Inter Process Communication

(Operating System)

Deepika H V

C-DAC Bengaluru

(deepikahv@cdac.in)

Agenda

- Inter process communication
- Message queues,
- Shared memory
- Pipes
- FIFO

- **Processes within a system may be independent or cooperating**
  - **Independent –** It cannot affect or be effected by the other processes executing in the system – Does not share any data with any other processes
  - **Cooperating -** It can affect or be effected by the other processes executing in the system.
- **Reasons for cooperating processes:**
  - Information sharing -- Computation speedup
  - Modularity --Convenience
- **Cooperating processes need inter process communication (IPC)**

Types

- **Pipe**
  - A pipe is a data channel that is unidirectional
- **File**
  - A file is a data record that may be stored on a disk or acquired on demand by a file server.
- **Signal**
  - signals are not used to transfer data but are used for remote commands between processes
- **Shared Memory**
  - memory that can be simultaneously accessed by multiple processes
- **Message Queue**
  - Multiple processes can read and write data to the message queue without being connected to each other
- **Socket**
  - the endpoint for sending or receiving data in a network

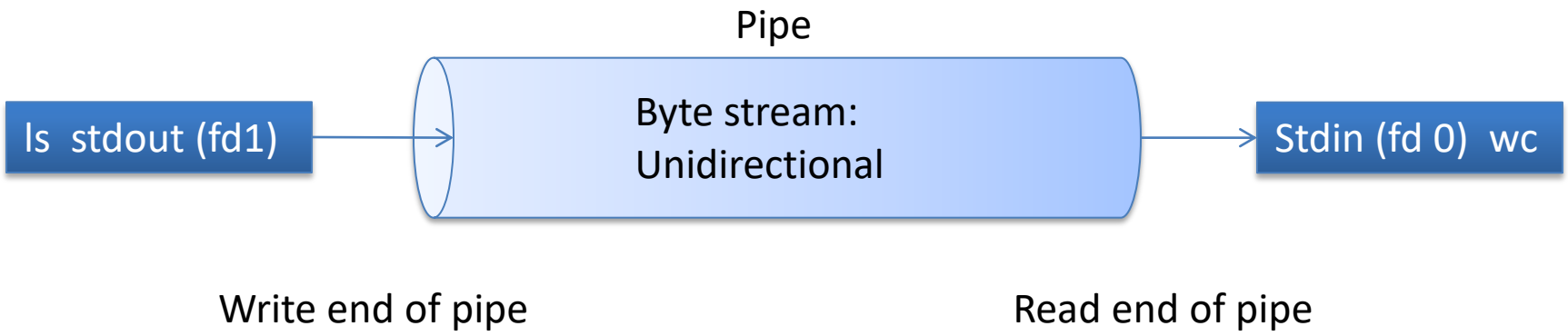
Pipes

Pipes

- Method that contains two end points. Data is entered from one end of the pipe by a process and consumed from the other end by the other process
- **Pipe - byte stream buffer in kernel**
  - Sequential (can't *lseek()*)
  - Multiple readers/writers difficult
- **Unidirectional**
  - Write end + read end
- **2 Types**
  - Ordinary Pipes
  - Named Pipes

Pipes

ls | wc -l



Piping is a process where the *output* of one process is made the *input* of another.

Functioning of a Pipe

Step 1 – Create a pipe.

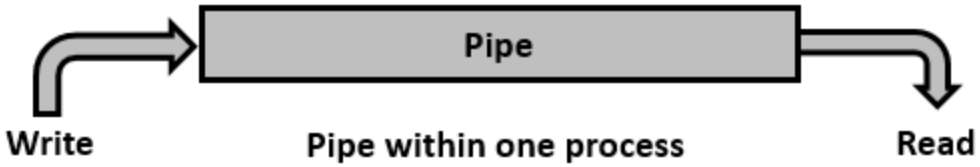
Step 2 – Send a message to the pipe.

Step 3 – Retrieve the message from the pipe and write it to the standard output.

1. Create Pipe

```
#include<unistd.h>

int pipe(int pipedes[2]);
```



- This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function

2. Send & receive message

Send message

```
#include<unistd.h>

ssize_t write(int fd, void *buf, size_t count)
```

Receive message

```
#include<unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

Program to write and read 2 messages using pipe

Algorithm

Step 1 – Create a pipe.

Step 2 – Send a message to the pipe.

Step 3 – Retrieve the message from the pipe and write it to the standard output.

Step 4 – Send another message to the pipe.

Step 5 – Retrieve the message from the pipe and write it to the standard output.

Note – Retrieving messages can also be done after sending all messages.

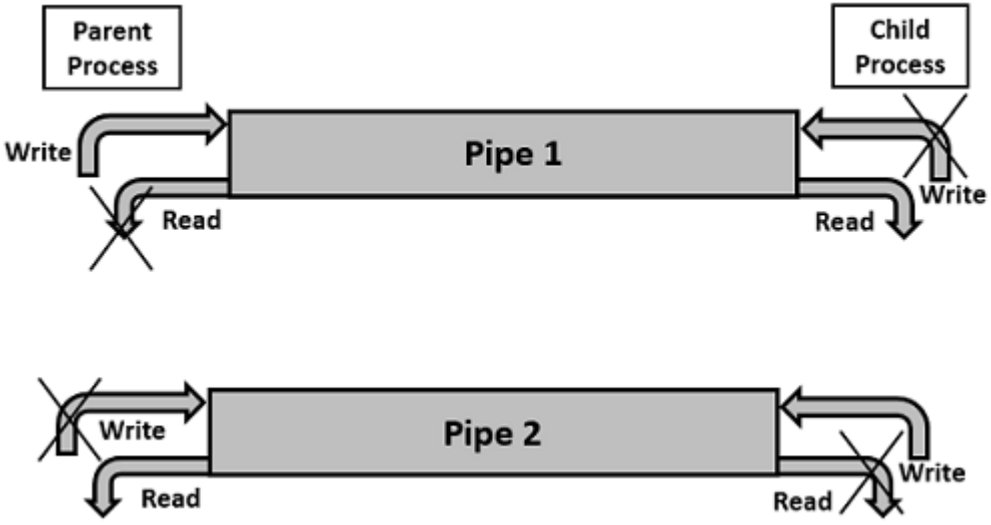
```
main() {
   int fds[2];
   int status;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   status = pipe(fds);

    write(fds[1], writemessages[0], sizeof(writemessages[0]));
    read(fds[0], readmessage, sizeof(readmessage));
    printf("Reading from pipe . Message 1 is %s\n", readmessage);

    write(fds[1], writemessages[1], sizeof(writemessages[1]));
    read(fds[0], readmessage, sizeof(readmessage));
    printf("Reading from pipe . Message 2 is %s\n", readmessage);
}
```

Program: Two-way Communication Using Pipes

- Pipe communication is only **one-way** communication i.e., either the parent process writes and the child process reads or vice-versa but not both.
- what if both the parent and the child needs to write and read from the pipes simultaneously.
- **Two pipes** are required to establish two-way communication.



Step 1 – Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

Step 2 – Create a child process.

Step 3 – Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step 4 – Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step 5 – Perform the communication as required.

FIFO – Named Pipes

- Pipes were meant for communication between **related processes**.
- Can we use pipes for unrelated process communication, say, we want to execute client program from one terminal and the server program from another terminal?
  - Can be achieved with Named Pipes also called FIFO
- We used one pipe for one-way communication and two pipes for bi-directional communication. Does the same condition apply for Named Pipes?
  - Named Pipe supports bi-directional communication

```
#include <sys/types.h>
#include <sys/stat.h>
 int mkfifo(const char *pathname, mode_t mode)
```

Mode:

S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH |S_IFIFO

Reverse String

Step 1 – Create two processes, one is fifo2way_server and another one is fifo2way_client.

Step 2 – Server process performs the following –

- Creates a named pipe (using library function **mkfifo()**) with name in /tmp directory
- Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.
- Opens the named pipe for read and write purposes.
- Waits infinitely for a message from the client.
- If the message received from the client , prints the message and reverses the string. The reversed string is sent back to the client.

Step 3 – Client process performs the following –

- Opens the named pipe for read and write purposes.
- Accepts string from the user.
- Sends message to the server.
- It waits for the message (reversed string) from the server and prints the reversed string on receiving.

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#define FIFO_FILE
"/tmp/fifo_twoway"
void reverse_string(char *str) {
  int last, limit, first;
  char temp;
  last = strlen(str) - 1;
  limit = last/2;
  first = 0;
  while (first < last) {
    temp = str[first];
    str[first] = str[last];
    str[last] = temp;
    first++;
    last--;
  } return; }
```

```
int main() {
  int fd, read_bytes;
  char readbuf[80];
/* Create the FIFO if it does not exist */
  mkfifo(FIFO_FILE, S_IFIFO|0640);

  fd = open(FIFO_FILE, O_RDWR);
read_bytes = read(fd, readbuf,
sizeof(readbuf));
    readbuf[read_bytes] = '\0';

 printf("FIFOSERVER: Received string: \"%s\"
and length is %d\n", readbuf,
(int)strlen(readbuf));

reverse_string(readbuf);
    printf("FIFOSERVER: Sending Reversed
String: \"%s\" and length is %d\n", readbuf,
(int) strlen(readbuf));
    write(fd, readbuf, strlen(readbuf));
sleep(2);
return 0;
}
```

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "/tmp/fifo_twoway"
int main() {
  int fd;
  int stringlen;
  int read_bytes;
  char readbuf[80];

  printf("FIFO_CLIENT: Send messages,
infinitely, to end enter \"end\"\n");

  fd = open(FIFO_FILE, O_CREAT|O_RDWR)

printf("Enter string: ");
    fgets(readbuf, sizeof(readbuf), stdin);
    stringlen = strlen(readbuf);
    readbuf[stringlen - 1] = '\0';
write(fd, readbuf, strlen(readbuf));
      printf("FIFOCLIENT: Sent string: \"%s\"
and string length is %d\n", readbuf,
(int)strlen(readbuf));
      read_bytes = read(fd, readbuf,
sizeof(readbuf));
      readbuf[read_bytes] = '\0';
      printf("FIFOCLIENT: Received string:
\"%s\" and length is %d\n", readbuf,
(int)strlen(readbuf));

close(fd);
return 0;
}
```

PIPEs Vs Named PIPEs

- **PIPE**
  - Between the processes which **share the same file descriptor table** (normally the parent and child processes or threads created by them)
- **Named PIPE**
  - Don't have to start the reading/writing processes at the same time
  - Can control ownership and permissions
  - Across different systems – If a common file system available

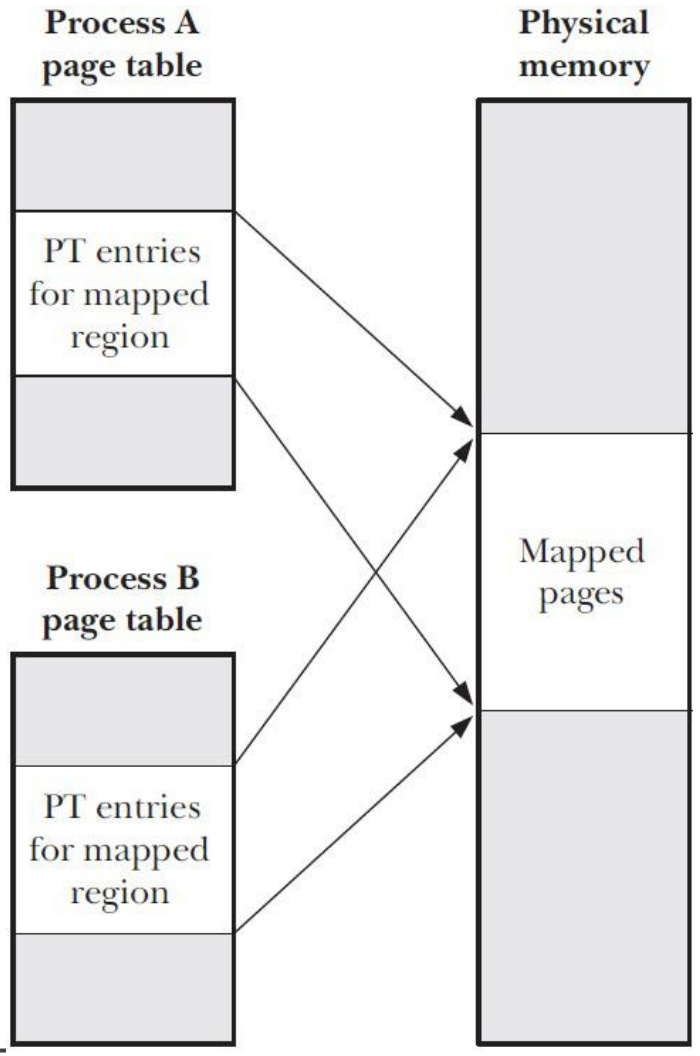
Shared Memory

Shared Memory

- **The shared memory is a mechanism that allows processes to exchange data.**
  - a portion of the memory that is shared between processes.
- **How it works**
  - One process creates the memory area
  - Other process, which has appropriate permissions accesses the memory.
- **Shared among how many process**
  - Shared memory allows several processes to attach a segment of physical memory to their virtual addresses.

Shared memory

- **Processes share physical pages of memory**



Shared memory

- **Processes share same physical pages of memory**
- **Communication means copy data to memory**
- **Efficient compared to Data transfer**
  - Data transfer: user space -> kernel -> user space
- **Shared memory: single copy in user space**
  - Access need to be sync!

Steps to be followed

- Create the shared memory segment or use an already created shared memory segment (**shmget()**)
- Attach the process to the already created shared memory segment (**shmat()**)
- Detach the process from the already attached shared memory segment (**shmdt()**)
- Control operations on the shared memory segment (**shmctl()**)

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg)
```

- Key – Key for the shared memory location for sharing
- Size – size of the memory location requested
- Shmflg – Permissions & creation flag (IPC_CREAT,IPC_EXCL)

```
#include <sys/types.h>
#include <sys/shm.h>
void * shmat(int shmid, const void *shmaddr, int shmflg)
```

attaches the System V shared memory segment identified by shmid to the address space of the calling process

Shmid – shared memory id

Shmaddr- The attaching address , always NULL

Shmflg- Setting permissions

- **Detach the shared memory**

```
#include <sys/types.h>
#include <sys/shm.h>
 int shmdt(const void *shmaddr)
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
 int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

- **Control & deallocation**
- **Cmd option**

✓ **IPC_SET** – Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.

✓ **IPC_RMID** – Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.

✓ **IPC_INFO** – Returns the information about the shared memory limits and parameters in the structure pointed by buf.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
        int i;
        void *shared_memory;
        char buff[100];
        int shmid;

        shmid=shmget((key_t)2345, 1024, 0666);
        printf("Key of shared memory is %d\n",shmid);

        //process attached to shared memory segment
        shared_memory=shmat(shmid,NULL,0);
        printf("Process attached at %p\n",shared_memory);
        printf("Data read from shared memory is : %s\n",(char *)shared_memory);

}
```

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()  {
int i , shmid ;
void *shared_memory;
char buff[100];

shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
printf("Key of shared memory is %d\n",shmid);

shared_memory=shmat(shmid,NULL,0); //process attached to  segment
printf("Process attached at %p\n",shared_memory);

printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

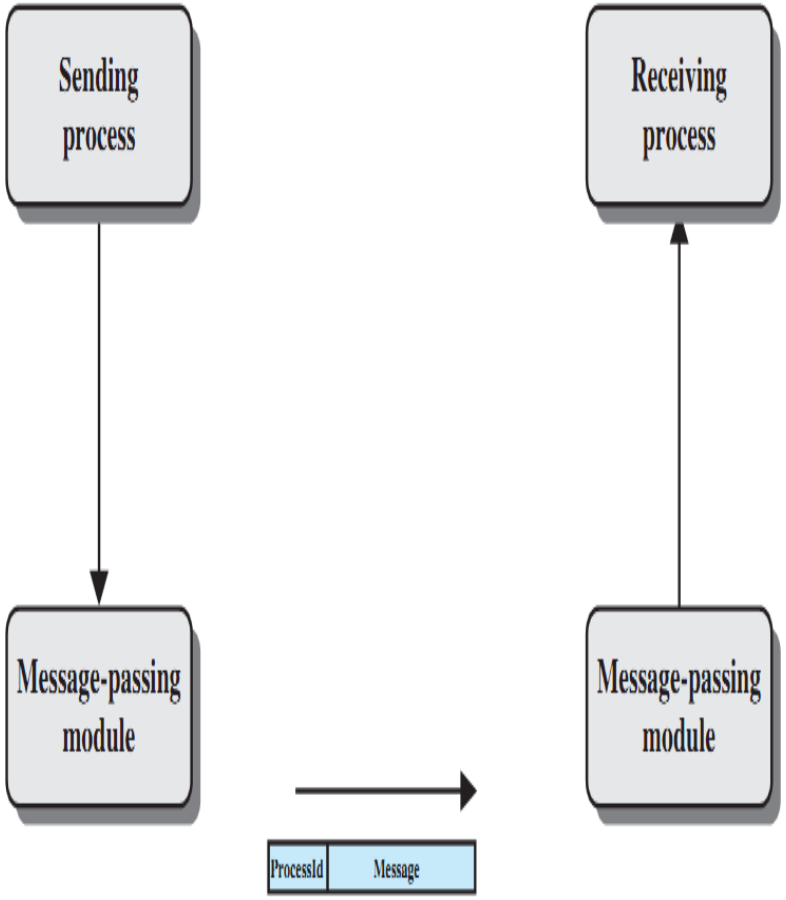
Message Queue

Message Queue

- **Type of Message passing mechanism**
  - processes communicate with each other without resorting to shared variables.
- **It is useful in distributed environments where the processes may reside on different computers connected by network**
- **We have at least two primitives:**
  - **send**(*destination, message*) or **send**(*message*)
  - **receive**(*source, message*) or **receive**(*message*)
- **Message sent can be of fixed or variable size**

- **For 2 processes to communicate a communication link must exist**
- **Link implemented in one of three ways:**
  - Zero capacity – 0 messages. Sender must wait for receiver
  - Bounded capacity – finite length of *n* messages. Sender must wait if link full.
  - Unbounded capacity – infinite length. Sender never waits.

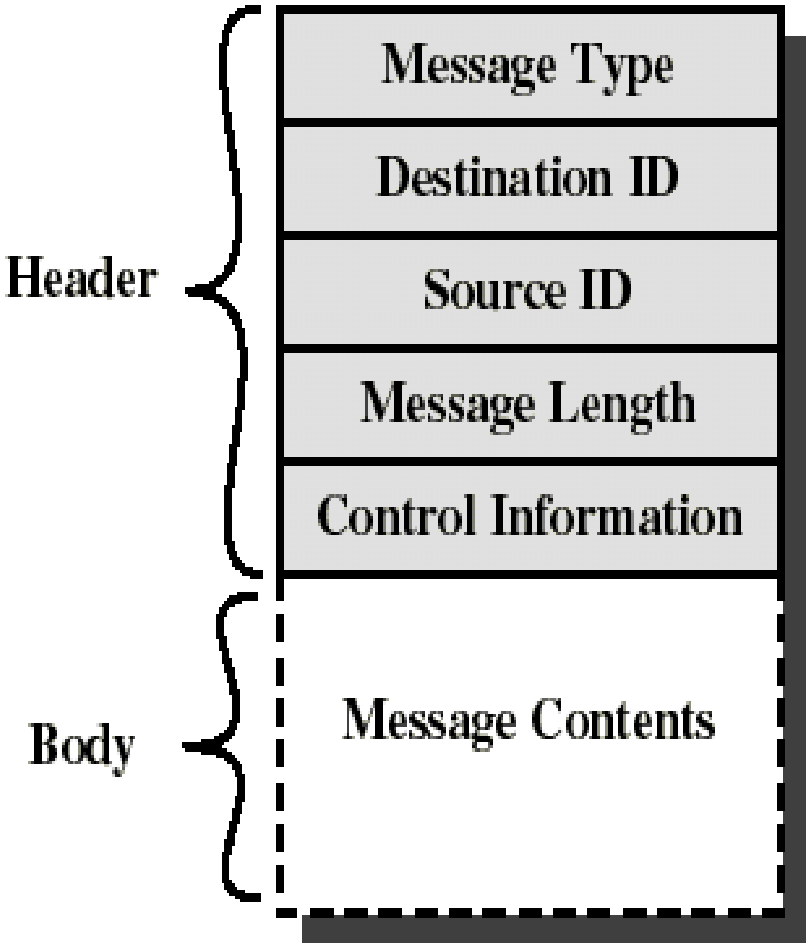
Message Queue



- In a message queue, the sending process calls a system routine for placing the message in a queue that can be read by another process.
- Each process is provided identification or a type, which allows other processes to identify and select it.
- A common key is shared between processes accessing the queue.

Message Format

- **Consists of header and body of message.**
- **Control info:**
  - what to do if run out of buffer space.
  - sequence numbers.
  - priority.
- **Queuing discipline: usually FIFO but can also include priorities.**



- To perform communication using message queues, following are the steps –

Step 1 – Create a message queue or connect to an already existing message queue (**msgget()**)

Step 2 – Write into message queue (**msgsnd()**)

Step 3 – Read from the message queue (**msgrcv()**)

Step 4 – Perform control operations on the message queue (**msgctl()**)

System calls for message queues:

- **msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
  - **int msgget(key_t key, int msgflg);**
- **ftok()**: is use to generate a unique key.
- **msgsnd()**: Data is placed on to a message queue by calling msgsnd().
  - **int msgsnd(int msgqid, const void *msgptr, size_t msgsize, int msgflag);**
- **msgrcv()**: messages are retrieved from a queue.
  - **int msgrcv(int msgqid, void *msgptr, size_t msgsize, long msgtype,int msgflag);**
- **msgctl()**: It performs various operations on a queue. Generally it is use to destroy message queue.
  - **int msgctl(int msqid, int cmd, struct msqid_ds *buf)**

Sending Message

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXSIZE     128
struct msgbuf
{
    long    mtype;
    char    mtext[MAXSIZE];
};
main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    struct msgbuf sbuf;
    size_t buflen;
    key = 1234;
```

```
if ((msqid = msgget(key, msgflg )) < 0){   //Get the message
queue ID for the given key
      perror("msgget");
        exit(1);
        }
    //Message Type
    sbuf.mtype = 1;
    printf("Enter a message to add to message queue : ");
    scanf("%[^\n]",sbuf.mtext);
    getchar();
    buflen = strlen(sbuf.mtext) + 1 ;
    if (msgsnd(msqid, &sbuf, buflen, IPC_NOWAIT) < 0)
    {
        printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype,
sbuf.mtext, buflen);
        perror("msgsnd");
        exit(1);
    }
    else
        printf("Message Sent\n");
    exit(0);
}
```

Receiving Message

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE     128
struct msgbuf
{
    long    mtype;
    char    mtext[MAXSIZE];
};
main()
{
    int msqid;
    key_t key;
    struct msgbuf rcvbuffer;
    key = 1234;
```

```
 if ((msqid = msgget(key, 0666)) < 0)
        {
      perror("msgget()");
        exit(1);
        }
     //Receive an answer of message type 1.
    if (msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0) < 0)
        {
      perror("msgrcv");
        exit(1);
        }
    printf("%s\n", rcvbuffer.mtext);
    exit(0);
}
```

Message Queue	Shared Memory
message is received by a process it would be no longer available for any other process	Message is available to access multiple times until the memory is updated or destroyed
communicate with small message formats	Can create based on the memory required
Messages are sent to destination process on receiving the message gets removed	Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.
Best when small messages needs to be exchanged	Can be used when frequency of writing and reading is high

Thank You