Allen Ben Philipose
18BIS0043 (L41+L42)
ECE4025: Embedded Programming
Lab Task - 5

1. Explain with a program, how to copy file data from server to client using shared memory.

Shared memory resource is a pool of memory that is being shared by two or more operations/programs. Each process has its own allocated set of resources and hence if a process wishes to transmit data from its own address space to other process we should address the concept of IPC (that is, inter process communication). Communication may occur between the processes that may be connected or even unrelated.

Typically inter-process communication is achieved using pipes or named pipes, but using shared memory is one of the easiest means of IPC. Shared memory enables two or more processes to control the very same storage in the same way as they can if they all named malloc and received references to the same physical memory. As one process modifies the memory, the update is visible to all the other processes.

→ Fast Local communication

Access to the shared method is as easy as the connection to a process's non-shared memory and it doesn't require a system call or a kernel entry. Additionally, it prevents excessively copying information.

Due to the kernel's lack of synchronization for shared memory accesses, you should implement your own synchronization. For instance, a process does not read from memory until after it has recorded data to it and no two functions should write to it concurrently.

Semaphores, are an often used method for avoiding these race conditions. However, the illustrative programs demostrate only one method accessing memory to emphasize the mutual memory structure and prevent clogging the sample code with synchronizational logic

→ The Memory model

To make sure of a shared memory segment, only one operation must assign it. Then, each procedure that requires access to the section must connect it. Each mechanism detaches the section until it has completed its usage of it. At some stage, the section must be deallocated by one operation.

Allocating a new shared memory section results in the development of virtual memory pages. Although allocating an established section does not create new pages, it does return the identifiers for the existing pages. To use the shared memory section, a method must connect it which creates entries linking the virtual memory to the shared memory segment's shared records.

When the section is full, these mapping entries are deleted. When no other processes are accessing the shared memory segments, only one process is required to deallocate the virtual memory files. Both shared memory segments are allocated as integral multiples of the system's page size, which is described as the number of bytes contained in a memory page.

→ Allocation

SHMGET (Shared Memory GET) has 3 parameters.

i) Integer key indicating the section to be formed. Though defining the same main value, unrelated operations will access the same shared section.

ii) Defines length in section of bytes. Actual amount of bytes alloted is summarily executed to an essential multiple of the page size.

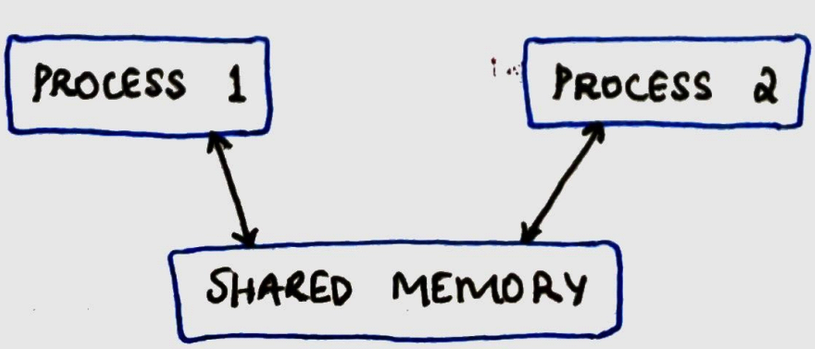
iii) Bitwise or array of flag values specifying choices

- IPC CREAT: Development of a new segment. This enables the creation of a new section when defining the importance of a key

- IPC EXCL: Causes shmget to fail if an already existing section key is defined. It sets up for the "unique" portion of the calling procedure

- Mode flags: Composed of nine bits showing the permissions given to the user, party, and others to manage the section access. Execution pieces are not considered.

As we know, in Inter-process communication, the interaction happens via shared memory which is mutual to both the processes and this allows one process to view the modifications made by the other process



Streams, FIFO and message queues make it difficult for the two processes to communicate since the data must pass through the kernel. Server reads data from the input file and sends it to a response using a pipe. FIFO or a message queue and the device reads data from the IPC channel causing the data to be transferred from the kernel's IPC buffer to the client's IPC buffer once again.

4 replicas are needed (2 write 2 read) for this procedure, but with shared memory the data will be copied only twice : Input to shared memory and shared memory to output.

ftok (): Device call to create a new key.

shmget (): Returns an identifier upon effective completion. It builds the shared memory section or use one that has already been developed.

```
int shmget (key t, size tsize, int shmflag);
```

shmat (): To assign or connect an operation to a shared memory section that has already been formed. shmid is a mutual memory identifier and shmaddr defines special address to use and if 0, OS would pick one for us.

```
void *shmat (int shmit, void *shmaddr, int shmflag);
```

shmdt (): To detach the process from the previously connected shared memory.

```
int shmdt (void *shmaddr);
```

shmctl (): To delete the shared memory space

```
void shmctl (int shmid, IPC RMID, NULL);
```

WRITER'S PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int main() {
    key_t key = ftok("shmfile",65);    // unique key
    int shmid = shmget(key,1024,0666|IPC_CREAT);
      // shmid receives the identifier returned
    char *str = (char*) shmat(shmid,(void *)0,0);

    cout << "Write Data to user Allen: ";
    gets(str)          // Input data

    printf("Data written in memory of
          user Allen : %s \n",str);
    //  Written data is printed

    shmdt(str)         // Detach from shared memory
    return 0;

}
```

READER'S PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

using namespace std;
int main () {
        key_t key = ftok ("shmfile", 65);   // unique key
        int shmid = shmget ( key, 1024, 0666 | IPC_CREAT);
           // shmid receives the identifier returned

        char *str2 = (char*) shmat (shmid, (void *) 0, 0);

        printf (" Data read from memory of
             user Allen: %s \n", str);
        // Read data is printed

         shmdt (str);     // Detach from shared memory
         shmctl (shmid, IPC_RMID, NULL);
         // Destroy the shared memory

         return 0;

}
```

2. Write a program and explain how to transfer a large amount of data between 2 processes using message queues

Advantages of message queues over shared memory

i) Data accessible to all processes sharing memory, hence not secure

ii) When several processes communicate concurrently shared memory data must be secured by synchronization

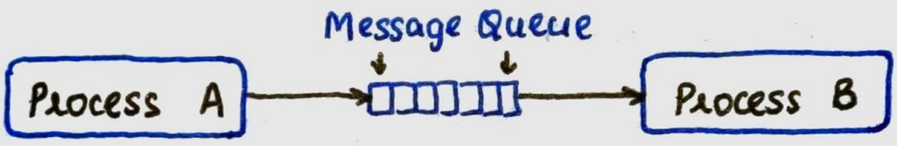
iii) If processes need to communicate via text messages, message queues are the solutions

If the frequency of writing and reading from the shared memory is strong, so implementing the feature will be extremely difficult, pointless to consider consumption. If not all processes need shared memory, its preferable to enforce MQs.

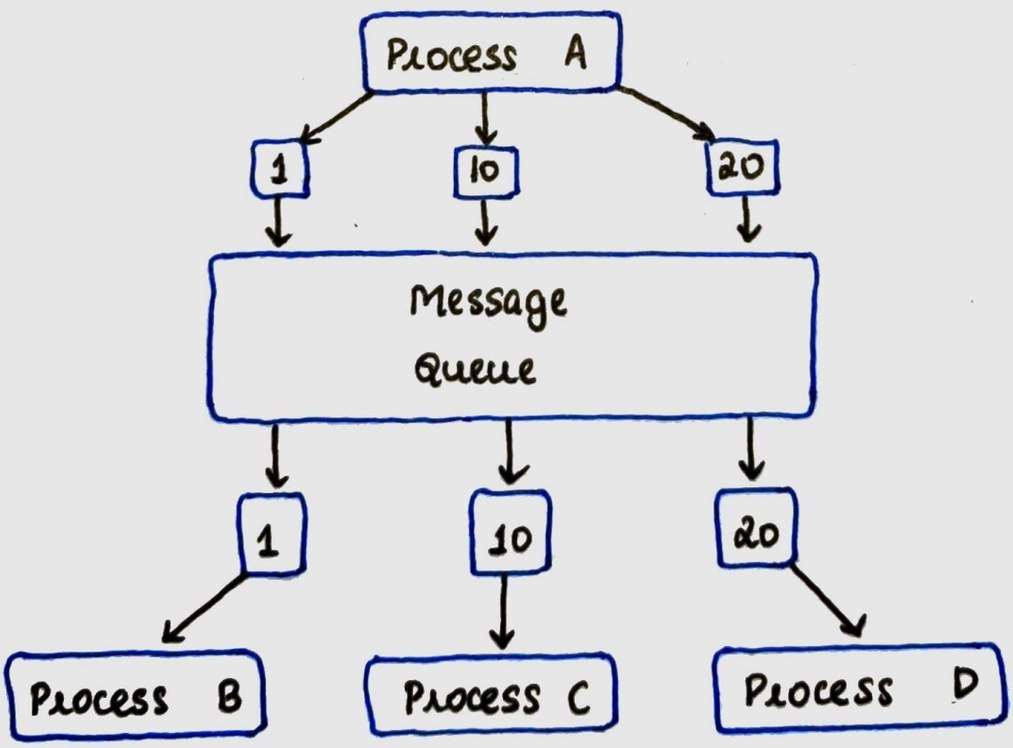
A message queue (MQ) is a related collection of messages that are processed in the kernel and assigned a message queue identifier. Both processes can communicate with one another through a shared device message queue. The transmitting process adds a message to the queue that can be interpreted by another application (via OS message-passing module). Each message is assigned a unique identifier which enables processes to pick the most suitable message.

To obtain entry to the queue in the first place, processes must have a shared key. And since its a queue ultimately, it follows the FIFO concept. The queue's first response is the first to be retrieved. Message Queue vs. Shared Memory should be decided solely on efficiency of the procedure. The following methods of communications are possible if message Queues are used:

- One process writes and another process reads.



- Writing seperate data packets into memory space by a single process and reading from that space done by multiple processes.



* System V is the device call that implements message queues

msgget(): To create a message queue or link to the current one. It either returns the identifier for a newly formed message queue or the indexes for all the queues that has the same key value

msgsnd(): To add a request and data to the message queue and will be added to the end. Each message contains a field of type positive long integer, a non-negative length, and the actual data bytes (corresponding to the length).

msgrcv(): Receive/read messages from the queue. We are not required to receive messages in the order they are sent. We receive messages through their sort sector if we need in order.

msgctl(): For the management of message queue, will be elaborated below.

These commands outline the commands that will manipulate data inside a message queue.

→ Queue management

Usually we use the msgctl() function to delete the message queues but it has multiple parameters with varying functionalities.

i) Element mtype is used to convey information about various message forms. This is the first parameter.

ii) Mtext is an array or other arrangement whose size is determined by the msgsz variable, with a positive value. Default is zero dimension which is permissible.

iii) msgsz specifies the scale of the sent packet and the message should end with a NULL character.

iv) msgtype specifies the message style:

= 0, method reads first message obtained in the queue.

> 0, method reads the first response in the queue if 10, method reads the first message of type 10.

< 0, method reads the first message of the lowest type that is smaller than or equivalent to the number entered. If -3, method reads the first message of type less than 3.

v) Array of flag values specifying choices - msgflg:

- IPC NOWAIT: Returns automatically if no response is detected in the queue.
- MSG NOERROR: Truncates message if its length is more than msgsz bytes
- MSG INFO: Returns structure containing information on device resource usage
- IPC STAT: Copies current values of all the members of struct to the structure referred to by buf. Includes access to read permissions.
- IPC SET: Modifies IDs and permissions by buf.
- IPC RMID: Automatically deletes notification queue.
- IPC INFO: Returns details about message queue limits and parameters specified by buf, which is the layout of type struct msginfo

WRITER'S PROCESS

```
# include <stdio.h>
# include <sys/ipc.h>
# include <sys/msg.h>

# define MAX 10

struct Allen {
        long type1;
        char text1[100];
} message;
// Structure for message queue

int main () {

        key_t key;
        int msgid;

        key = ftok ("progfile", 65);    // Unique key

        msgid = msgget (key, 0666 | IPC_CREAT);
        message.type1 = 1;
        // Create queue and return identifier

        printf ("Write Data: ");
        fgets (message.text1, MAX, stdin);
```

```
        msgsnd (msgid, &message, sizeof (message), 0);

        printf ("Data sent: %s \n",
                message.text1);
        // Display the sent message

        return 0;
}
```

READER'S PROCESS

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct Allen {
        long type1;
        char text1 [100];
} message;
// Structure for message queue

int main() {

        key_t key;
        int msgid;

        key = ftok ("progfile", 65);   // Unique key
```

```
    msgid = msgget(key, 0666 | IPC_CREAT);
    // Create queue and returns identifier

    msgrcv(msgid, &message, sizeof(message), 1, 0);
    // Receive message

    print("Data Received: %s \n",
          message.text1);
    // Display the received message

    msgctl(msgid, IPC_RMID, NULL);
    // Destroy the message queue
    return 0;
}
```

Submitted to:

Prof. Manish Kumar