Andrew Ma

Lab 5

CPE 435

2/17/21

Theory

This lab is for exploring message queues. Message queues are one form of IPC (inter-process communication), and messages can be passed between processes using message queues. All the messages are stored in the kernel, and each message has an associated message queue id (msqid). The msg queue structure (msqid_ds) is in <sys/msg.h>, and the user defines the msg structure. Each msg has a long messageType and a data portion, which can be any size. The msgqid_ds structure has the PID of last sent and received messages, the max and current number of bytes allowed on the queue, the time of last sent, received, and changed messages, and a struct ipc_perm describing the operation permissions.

The int msgget(key_t key, int msgflg) function takes in a key to associate with a message queue, and it can either create a new message queue or get an existing message queue. The return value is the newly created or existing message queue identifier (msqid) that can be used in msgsnd, msgrcv, and msgctl. If the msgflg is set to IPC_CREAT and no message queue with the same key exists, then a new message queue is created. The permissions for the message queue can be set by ORing the permissions number with IPC_CREAT.

The int msgsnd(int msqid, void const* msgp, size_t msgsz, int msgflg) function is for sending messages onto the message queue specified by msqid. The msgp pointer points to a user defined message structure that has a long mtype (specifies type of message and must be a positive number), and then a data portion that holds the data bytes of the message. The msgsz is the size of the data portion not the size of the struct.

The msgflg sets the action to take if the number of bytes on the queue has reached the maximum number of bytes on the queue, or we have reached the system's maximum number of messages. If the IPC_NOWAIT is set, the message is not sent and the calling thread returns immediately. If the IPC_NOWAIT is not set, then the calling thread suspends execution until 1) the flag is no longer 0 which will then send the message, 2) the message queue identifier is removed which will cause an error and return -1, or 3) the calling thread receives a signal causing the message to not be sent and resuming execution in sigaction.

If the message is successfully sent, the message queue number adds 1, the message queue last PID is set to the caller's PID, and the message queue last set time is set to the current time, and it returns 0. If it fails, it returns -1.

The msgrcv(int msqid, void* msgp, size_t msgsz, long msgtyp, int msgflg) function is for receiving messages. It reads a message from the specified msqid queue, and places the message into a buffer msgp that the user has created, and this buffer can be cast to the message struct type to access the member variables by name. The msgsz is the size in bytes of the message data (not the size of the message struct). The long msgtyp is the type of message, and if the type number is 0, we get the first message on the queue. If the type number is greater than 0, then we get the first message on the queue with that type number.

The msgflg sets the action to take if the message type is not on the queue. If IPC_NOWAIT is set, the calling thread returns -1 immediately and sets errno=ENOMSG. If IPC_NOWAIT is not set, then the calling thread will be suspended until 1) a message of the desired type is placed on the queue (waits for message of that type), 2) the message queue is removed and errno=EIDRM and -1 is returned, or 3) the calling thread receives a signal and the message will not be received and the calling thread will resume according to sigaction.

If the message is successfully received, the message queue number subtracts 1, the last read PID is set to the caller's PID, and the last read time is set to the current time. If it is successful, it will return the number of bytes actually placed into the buffer *mtext*. If it fails, it will return -1;

The msgctl(int msqid, int cmd, struct msqid_ds* buf) function is for performing a control operation on a message queue specified by msqid. If the struct msqid_ds* buf is not NULL, then it points to a msqid_ds structure with the properties of the message queue. Some cmd operations are IPC_STATE (read the message queue's msqid_ds values into the local pointer buf), IPC_SET (updates the message queue's member variables with the ones in the local pointer buf), and IPC_RMID (immediately removes the message queue and awakens all waiting reader and writer processes with an error return and errno=EIDRM).

Observations

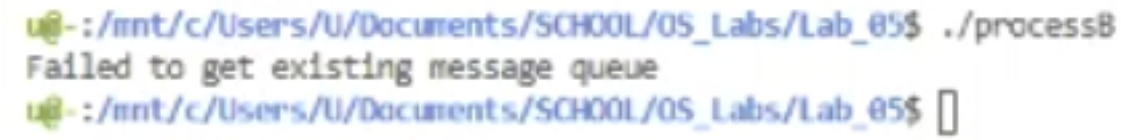
Process A sends message to process B and waits

Process B sends message to process A and waits

Process A sends “Exit”, causing both to exit

Process B sends “Exit”, causing both to exit

If process B runs before process A creates a message queue



Lab Questions

a. How do you make a process wait to receive a message and not return immediately?
   In the msgrcv() function, 0 can be passed in as the flag to wait to receive a message. To return immediately, IPC_NOWAIT is passed in as the flag.

b. Message Queue vs Shared Memory (use and differences)
   Message Queues and Shared Memory are both used for IPC to share information between processes. Shared Memory can be accessed with random access, so it is very fast. Message Queues can be accessed with linear access since it is a linked list of messages. Both of them must be created with a {shm,msg}get() function and both can be destroyed with {shm,msg}ctl() with the IPC_RMID flag. Shared Memory can use a local pointer that points to the shared memory address, and use the local pointer normally to read and write data. But, it must detach the local pointer to the shared memory address when the process ends. Message Queues must use msgsnd() and msgrcv() with a msg buffer to send and receive data. There is no attached pointer in message queues, so there is no detach function for message queues. Another main difference is the synchronization. Shared memory has no synchronization provided, so there can be a flag in the shared structure that different processes can use to take or receive control; but, all the logic to handle the flag variable will have to be defined by each process. In Message Queues, synchronization is provided, and the IPC_NOWAIT flag can be set for both sending or receiving to not wait, or the flag can be 0 to wait until a message is able to be sent or received.

c. Use of function ftok(), what is its use?
   The ftok() function converts a pathname and a project identifier to an IPC key. It takes in a filename, and a project id integer to generate a key_t type IPC key that can be used with msgget(), semget(), or shmget(). It will generate the same key if the same filename and same project id integer is

used. In my project I set the key_t KEY=12345, but ftok() can be used to generate a key automatically.

d. What does IPC_NOWAIT do?
In the msgsnd() function, the IPC_NOWAIT flag will return immediately if the number of bytes on the queue has reached the max number of bytes on the queue, or if we have reached the maximum number of messages. In the msgrcv() function, the IPC_NOWAIT flag will return immediately if the requested message type (if message type is 0, then it means all message types) is not on the queue.

Conclusion

The program worked as expected, and the processes could communicate back and forth sending and receiving messages on the message queue. I learned how to use message queues, and what the benefits of message queues are over previous IPC methods like Shared Memory and Pipes. I learned how message queues have synchronization provided so there doesn't need to be a flag variable, but the IPC_NOWAIT flag can be set if a process wants to return immediately, or the flag can be 0 to wait until a message is able to be sent or received. This makes the code easier to write, but the cost is slower access compared to Shared Memory. But, since each message is separate from other messages, it can provide safety for values compared to shared values in Shared Memory.

Source Code

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

```c
#include <string.h>
#include <signal.h>

#include "msg.h"

void INThandler(int);
int msid;

void closeMessageQueue(int msqid) {
    // close message queue
    if (msgctl(msid, IPC_RMID, NULL) == -1) {
        // error closing message queue
        fprintf(stderr, "Failed to close message queue\n");
    }
    else {
        printf("Closed Message Queue Successfully\n");
        fflush(stdout);
    }
}

int main(void) {
    signal(SIGINT, INThandler);

    // processA creates a new message queue with permissions rw-rw-rw
    msid = msgget(MSG_QUEUE_KEY, IPC_CREAT | 0666);
    if (msid == -1) {
        // error creating new message queue
        fprintf(stderr, "Failed to create new message queue\n");
        exit(1);
    }

    // we don't have to worry about synchronization since message queues
handle that in the msgsnd and msgrcv functions

    // set processA's to send type 1, and receive type 2
```

```
    long sendMsgType = 1;
    long receiveMsgType = 2;

    struct msg sendMsg;
    struct msg receiveMsg;
    // initialize defaults (will be using sendMsg and receiveMsg buffers
so mtype will persist until changed manually)
    sendMsg.mtype = sendMsgType;
    receiveMsg.mtype = receiveMsgType;

    while (1) {
        printf("A > ");
        fflush(stdout);
        // get user input and save to sendMsg
        fgets(sendMsg.mtext, MSG_DATA_LENGTH, stdin);
        // remove \n character than fgets keeps
        sendMsg.mtext[strlen(sendMsg.mtext) - 1] = '\0';

        // send message to Queue
        if (msgsnd(msid, &sendMsg, MSG_DATA_LENGTH, IPC_NOWAIT) == -1) {
            // error
            fprintf(stderr, "Failed to send message: type %ld, text
'%s'\n", sendMsg.mtype, sendMsg.mtext);
            closeMessageQueue(msid);
            exit(1);
        }
        else {
            // sent successful
            // If processA sent "Exit" to processB, then break
            if (strncmp(sendMsg.mtext, "Exit", strlen(sendMsg.mtext)) ==
0) {
                break;
            }
        }

        printf("Waiting ...\n");
```

```
        fflush(stdout);
        // flag is 0, so it will wait until it receives a message
        if (msgrcv(msid, &receiveMsg, MSG_DATA_LENGTH, receiveMsgType,
0) == -1) {
            // error
            fprintf(stderr, "Failed to receive message: type %ld\n",
receiveMsg.mtype);
            closeMessageQueue(msid);
            exit(1);
        }
        else {
            // receive successful

            // if B sent "Exit" to A, exit A
            if (strncmp(receiveMsg.mtext, "Exit",
strlen(receiveMsg.mtext)) == 0) {
                break;
            }

            printf("B: '%s'\n", receiveMsg.mtext);
            fflush(stdout);
        }
    }


    closeMessageQueue(msid);
    exit(0);
}


void INThandler(int sig) {
    closeMessageQueue(msid);
    exit(0);
}
```

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

#include <string.h>
#include <signal.h>

#include "msg.h"

void INThandler(int);
int msid;

int main(void) {
    signal(SIGINT, INThandler);

    // processB looks for existing message queue
    msid = msgget(MSG_QUEUE_KEY, 0);
    if (msid == -1) {
        // error getting message queue
        fprintf(stderr, "Failed to get existing message queue\n");
        exit(1);
    }

    // we don't have to worry about synchronization since message queues handle that in the msgsnd and msgrcv functions

    // set processB's to send type 2, and receive type 1
    long sendMsgType = 2;
    long receiveMsgType = 1;

    struct msg sendMsg;
    struct msg receiveMsg;
```

```
    // initialize defaults (will be using sendMsg and receiveMsg buffers
so mtype will persist until changed manually)
    sendMsg.mtype = sendMsgType;
    receiveMsg.mtype = receiveMsgType;


    while (1) {
        printf("Waiting ...\n");
        fflush(stdout);
        // flag is 0, so it will wait until it receives a message
        if (msgrcv(msid, &receiveMsg, MSG_DATA_LENGTH, receiveMsgType,
0) == -1) {
            // error
            fprintf(stderr, "Failed to receive message: type %ld\n",
receiveMsg.mtype);
            exit(1);
        }
        else {
            // read successful

            // if received message text is "Exit", then exit processB
            if (strncmp(receiveMsg.mtext, "Exit",
strlen(receiveMsg.mtext)) == 0) {
                break;
            }

            printf("A: '%s'\n", receiveMsg.mtext);
            fflush(stdout);
        }

        printf("B > ");
        fflush(stdout);
        // get user input and save to sendMsg
        fgets(sendMsg.mtext, MSG_DATA_LENGTH, stdin);
        // remove \n character than fgets keeps
        (sendMsg.mtext)[strlen(sendMsg.mtext) - 1] = '\0';
```

```
        // send message to Queue
        if (msgsnd(msid, &sendMsg, MSG_DATA_LENGTH, IPC_NOWAIT) == -1) {
            // error
            fprintf(stderr, "Failed to send message: type %ld, text
'%s'\n", sendMsg.mtype, sendMsg.mtext);
            exit(1);
        }
        else {
            // send successful

            // if processB sent "Exit" to processA, exit processB
            if (strncmp(sendMsg.mtext, "Exit", strlen(sendMsg.mtext)) ==
0) {
                break;
            }
        }
    }

    exit(0);
}


void INThandler(int sig) {
    exit(0);
}
```

```
//msg.h
#ifndef MSG_H
#define MSG_H

#include <sys/types.h>

#define MSG_DATA_LENGTH 100
```

```
struct msg
{
    __syscall_slong_t mtype;    /* type of received/sent message */
    char mtext[MSG_DATA_LENGTH];        /* text of the message */
};

key_t MSG_QUEUE_KEY = 12345;

#endif
```