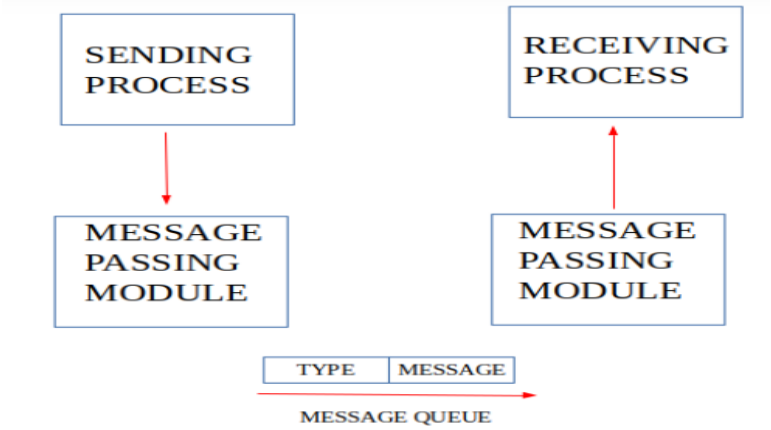
Message Queue

A **message queue** is an inter-process communication (IPC) mechanism that allows processes to exchange data in the form of messages between two processes. . It allows processes to communicate asynchronously by sending messages to each other where the messages are stored in a queue, waiting to be processed, and are deleted after being processed



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())

msgget

Purpose: The **msgget** system call is used to obtain access to a message queue. If a new queue is to be created, **msgget** can also initialize it.

Syntax: The function is defined in C as follows:

int msgget(key_t key, int msgflg);

key_t key: This is an identifier for the message queue. It can be specified explicitly by the programmer, or **IPC_PRIVATE** can be used to generate a unique key.

int msgflg: These flags determine the action to take if the message queue already exists or doesn't exist. It can also include permissions for the queue. Common flags include **IPC_CREAT** (create the message queue if it does not exist), **IPC_EXCL** (ensure the message queue is being created for the first time), along with the usual permission bits (like **0666**).

Return Value: It returns the message queue identifier (a non-negative integer) on success. If it fails, it returns -1 and sets the **errno** variable to indicate the error.

msgsnd

Purpose: **msgsnd** sends a message to the message queue. It allows processes to communicate asynchronously by placing messages onto a queue, which can then be read by other processes.

Syntax: int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

int msqid: This is the message queue identifier returned by **msgget**.

const void *msgp: This is a pointer to the message that needs to be sent. The message must start with a long integer that represents the message type, followed by the actual data.

size_t msgsz: This is the size of the message data in bytes, not including the message type.

int msgflg: These flags control what the function should do if the message queue is full. Common flags include **IPC_NOWAIT** which causes the function to return immediately if the message cannot be sent because the queue is full.

Return Value: On success, **msgsnd** returns 0. On failure, it returns -1

msgrcv

Purpose : **msgrcv** is used to receive messages from a queue. It can be configured to retrieve messages of specific types or simply fetch messages in a first-in, first-out (FIFO) order.

Syntax : size_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

- **int msqid**: This is the message queue identifier returned by **msgget**.
- **void msgp*: This is a pointer to the message buffer into which the received message will be placed. This buffer should be structured to match the message being received, typically starting with a long integer that indicates the message type.
- **size_t msgsz**: This specifies the size of the message buffer, excluding the size of the message type field.
- **long msgtyp**: This parameter determines which message to receive from the queue:

If **msgtyp** is 0, the next message in the queue is received.

If **msgtyp** is greater than 0, the next message of that type is received.

If **msgtyp** is less than 0, the first message of the lowest type that is less than or equal to the absolute value of **msgtyp** is received.

- **int msgflg**: These flags determine how the function behaves if the desired message is not immediately available. Common flags include:

IPC_NOWAIT: Return immediately if no message is available that matches the criteria.

MSG_EXCEPT: Used with a positive **msgtyp** to receive the next message in the queue that is not of type **msgtyp**.

MSG_NOERROR: If the message is larger than **msgsz**, truncate it instead of returning an error.

Return Value:On success, **msgrcv** returns the size of the received message. If it fails, it returns -1

msgctl

Purpose : **msgctl** is used to control or modify the properties of a message queue, or to remove a message queue entirely from the system.

Syntax : int msgctl(int msqid, int cmd, struct msqid_ds *buf);

- **int msqid**: This is the message queue identifier returned by **msgget**.
- **int cmd**: This command argument specifies the action to be performed on the message queue. Common commands include:

  IPC_STAT: Copy information from the kernel data structure associated with **msqid** into the **msqid_ds** structure pointed to by **buf**.

  IPC_SET: Set the message queue attributes using the information in the **msqid_ds** structure pointed to by **buf**.

  IPC_RMID: Remove the message queue. This command immediately deletes the queue and all its messages.

- ***struct msqid_ds buf:** This is a pointer to an instance of **msqid_ds** structure which is used to store or modify message queue attributes. This structure is defined as follows:

```
struct msqid_ds {
   struct ipc_perm msg_perm;  // Ownership and permissions
   time_t        msg_stime; // Time of last msgsnd
   time_t        msg_rtime; // Time of last msgrcv
   time_t        msg_ctime; // Time of last change
   unsigned long   msg_cbytes; // Current number of bytes in queue (non-standard)
   msgqnum_t      msg_qnum;  // Current number of messages in queue
   msglen_t       msg_qbytes; // Maximum number of bytes allowed in queue
   pid_t         msg_lspid;  // PID of last msgsnd
   pid_t         msg_lrpid;  // PID of last msgrcv
};
```

Return Value: On success, msgctl returns 0. If it fails, it returns -1

Sample Program

Sender Process

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct msgbuf {
   long mtype;  // message type must be > 0
   char mtext[200]; // message data
};
int main() {
   key_t key = 1234;  // predefined key
   int msgid;
   struct msgbuf msg;
   // Create a message queue
   msgid = msgget(key, 0666 | IPC_CREAT);
   if (msgid < 0) {
      perror("msgget");
      exit(EXIT_FAILURE);
   }
   // Prepare a message to send
   msg.mtype = 1;
   strcpy(msg.mtext, "Hello, this is message 1");
   // Send the message
   if (msgsnd(msgid, &msg, strlen(msg.mtext), 0) < 0) {
      perror("msgsnd");
      exit(EXIT_FAILURE);
   }
   printf("Sent: %s\n", msg.mtext);
   // Prepare another message
   msg.mtype = 2;
```

```
    strcpy(msg.mtext, "Hello, this is message 2");
    // Send the message
    if (msgsnd(msgid, &msg, strlen(msg.mtext), 0) < 0) {
        perror("msgsnd");
        exit(EXIT_FAILURE);
    }
    printf("Sent: %s\n", msg.mtext);
    return 0;
}
```

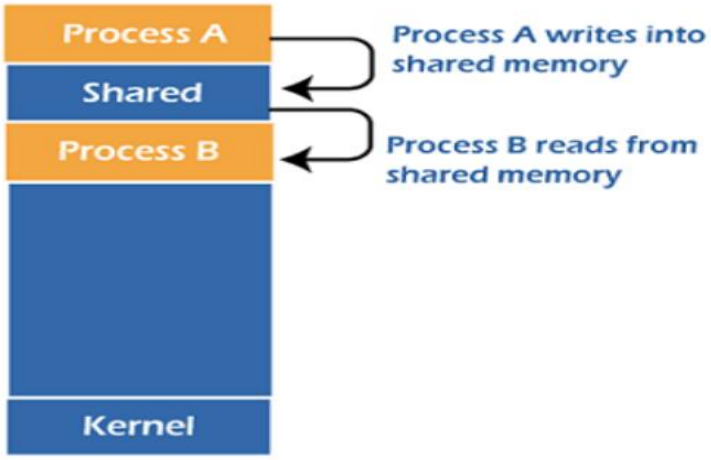
Receiver Process

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct msgbuf {
    long mtype;  // message type must be > 0
    char mtext[200]; // message data
};
int main() {
    key_t key = 1234;  // predefined key
    int msgid;
    struct msgbuf msg;
    // Access the message queue
    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid < 0) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    // Receive a message of any type
    if (msgrcv(msgid, &msg, sizeof(msg.mtext), 0, 0) < 0) {
        perror("msgrcv");
```

```
        exit(EXIT_FAILURE);
    }
    printf("Received: %s\n", msg.mtext);
    // Optionally remove the message queue
    if (msgctl(msgid, IPC_RMID, NULL) < 0) {
        perror("msgctl");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

Shared memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory and communication is done via this shared memory where changes made by one process can be viewed by another process.



The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally, the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

shmget: This function is used to allocate a shared memory segment.

Syntax: int shmget(key_t key, size_t size, int shmflg);

- **key_t key**: Unique identifier for the shared memory segment.
- **size_t size**: The size of the shared memory segment in bytes.
- **int shmflg**: Permission flags and creation flags (**IPC_CREAT**, **IPC_EXCL**).

Returns: On success, it returns the identifier of the shared memory segment; on error, it returns -1

Shmat : This function attaches the shared memory segment identified by the shared memory ID to the address space of the calling process.

Syntax : void *shmat(int shmid, const void *shmaddr, int shmflg);

- **nt shmid**: Shared memory identifier returned by **shmget**.
- **const void shmaddr*: Suggested starting address for the attachment; usually set to **NULL** to let the system choose.
- **int shmflg**: Flags for the operation, typically **0** or **SHM_RDONLY** for read-only access.

Returns: On success, it returns the address of the attached shared memory segment; on failure, it returns **(void *) -1**.

Shmdt: This function detaches the shared memory segment from the address space of the calling process.

Syntax: int shmdt(const void *shmaddr);

- **const void shmaddr*: Address of the shared memory segment to detach.

Returns: On success, returns 0; on error, returns -1.

Shmctl: This function performs various control operations on the shared memory segment.

Syntax: int shmctl(int shmid, int cmd, struct shmid_ds *buf);

- **int shmid**: Shared memory identifier.
- **int cmd**: Control command (**IPC_STAT**, **IPC_SET**, **IPC_RMID**).
- **struct shmid_ds buf*: Pointer to a **shmid_ds** structure to store or modify shared memory metadata.

```
struct shmid_ds {
   struct ipc_perm shm_perm;  // Operation permission structure
   size_t         shm_segsz; // Size of segment in bytes
   time_t         shm_atime; // Last attach time
   time_t         shm_dtime; // Last detach time
   time_t         shm_ctime; // Last change time
   pid_t          shm_cpid;  // PID of creator
   pid_t          shm_lpid;  // PID of last operation
   shmatt_t       shm_nattch; // Number of current attaches
};
```

Return Value: On success, returns 0; on error, returns -1.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int main() {
   key_t key = 1234;  // Identifier for the shared memory segment
   int shmid;
   char *data;
   // Create the shared memory segment
   shmid = shmget(key, 1024, 0666 | IPC_CREAT);
   if (shmid < 0) {
      perror("shmget");
      exit(1);
   }
   // Attach the shared memory segment
   data = shmat(shmid, NULL, 0);
   if (data == (char *)(-1)) {
      perror("shmat");
      exit(1);
```

```
}
// Write to the shared memory
printf("Writing to shared memory: \"Hello, World!\"\n");
strncpy(data, "Hello, World!", 1024);
// Detach the shared memory
if (shmdt(data) < 0) {
   perror("shmdt");
   exit(1);
}

// Reattach the shared memory to read from it
data = shmat(shmid, NULL, 0);
if (data == (char *)(-1)) {
   perror("shmat");
   exit(1);
}
// Read from the shared memory
printf("Reading from shared memory: \"%s\"\n", data);
// Detach the shared memory
if (shmdt(data) < 0) {
   perror("shmdt");
   exit(1);
}
// Remove the shared memory segment
if (shmctl(shmid, IPC_RMID, NULL) < 0) {
   perror("shmctl");
   exit(1);
}
return 0;}
```

Ipc status commands

Interprocess Communication (IPC) mechanisms in Unix-like operating systems, such as semaphores, message queues, and shared memory, offer status commands to help manage and monitor IPC resources. These status commands are generally invoked using control system calls like **semctl**, **msgctl**, and **shmctl**, particularly with the **IPC_STAT** command.

1. Shared Memory (shmctl with IPC_STAT)

shmctl with the **IPC_STAT** command is used to fetch the status of a shared memory segment.

Syntax: int shmctl(int shmid, int cmd, struct shmid_ds *buf);

- **shmid**: Identifier for the shared memory segment.
- **cmd**: Command, where **IPC_STAT** is used to get the status.
- **buf**: Pointer to **shmid_ds** structure where the status data will be stored.

Data Fetched Includes:

- Segment size, owner, permissions
- Time of last attach, detach, and changes
- PIDs of creator and last operator
- Number of current attachments

2. Message Queues (msgctl with IPC_STAT)

msgctl with the **IPC_STAT** command retrieves the status of a message queue.

Syntax: int msgctl(int msqid, int cmd, struct msqid_ds *buf);

- **msqid**: Identifier for the message queue.
- **cmd**: Command, where **IPC_STAT** retrieves the current status.
- **buf**: Pointer to **msqid_ds** structure to store the status data.

Data Fetched Includes:

- Queue size, owner, permissions
- Time of last send and receive operations
- Number of messages currently in the queue
- Total bytes allowed in the queue
- PIDs of processes that last sent or received a message

3. Semaphores (semctl with IPC_STAT)

semctl with the **IPC_STAT** command is used to obtain the status of a semaphore set.

Syntax: int semctl(int semid, int semnum, int cmd, union semun arg);

- **semid**: Identifier for the semaphore set.
- **semnum**: Semaphore number within the set (usually ignored for **IPC_STAT**).
- **cmd**: Command, with **IPC_STAT** used for fetching status.
- **arg**: A union where one member is a pointer to **semid_ds**.

Data Fetched Includes:

- Semaphore array sizes, owner, permissions
- Time of last operations
- Number of semaphores in the set