CPE 435: OPERATING SYSTEMS LABORATORY.

Lab05

Message Queue.

Submitted by: Dan Otieno.

Date of Experiment: 02/10/23.

Report Deadline: 02/17/23.

Demonstration Deadline: 02/17/23.

Introduction:

This lab is similar to Lab 4 where we use different process that are not a parent-child relationship to create a message queue process. The goal is to apply various inter-process communication concepts in the LINUX environment in creating the message queue.

Theory:

Message queues are an example of inter-process communication in LINUX. All messages are stored in the kernel and have a queue identifier, or numeric key called msqid . The msqid identifies particular message queues, and then processes read and write messages to arbitrary message queues. Every message queue contains a long int datatype, the data length, and the data itself if its length is greater than 0. For this lab, we apply these concepts to create a chat server in the Linux environment. Some of the theory topics explored in this lab are below:

- ## int msgget():
    - This is a function that returns the message queue identifier associated with the value of the key. The function takes two parameters, key_t key and int msgflg. may be used either to obtain the identifier of a previously created message queue (when msgflg is 0), or to create a new set. Msgget return value is the message queue identifier or an error if it fails. According to an article from the linux manual website cited below:
        - [A new message queue is created if key has the value IPC_PRIVATE or key isn't IPC_PRIVATE, no message queue with the given key *key* exists, and IPC_CREAT is specified in msgflg.
        - 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. (This analogous to the effect of the combination O_CREAT | O_EXCL for open(2).)
        - Upon creation, the least significant bits of the argument msgflg define the permissions of the message queue. These permission bits have the same format and semantics as the permissions specified for the mode argument of open(2). (The execute permissions are not used.)] - Source

- ## int msgsnd() and int msgrcv():
    - The msgsnd function is called to send messages and the msgrcv function call is used to receive messages from a message queue. The process to call the functions must include write and read permissions in the queue, to send and receive messages respectively. According to the LINUX manual page,
        - [The msgsnd() system call appends a copy of the message pointed to by msgp to the message queue whose identifier is specified by msqid.
        - If sufficient space is available in the queue, msgsnd() succeeds immediately. The queue capacity is governed by the msg_qbytes field in the associated data structure for the message queue. During queue creation this field is initialized to MSGMNB bytes, but this limit can be modified using msgctl(2).

A message queue is considered to be full if either of the following conditions is true:

- Adding a new message to the queue would cause the total number of bytes in the queue to exceed the queue's maximum size (the msg_qbytes field).
- Adding another message to the queue would cause the total number of messages in the queue to exceed the queue's maximum size (the msg_qbytes field). This check is necessary to prevent an unlimited number of zero-length messages being placed on the queue. Although such messages contain no data, they nevertheless consume (locked) kernel memory.

- If insufficient space is available in the queue, then the default behavior of msgsnd() is to block until space becomes available.
- The msgrcv() system call removes a message from the queue specified by msqid and places it in the buffer pointed to by msgp.
- The argument msgsz specifies the maximum size in bytes for the member mtext of the structure pointed to by the msgp argument. If the message text has length greater than msgsz, then the behavior depends on whether MSG_NOERROR is specified in msgflg. If MSG_NOERROR is specified, then the message text will be truncated (and the truncated part will be lost); if MSG_NOERROR is not specified, then the message isn't removed from the queue and the system call fails returning -1 with errno set to E2BIG.] - Source

- int msgctl():
  - The msgctl() function carries out the operations specified in the message queue, using the msqid identifier. According to LINUX manual, [On success, IPC_STAT, IPC_SET, and IPC_RMID return 0. A successful IPC_INFO or MSG_INFO operation returns the index of the highest used entry in the kernel's internal array recording information about all message queues. (This information can be used with repeated MSG_STAT or MSG_STAT_ANY operations to obtain information about all queues on the system.) A successful MSG_STAT or MSG_STAT_ANY operation returns the identifier of the queue whose index was given in msqid.
  - On failure, -1 is returned and errno is set to indicate the error.] - Source

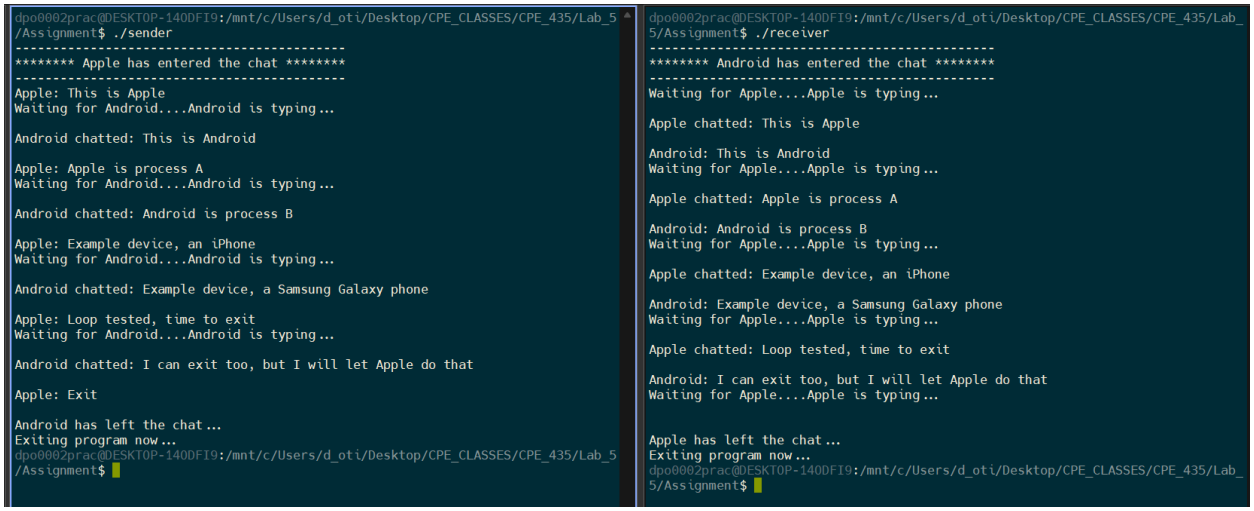
Results & Observation:

Assignment 1:

Description:

The goal for this assignment was to use C/C++ to write a chat server program that demonstrates the interaction between two separate processes that are not part of a parent-child relationship. Process B sends data to Process A, which waits for receipt of that data and then sends a message to Process B. This is achieved through user inputs on both command lines. User should type exit whether on the terminal of one of the processes to exit the entire program. In my example, I just named them "Apple" and "Android" instead of Process A and Process B.

Program Output:



Lab Research Questions:

1. How do you make a process wait to receive a message and not return immediately?
    a. We can set the IPC_NOWAIT flag in the msgrcv system call, where a block will be enabled until messages arrive in the queue in a way that adheres to the system call parameters.
2. Message Queue vs Shared Memory (discuss use and differences).
    a. Message Queue: Provide a means to send blocks of data from a process to another process, where each block has a type, and the receiver gets the blocks with different values independently. They provide an easy way of exchanging data between two unrelated processes. They operate like named pipers, without necessarily opening and closing those pipes.
    b. Shared Memory: This IPC method provides a means to share data across multiple processes but does not have any synchronization facility. Two unrelated processes can

access the same logical memory. Shared memory, therefore, is really a range of addresses created by an IPC for a process, and in the address space of that particular process. When a process writes to the shared memory, changes can be seen by any other processes that have access to that same memory.

   c. Source for Shared Memory and Message Queue explanation: https://coggle.it/diagram/XXbAqkd5XQ-bu1qd/t/semaphores%2C-shared-memory%2C-and-message-queues

3. Research use of function ftok(), what is its use?
   a. This is a function that uses the identity of a file named by a given pathname, where the resulting value is the same for all pathnames that name that file. When it runs successfully ftok returns the key_t value and a -1 if it fails. Source - https://man7.org/linux/man-pages/man3/ftok.3.html
4. What does IPC_NOWAIT do?
   a. Ipc_nowait is an argument, or one of the parameters for IPC message queue processes. If IPC_NOWAIT is passed as a flag, and no messages are available, the call returns ENOMSG to the calling process. Otherwise, the calling process blocks until a message arrives in the queue that satisfies the msgrcv() parameters. If the queue is deleted while a client is waiting on a message, EIDRM is returned. EINTR is returned if a signal is caught while the process is in the middle of blocking, and waiting for a message to arrive. Source - https://tldp.org/LDP/lpg/node35.html#SECTION00742400000000000000

Conclusion:

All programs in this lab functioned as expected and helped understand other concepts of IPC processes within the LINUX environment, which built upon the concepts learned in the previous lab.

Appendix:

Assignment 1 code – Process A.

```
/*
DAN OTIENO
CPE 434-01
LAB 5
Exercise 1
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#define key ((key_t)(1234))
#define charMAX 100

struct text_message {
        long mymsg_type;
        char mymsg_text[charMAX];
};

int main()
{
        struct text_message message;
        int msid = msgget(key, 0666 | IPC_CREAT);

        printf("----------------------------------------\n");
        printf("******** Apple has entered the chat ********\n");
        printf("----------------------------------------\n");

        while(1)
        {
                printf("Apple: ");
                fgets(message.mymsg_text, charMAX, stdin);
                message.mymsg_type = 1;
                msgsnd(msid, &message, sizeof(message), 0);

                if(strcmp(message.mymsg_text, "Exit\n") == 0)
                {
                        printf("\nAndroid has left the chat...\n");
                        printf("Exiting program now...\n");
                        msgctl(msid, IPC_RMID, 0);
                        exit(0);
                }
                printf("Waiting for Android....Android is typing...\n\n");
```

```
                msgrcv(msid, &message, sizeof(message), 2, 0);

                if(strcmp(message.mymsg_text, "Exit\n") == 0)
                {
                        printf("\nAndroid has left the chat...\n");
                        printf("Exiting program now...\n");
                        msgctl(msid, IPC_RMID, 0);
                        exit(0);
                }
                printf("Android chatted: %s\n", message.mymsg_text);
        }
        msgctl(msid, IPC_RMID, 0);

        return 0;
}
```

Assignment 1 code – Process B.

```
/*
DAN OTIENO
CPE 434-01
LAB 5
Exercise 1
*/
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#define key ((key_t)(1234))
#define charMAX 100
```

```
struct text_message {
        long mymsg_type;
        char mymsg_text[charMAX];
};
int main()
{
        struct text_message message;
        int msid = msgget(key, 0666 | IPC_CREAT);
        printf("-----------------------------------------\n");
        printf("******** Android has entered the chat ********\n");
        printf("-----------------------------------------\n");
        while(1)
        {
                printf("Waiting for Apple....Apple is typing...\n\n");
                msgrcv(msid, &message, sizeof(message), 1, 0);

                if(strcmp(message.mymsg_text, "Exit\n") == 0)
                {
                        printf("\nApple has left the chat...\n");
                        printf("Exiting program now...\n");
                        msgctl(msid, IPC_RMID, 0);
                        exit(0);
                }
                printf("Apple chatted: %s\n", message.mymsg_text);

                printf("Android: ");
                fgets(message.mymsg_text, charMAX, stdin);
```

```
                message.mymsg_type = 2;
                msgsnd(msid, &message, sizeof(message), 0);
                if(strcmp(message.mymsg_text, "Exit\n") == 0)
                {
                        printf("\nApple has left the chat...\n");
                        printf("Exiting program now...\n");
                        msgctl(msid, IPC_RMID, 0);
                        exit(0);
                }
        }
        msgctl(msid, IPC_RMID, 0);

        return 0;
}
```