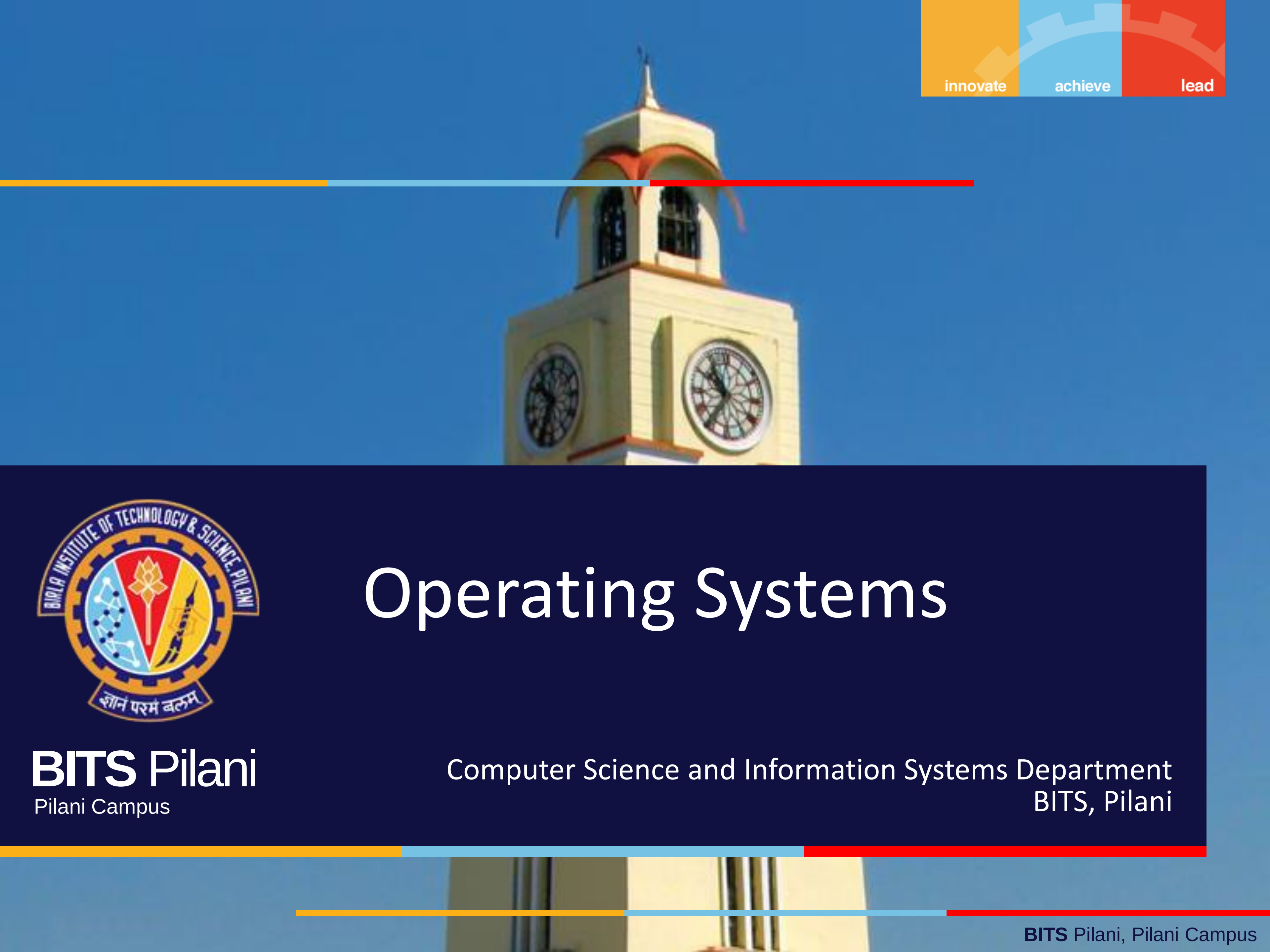


BITS Pilani
Pilani Campus

Operating Systems

Computer Science and Information Systems Department
BITS, Pilani

innovate achieve lead

BITS Pilani
Pilani Campus

Inter Process Communication Tutorial 10 & 11

Today’s Agenda

- Inter Process Communication
  - Pipes
  - FIFOs
  - Message Queues
  - Shared Memory

FIFO

- Semantically, a FIFO is similar to a pipe
- The principal difference is that a FIFO has a name within the file system and is opened in the same way as a regular file.
- This allows a FIFO to be used for communication between unrelated processes
- Once a FIFO has been opened, we use the same I/O system calls as are used with pipes and other files (i.e., read(), write(), and close()
- Any process can open and use FIFO
- FIFOs are also known as named pipes.

Creating a new FIFO

Syntax: mkfifo(pathname, permissions)

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

- Both return: 0 if OK, −1 on error.
- The specification of the mode argument is the same as for the open function.
- The mkfifoat function is similar to the mkfifo function, except that it can be used to create a FIFO in a location relative to the directory represented by by the fd file descriptor argument.

C program to implement FIFO

```
int main() {
   int fd; char arr1[80], arr2[80];
   char * myfifo = "/tmp/myfifo"; // FIFO file path
   //Creating the named file(FIFO)
   //mkfifo(<pathname>, <permission>)
   mkfifo(myfifo, 0666);
   while (1) {
        fd = open(myfifo, O_WRONLY); // Open FIFO for write only
        fgets(arr2, 80, stdin); // Take an input in arr2 from user.
        write(fd, arr2, strlen(arr2)+1);// Write the input in arr2 on FIFO
        close(fd);
        fd = open(myfifo, O_RDONLY); // Open FIFO for Read only
        read(fd, arr1, sizeof(arr1)); // Read from FIFO
        printf("User2: %s\n", arr1); // Print the read message
        close(fd);    }
   return 0;    }
```

Programming Exercises

- Write a C program to implement the above command using FIFO.

Inter Process Communication

- IPC is the set of mechanisms that facilitates the processes
  - to communicate with one another
  - to synchronize their actions
- It is divided into three broad functional categories:
  - **Communication:**
    - These facilities are concerned with exchanging data between processes.
  - **Synchronization:**
    - These facilities are concerned with synchronizing the actions of processes.
  - **Signals:**
    - A signal is a notification to a process that an event has occurred. Signals are sometimes described as software interrupts.
    - Examples: Control-C, Control-Z, division by zero etc.

Other IPC Mechanism

- Message Queues
  - Permit exchange of data between processes in the form of fixed length messages.
- Semaphores
  - Permit multiple processes to synchronize there action.
  - Can be used to implement critical-section problems; allocation of resources.
- Shared Memory
  - enables multiple processes to share the same region of memory
  - It is the fastest form of data communication.

System Calls

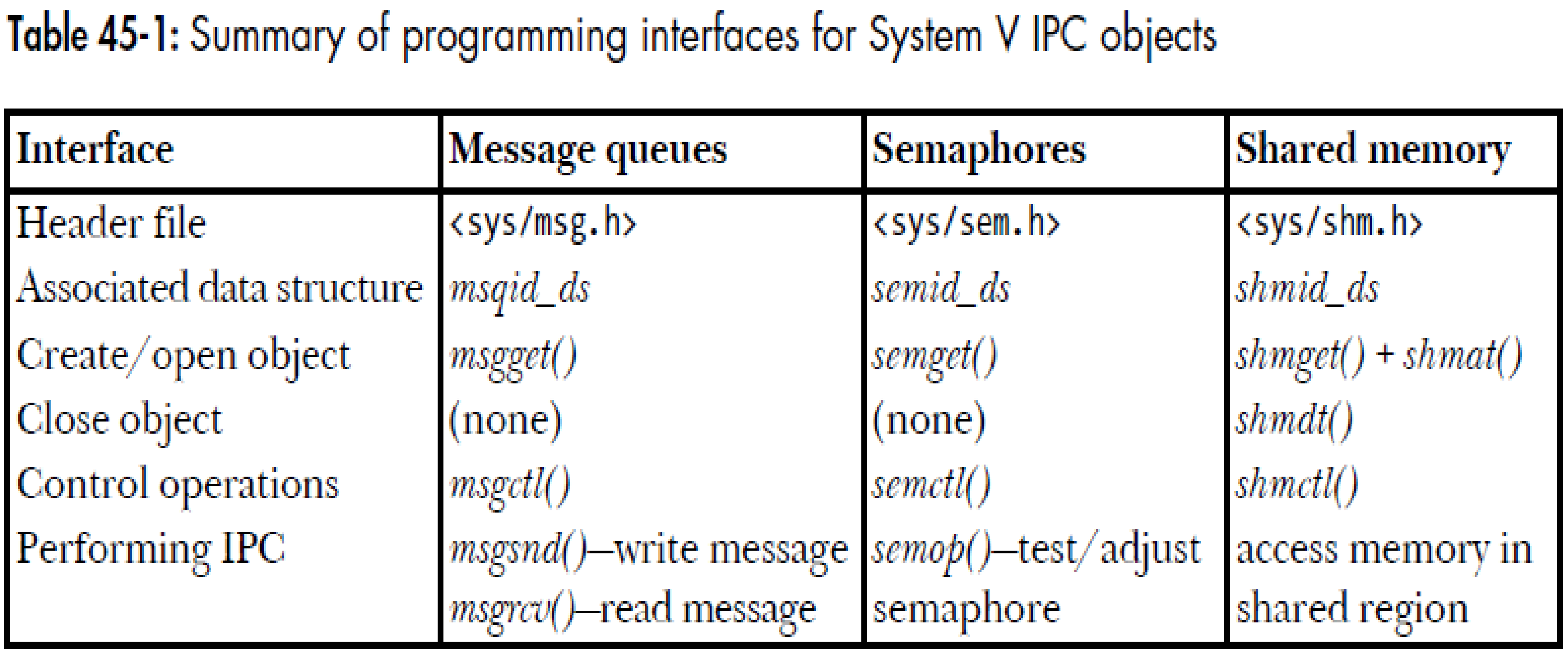
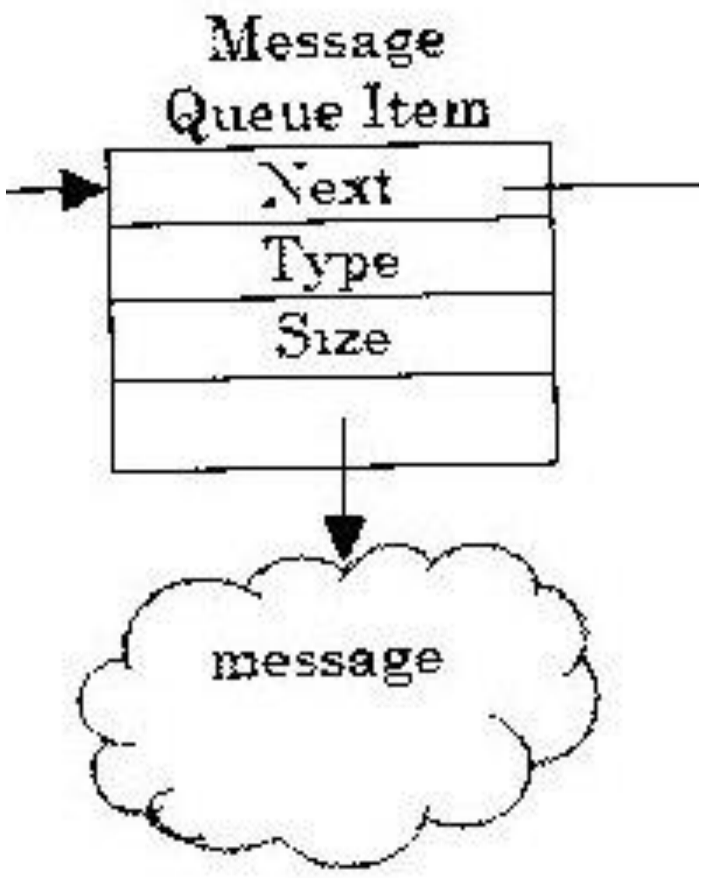


Table 45-1: Summary of programming interfaces for System V IPC objects

Interface	Message queues	Semaphores	Shared memory
Header file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Associated data structure	*msqid_ds*	*semid_ds*	*shmid_ds*
Create/open object	*msgget()*	*semget()*	*shmget()* + *shmat()*
Close object	(none)	(none)	*shmdt()*
Control operations	*msgctl()*	*semctl()*	*shmctl()*
Performing IPC	*msgsnd()*—write message *msgrcv()*—read message	*semop()*—test/adjust semaphore	access memory in shared region

Message Queues

- Data transfer happens through fixed length messages.
- Messages are delimited.
- Messages are transferred in its entirety.
- Transfer of partial message is not allowed.



Structure of Message Queue

```
struct   msg  {
        struct  msg    *msg_next;  /* pointer to next
        message on q */
        long             msg_type;  /*  message type */
        ushort           msg_ts;     /*  message text size */
        short            msg_spot;  /* message text map
        address */
};
```

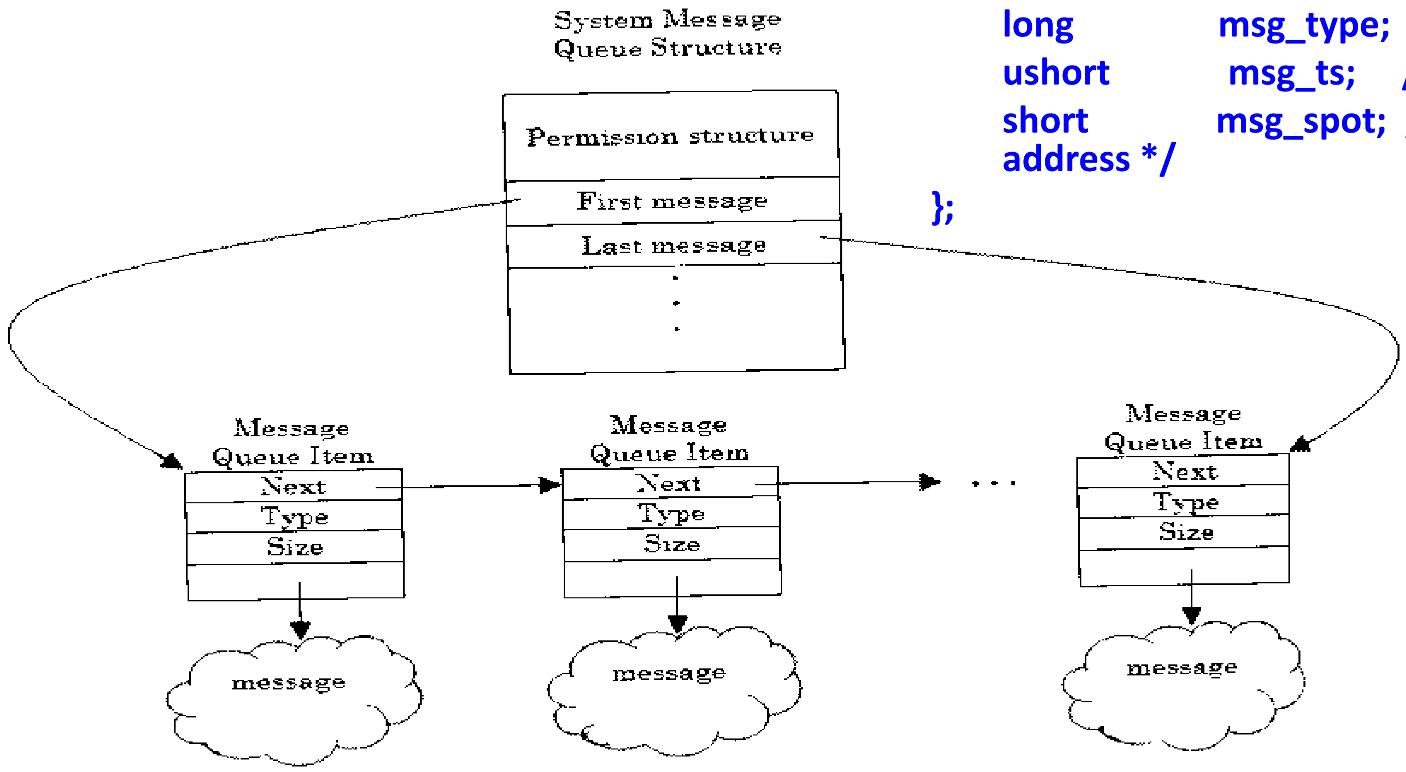


Figure 6.5 A message queue with *N* items.

System calls used for message queues:

- **ftok():** is use to generate a unique key.
- **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.
- **msgsnd():** Data is placed on to a message queue by calling msgsnd().
- **msgrcv():** messages are retrieved from a queue.
- **msgctl():** It performs various operations on a queue. Generally it is use to destroy message queue.

Creating a Message Queue

int msgget(key_t key, int msgflg);

Returns: Success: message queue identifier ; Failure: -1;

- Arguments:
  - ***key***: Can be specified directly by the user, IPC_PRIVATE or generated using ftok.
  - ***msgflg***: IPC_CREAT, S_IRUSR , S_IWUSR or permission value.
  - Generating a unique key with IPC_PRIVATE

    id = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR);
  - Generating a unique key with ftok()

    #include <sys/ipc.h>

    key_t ftok(char *pathname, int proj_id);

    Returns integer key on success, or –1 on error

A typical usage of ftok() is the following:

```
int main()
{
    key_t key;
    int id;

    key = ftok("/mydir/myfile", 1);
    if (key == -1)
        errExit("ftok");

    id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);
    if (id == -1)
        errExit("msgget");
}
```

Sending Message

int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg)

Returns 0 on success, or –1 on error

- The first argument, msgid, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, msgp, is the pointer to the message, sent to the caller
- The third argument, msgsz, is the size of message (the message should end with a null character)
- The fourth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in queue or MSG_NOERROR (truncates message text, if more than msgsz bytes)

Example:

```
// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main() {
    key_t key;     int msgid;
    key = ftok("progfile", 65);                              // ftok to generate unique key

    // msgget creates a message queue and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data : ");
    gets(message.mesg_text);

    msgsnd(msgid, &message, sizeof(message), 0);         // msgsnd to send message

    printf("Data send is : %s \n", message.mesg_text);      // display the message
    return 0;
}
```

Receiving Message

int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)

Returns the number of bytes actually received on success, or –1 on error

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, msgp, is the pointer of the message received from the caller.
- The third argument, msgsz, is the size of the message received (message should end with a null character)
- The fouth argument, msgtype, indicates the type of message −
  - If msgtype is 0 − Reads the first received message in the queue

Receiving Message

  - If msgtype is +ve – Reads the first message in the queue of type msgtype (if msgtype is 10, then reads only the first message of type 10 even though other types may be in the queue at the beginning)
  - If msgtype is –ve – Reads the first message of lowest type less than or equal to the absolute value of message type (say, if msgtype is -5, then it reads first message of type less than 5 i.e., message type from 1 to 5)

- The fifth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in the queue or MSG_NOERROR (truncates the message text if more than msgsz bytes)

Message Queue Control: msgctl()

- **Function:**
  - With *msgctl*, we can do control operations on a message queue identified by msqid.
  - Ownership and access permissions, established when the message queue was created, can be examined and modified using the *msgctl* system call.

Message Queue Control: msgctl()

< sys/types.h> <sys/ipc.h> < sys/msg.h>

int msgctl(int msgid, int cmd, struct msqid_ds *buf)

Returns 0 on success, or –1 on error

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, cmd, is the command to perform the required control operation on the message queue. Valid values for cmd are −
  - **IPC_STAT** − Copies information of the current values of each member of struct msgid_ds to the passed structure pointed by buf. This command requires read permission on the message queue.

Message Queue Control: msgctl()

- **IPC_SET** – Sets the user ID, group ID of the owner, permissions etc pointed to by structure buf.
- **IPC_RMID** – Removes the message queue immediately.
- **IPC_INFO** – Returns information about the message queue limits and parameters in the structure pointed by buf, which is of type struct msginfo
- **MSG_INFO** – Returns an msginfo structure containing information about the consumed system resources by the message queue.

• The third argument, buf is a pointer to the message queue structure named struct msqid_ds. The values of this structure would be used for either set or get as per cmd.

msqid_ds Structure

```
struct  msqid_ds {
    struct ipc_perm  msg_perm;              /* defines permissions for
                                               message operations */
    struct  msg  *msg_first;                /* pointer to first message on q*/
    struct  msg  *msg_last;                 /* point to last message on q */
    ulong         msg_cbytes;               /* current # bytes on q */
    ulong         msg_qnum;                 /* # of message on q */
    ulong         msg_qbytes;               /* max # of bytes on q */
    pid_t         msg_lspid;                /* pid of last msgsnd */
    pid_t         msg_lrpid;                /* pid of last msgrcv */
    time_t        msg_stime;                /* last msgsnd time  */
    time_t        msg_rtime;                /* time of last msgrcv() */
  ………….
  ………….
};  /* total 17 members */
```

ipc_perm Structure

- struct ipc_perm {

```
        key_t     key;           /* Key used for msgget */
        uid_t     uid;           /*owner's user ID */
        gid_t     gid;           /* owner's group ID */
        uid_t     cuid;          /* creator's user ID */
        gid_t     cgid;          /* creator's group ID */
        mode_t    mode;          /* access modes */
        ulong     seg;           /* slot usage sequence number */
        long      pad[4];        /* reserve area */
    };
```

- Struct msqid_ds {

```
    struct          ipc_perm          msg_perm; .....};
```

Access Mode

- The mode field is a 9-bit field that contains the permissions for message operations.
- The first three bits identify owner permissions;
- the second three bits identify group permissions; and
- the last three bits identify other permissions.
- In each group, the first bit indicates read permission; the second bit indicates write permission; and the third bit is not used.

IPC_STAT

- The kernel maintains an instance of this structure for each queue which exists in the system. By using the IPC_STAT command, we can retrieve a copy of this structure for examination. Let's look at a quick wrapper function that will retrieve the internal structure and copy it into a passed address:

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
        if( msgctl( qid, IPC_STAT, qbuf) == -1) { return(-1); }
        return(0);
}
```

- The passed buffer should contain a copy of the internal data structure for the message queue represented by the passed queue identifier (qid)

IPC_SET

- The only modifiable item in the data structure is the ipc_perm member. This contains the permissions for the queue, as well as information about the owner and creator. However, the only members of the ipc_perm structure that are modifiable are mode, uid, and gid.
- You can change the owner's user id, the owner's group id, and the access permissions for the queue.

IPC_SET

```
int change_queue_mode( int qid, char *mode )
{
	struct msqid_ds tmpbuf;

	/* Retrieve a current copy of the internal data structure */
	get_queue_ds( qid, &tmpbuf);

	/* Change the permissions using an old trick */
	sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);

	/* Update the internal data structure */
	if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
	{ return(-1); }
	return(0);
}
```

IPC_RMID

- After successfully retrieving a message from a queue, the message is removed. However, as mentioned earlier, IPC objects remain in the system unless explicitly removed, or the system is rebooted. Therefore, our message queue still exists within the kernel. To complete the life cycle of a message queue, they should be removed using the IPC_RMID command:

```
int remove_queue( int qid )
{
        if( msgctl( qid, IPC_RMID, 0) == -1) { return(-1); }
        return(0);
}
```

Example:

```
struct mesg_buffer {
   long mesg_type;
   char mesg_text[100];
} message;

int main() {
   key_t key;
   int msgid;

   key = ftok("progfile", 65);                    // ftok to generate unique key

   // msgget creates a message queue and returns identifier
   msgid = msgget(key, 0666 | IPC_CREAT);

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

   // display the message
   printf("Data Received is : %s \n", message.mesg_text);

   msgctl(msgid, IPC_RMID, NULL);                 // to destroy the message queue

   return 0;
}
```

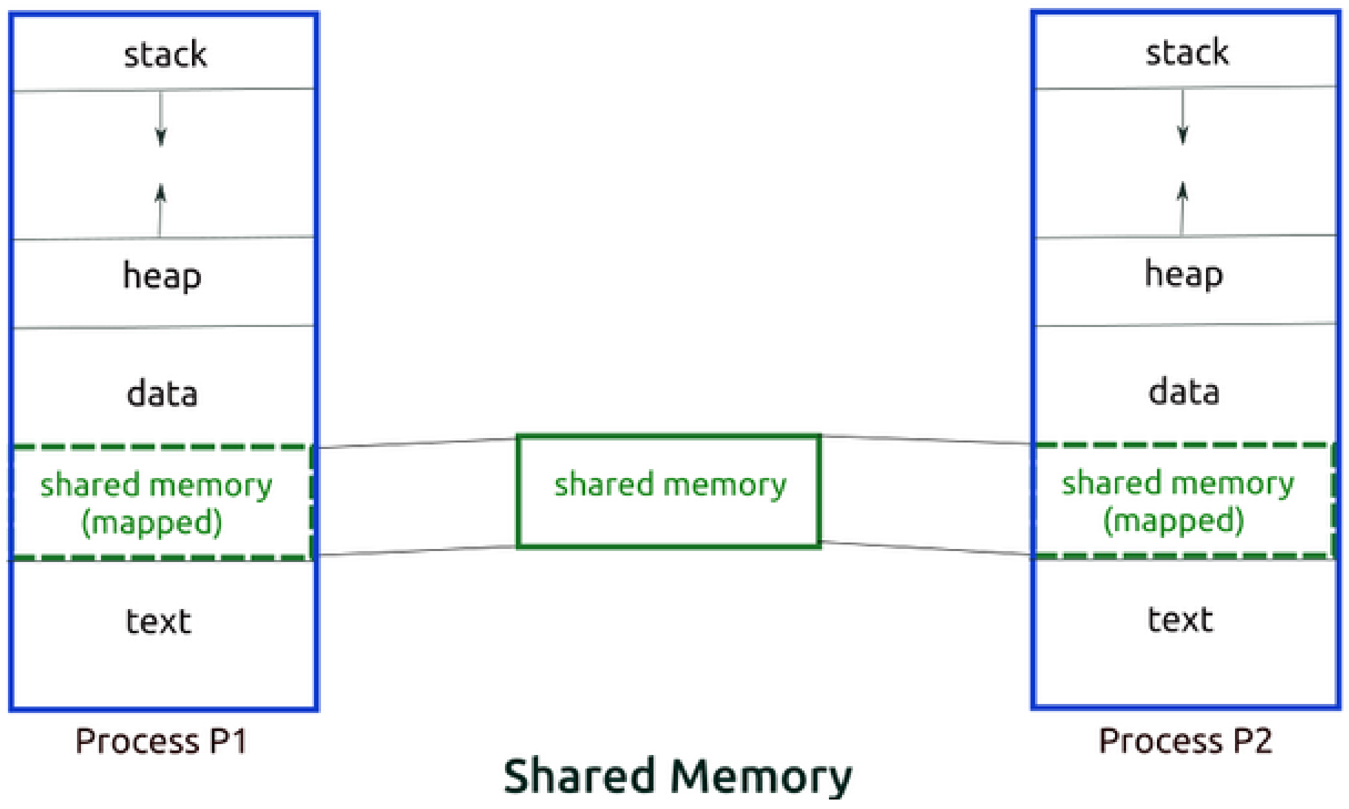
Command meaning

S.No.	command name	Meaning
1	IPC_RMID	The message queue is removed
2	IPC_STAT	Transfers the kernel information for the message queue in the msqid_ds structure pointed by buf
3	IPC_SET	The message queue properties in the kernel for the queue from the msqid_ds structure pointed by buf
4	IPC_CREAT	The queue is created, if it does not already exist
5	IPC_EXEC with IPC_CREAT	The message queue exists, msgget returns -1, with errno set to EEXIST
6	IPC_EXEC	Useless
7	IPC_PRIVATE	Generating key
8	IPC_NOWAIT	don’t wait if message queue is full

Shared Memory

- Shared memory segment is created by the kernel and mapped to the data segment of the address space.
- Process can use it just like any other global variable in its address space.
- This IPC mechanism removes the copy of messages from user to kernel space and from kernel to user space as in pipes, FIFOs and message queues.
- The process simply write data to the shared memory. As soon as it is written, it is available for the other process.
- Fastest way of communication.
- But sharing of data needs synchronization.

Shared Memory



Shared Memory: System Calls

- **int shmget (key_t key, size_t size, int shmflg);**

  (Returns: Success: message queue identifier ; Failure: -1;**)**
- **key_t ftok(char *pathname, int proj);**

  (Returns integer key on success, or –1 on error**)**
- **void *shmat (int shmid, const void *shmaddr, int shmflg);**

  (returns pointer to the attached shared memory segment. On error, (void *) -1 is returned)
- **int shmdt (const void *shmaddr);**

  (On success, shmdt returns 0. On error, shmdt returns -1.)
- **int shmctl (int shmid, int cmd, struct shmid_ds *buf);**

Shared Memory: System Calls

- **shmget**

#include <sys/ipc.h>
#include <sys/shm.h>

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

Arguments:

- ***key***: Can be specified directly by the user, IPC_PRIVATE or generated using ftok.
- ***Size:*** size specifies the size of the shared memory segment to be created
- ***shmflg***: IPC_CREAT, S_IRUSR , S_IWUSR or permission value.
- Generating a unique key with IPC_PRIVATE
  - **Id** = **shmget(key,SHMSIZE,IPC_CREAT|0666))**
- Generating a unique key with ftok()

  #include <sys/ipc.h>

  key_t ftok(char *pathname, int proj);

  Returns integer key on success, or –1 on error

Shmat: shared memory attach

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

  On success, shmat returns pointer to the attached shared memory segment. On error, (void *) -1 is returned

- **int shmdt (const void *shmaddr);**

  On success, shmdt returns 0. On error, shmdt returns -1.

Shmctl: shared memory control operation

- **int shmctl (int shmid, int cmd, struct shmid_ds *buf);**
- commands are IPC_STAT, IPC_SET and IPC_RMID.
- IPC_STAT command copies the data in the kernel data structure shmid_ds for the shared memory into the location pointed by the parameter buf
- IPC_RMID, marks a shared memory segment for removal from the system.
- IPC_SET command, we can set some of the fields in the shmid_ds structure in the kernel for the shared memory segment.

The data structure for shared memory segments in the kernel

```
struct shmid_ds {
    struct ipc_perm shm_perm;  /* Ownership and permissions */
    size_t shm_segsz;          /* Size of segment (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;            /* ID of last shmat(2)/shmdt(2) */
    shmatt_t shm_nattch;       /* No. of current attaches */ ...
};
```

ipc_perm Structure

```
struct ipc_perm {
    key_t __key;              /* Key supplied to shmget(2) */
    uid_t uid;                /* Effective UID of owner */
    gid_t gid;                /* Effective GID of owner */
    uid_t cuid;               /* Effective UID of creator */
    gid_t cgid;               /* Effective GID of creator */
    unsigned short mode;      /* Permissions + SHM_DEST and
                                 SHM_LOCKED flags */
    unsigned short __seq;     /* Sequence number */
};
```

Shared Memory for Writer Process:

```
int main()
{
   key_t key = ftok("shmfile",65);           // ftok to generate unique
key

   // shmget returns an identifier in shmid
   int shmid = shmget(key,1024,0666|IPC_CREAT);

   // shmat to attach to shared memory
   char *str = (char*) shmat(shmid,(void*)0,0);

   cout<<"Write Data : ";
   gets(str);
   printf("Data written in memory: %s\n",str);

   shmdt(str);                               //detach from shared memory
   return 0;
}
```

Shared Memory for Reader Process

```
int main()
{
   key_t key = ftok("shmfile",65);              // ftok to generate unique key

   // shmget returns an identifier in shmid
   int shmid = shmget(key,1024,0666|IPC_CREAT);

   // shmat to attach to shared memory
   char *str = (char*) shmat(shmid,(void*)0,0);

   printf("Data read from memory: %s\n",str);

   shmdt(str);                                   //detach from shared memory

   shmctl(shmid,IPC_RMID,NULL);                  // destroy the shared memory

   return 0;
}
```

Any Queries?