Austin Brown

CPE 434-01

2/7/2021

Lab 5

Theory

The purpose of this lab is to implement a message queue to pass information from one process to another. They operate in a first in first out format. To do this, we use the msgget, msgsnd, msgctl, and msgrcv functions.

The msgget function is what creates a message queue. It returns a message identifier. Just like in the last lab, we pass in a key to attach the process to the queue.

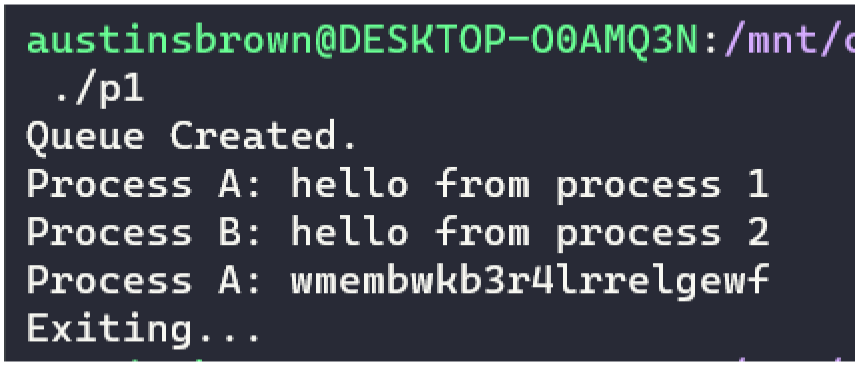
The msgrcv function receives a message from the queue. It removes it from the queue in the process. If it fails, it will return -1. By default, msgrcv will wait to receive something from the queue. In the demo code, this behavior is changed by setting the flag to *IPC_NOWAIT.* This will cause msgrcv to send an error code if nothing is received.

The msgsnd function sends a message to the queue. The first argument passed in is the queue id that was returned by msgget. The second argument is a pointer to the structure that holds the message and message type. The third argument is the length of the message.

The msgctl function performs a command of the queue. This function is used to delete the message queue that was created.

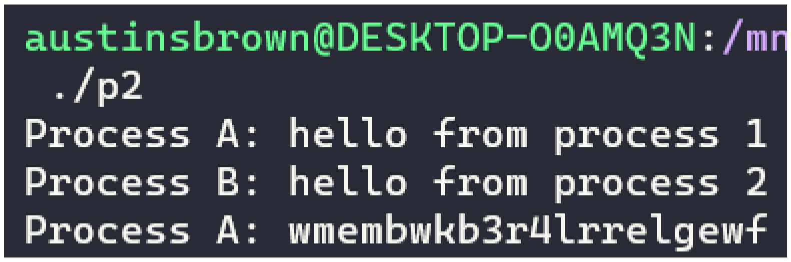
Outputs

Output from Process 1



```
austinsbrown@DESKTOP-O0AMQ3N:/mnt/
 ./p1
Queue Created.
Process A: hello from process 1
Process B: hello from process 2
Process A: wmembwkb3r4lrrelgewf
Exiting...
```

Output from Process 2



```
austinsbrown@DESKTOP-O0AMQ3N:/m
 ./p2
Process A: hello from process 1
Process B: hello from process 2
Process A: wmembwkb3r4lrrelgewf
```

Questions

1. When you use the fgets function, the process will wait for the return key before it continues onward. It will not continue without the user giving input first.
2. A shared memory space is just a block of memory. It is up to the programmer to manage how this space is managed. You must maintain flags to make sure that the processes do not interfere with each other. A queue is a first in first out memory structure. It is much easier to maintain since the memory structure is defined. Both are memory spaces outside of the process.
3. The ftok function returns a key. It does this based on a path and an id that are passed in. So long as the path and id are the same, it will create the same key every time. It is useful for creating keys for shmget and msgget.
4. IPC_NOWAIT is passed into msgsend as the flag. It makes it so that if the queue is full then the function will fail instead of waiting.

Conclusion

A message queue is an easy way to share information between one process and another. With a queue, you do not have to worry about your data being overwritten because of the queue structure. It is slower to access then a shared memory region, however.

Code

```
#define MTEXTSIZE 100
#define QUEUEKEY (key_t)1234

struct TextMessage
{
    long mtype; // type of message being sent
    char mtext[QUEUEKEY]; // the message to be sent
};
```

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include "header.h"

int main()
{
    int queueID; // id of a message queue
    int sendStatus; // return variable for msgsnd
    struct TextMessage message;

    if((queueID = msgget(QUEUEKEY, IPC_CREAT | 0666)) == -
1) // create message queue
    {
        printf("An error has occured!\n"); // error handling
        exit(-1);
    }

    printf("Queue Created.\n");

    for(;;)
    {
        printf("Process A: ");
        fgets(message.mtext, MTEXTSIZE, stdin); // get input from the
 user
        message.mtext[strcspn(message.mtext, "\n")] = 0; // get rid o
f newline character
        message.mtype = 1; // set message type

        sendStatus = msgsnd(queueID, &message, strlen(message.mtext)+
1, 0); // send a message
        if(sendStatus < 0)
        {
            printf("Could not send Message!\n");
            exit(-1);
        }
        if(msgrcv(queueID, &message, MTEXTSIZE, 2, 0) < 0) // try to
recieve message
        {
            printf("An error has occured. Could not recieve message\n
.");
            exit(-1);
        }

        else
        {
            if(strcasecmp(message.mtext, "EXIT") == 0)
            {
                printf("Exiting...\n");
                break;
            }

            printf("Process B: %s\n", message.mtext);
        }
    }

    msgctl(queueID, IPC_RMID, NULL); // delete the message queue
    return 0;
}
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include "header.h"

int main()
{
    int queueID;
    int recieveSataus;
    struct TextMessage message;

    queueID = msgget(QUEUEKEY, 0);
    if (queueID == -1)
    {
        printf("Could not attach to message queue!\n");
        exit(-1);
    }

    while(!(strcasecmp(message.mtext, "EXIT") == 0))
    {
        recieveSataus = msgrcv(queueID, &message, MTEXTSIZE, 1, 0);
        if (recieveSataus < 0 )
        {
            printf("Could not recieve message from queue!\n");
            exit(-1);
        }
        else
        {
            if(strcasecmp(message.mtext, "EXIT") == 0)
            {
                printf("Exiting.\n");
                break;
            }
            printf("Process A: %s\n", message.mtext);
        }

        printf("Process B: ");
        fgets(message.mtext, MTEXTSIZE, stdin);
        message.mtext[strcspn(message.mtext, "\n")] = 0; // get rid o
f newline character
        message.mtype = 2;
        if (msgsnd(queueID, &message, strlen(message.mtext)+1, 0) < 0
)
        {
            printf("Error: Failed to send message. Terminating...\n")
;
            break;
        }
    }
    msgctl(queueID, IPC_RMID, 0 );
    return 0;
}
```