call and IPC mechanism to write effective application programs

**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

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.

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.

1. 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.

* + 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.

**Data Dictionary:**

|  |  |  |  |
| --- | --- | --- | --- |
| **SR.NO** | **Variable/function** | **DataType** | **Use** |
| 1. | msqid | int | For Socket Tuple |
| 2. | msgflg | int | For Semaphore |
| 3. | key | key\_t | Semaphore id |
| 4. | sbuf | struct msgbuf |  |

**Program-**

**Sender.c**

**#include <stdio.h>**

**#include <stdlib.h>**

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

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

**#include <sys/types.h>**

**struct message {**

**long mtype;**

**char mtext[100];**

**};**

**int main() {**

**key\_t key;**

**int msgid;**

**struct message msg;**

**// Generate a unique keykey = ftok("message\_queue", 65);**

**// Create a message queue**

**msgid = msgget(key, 0666 | IPC\_CREAT);**

**// Send the message**

**msg.mtype = 1;**

**sprintf(msg.mtext, "Are you hearing me?");**

**msgsnd(msgid, &msg, sizeof(msg), 0);**

**// Wait for the reply**

**msgrcv(msgid, &msg, sizeof(msg), 2, 0);**

**printf("Process 2: %s\n", msg.mtext);**

**// Send the response**

**msg.mtype = 1;**

**sprintf(msg.mtext, "I can hear you too");**

**msgsnd(msgid, &msg, sizeof(msg), 0);**

**// Remove the message queue**

**msgctl(msgid, IPC\_RMID, NULL);**

**return 0;**

**}**

**Receiver.c**

**#include <stdio.h>**

**#include <stdlib.h>**

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

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

**#include <sys/types.h>**

**struct message {**

**long mtype;**

**char mtext[100];**

**};**

**int main() {**

**key\_t key;**

**int msgid;**

**struct message msg;**

**// Generate a unique keykey = ftok("message\_queue", 65);**

**// Access the message queue**

**msgid = msgget(key, 0666 | IPC\_CREAT);**

**// Receive the message**

**msgrcv(msgid, &msg, sizeof(msg), 1, 0);**

**printf("Process 1: %s\n", msg.mtext);**

**// Send the reply**

**msg.mtype = 2;**

**sprintf(msg.mtext, "Loud and Clear");**

**msgsnd(msgid, &msg, sizeof(msg), 0);**

**// Receive the response**

**msgrcv(msgid, &msg, sizeof(msg), 1, 0);**

**printf("Process 1: %s\n", msg.mtext);**

**return 0;**

**}**

**RUN-**

**Create two files named Sender.c and Receiver.c.**

* Copy and paste the respective code into each file.
* Open a terminal.
* Navigate to the directory where the C files are located using the cd command. For example, if the files are in the home directory, you can use:

bash

* cd ~
* Compile the Sender.c file using the following command:
* gcc Sender.c -o sender

This command will compile the C file and generate an executable binary named sender.

* Compile the Receiver.c file using the following command:
* gcc Receiver.c -o receiver

This command will compile the C file and generate an executable binary named receiver.

* Once the compilation is successful without any errors, you can run the programs.
* Open two separate terminals.
* In the first terminal, execute the receiver binary using the following command:

bash

* ./receiver

This command will start the receiver program, which will wait for a message.

* In the second terminal, execute the sender binary using the following command:

bash

* + ./sender

This command will start the sender program, which will send a message to the receiver.

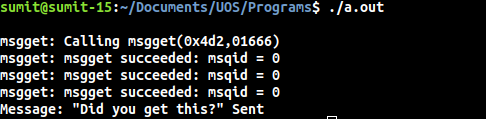
1. You should see the output in the respective terminals, indicating the communication between the sender and receiver processes.

Note: Make sure you have the GCC compiler installed on your Ubuntu system. If not, you can install it by running the following command:

arduino

sudo apt-get install build-essential

**Output-**



**Conclusion-**

Use of message queue functions like msgget, msgsend, and msgrecv to implement message passing mechanism between server and client studied and implemented it to intoduce concept of chatting.

**Reference-**

Dave’s Programming in C Tutorials