Course	Inter Process Communication Message Queue and Shared Memory and Semaphore	
Module Day 5		3 Hours
	At the end of this module you will be able to know: ▪ How to overcome the limitations of FIFO's ▪ What is a Message Queue and working with message Queue? ▪ Advantages and limitations of Message Queues ▪ System Calls related to Message Queue ▪ What is Shared Memory and working with Shared Memory? ▪ What are the advantages and limitations of Shared Memory? ▪ System Calls related to Shared Memory. ▪ What is Semaphore? ▪ Why do we need to use Semaphores? ▪ Advantages and Disadvantages of Semaphores, ▪ System Calls related to Semaphores,	

Session Plan	
1	Introduction to Message Queue
2	Working with Message Queue
3	System Calls related Message Queue
4	Introduction to Shared Memory
5	Working with Shared Memory
6	System Calls related to Shared Memory
7	Introduction to Semaphores, advantages and disadvantages of Semaphores, System calls related to Semaphores.
8	Examples and Assignments related to Semaphores

	Module Overview

- ✓ Introduction to Message Queues
- ✓ Introduction to msqid_ds structure, ipc_perm structure

- ✓ Advantages and limitations of Message Queues
- ✓ System calls related to Message Queues
- ✓ Introduction to Shared Memory and shmid_ds structure
- ✓ System calls related to Shared Memory.
- ✓ Introduction to Semaphores,
- ✓ semid_ds , sem and ipc_perm Structures,
- ✓ System Calls related to Semaphores,

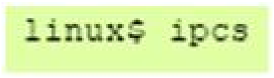
Introduction to Message Queues

The designers of UNIX found the types of interprocess communications that could be implemented using signals and pipes to be restrictive. To increase the flexibility and range of interprocess communication, supplementary communication facilities were added. These facilities, added with the release of System V in the 1970s, are grouped under the heading IPC (Interprocess Communication).

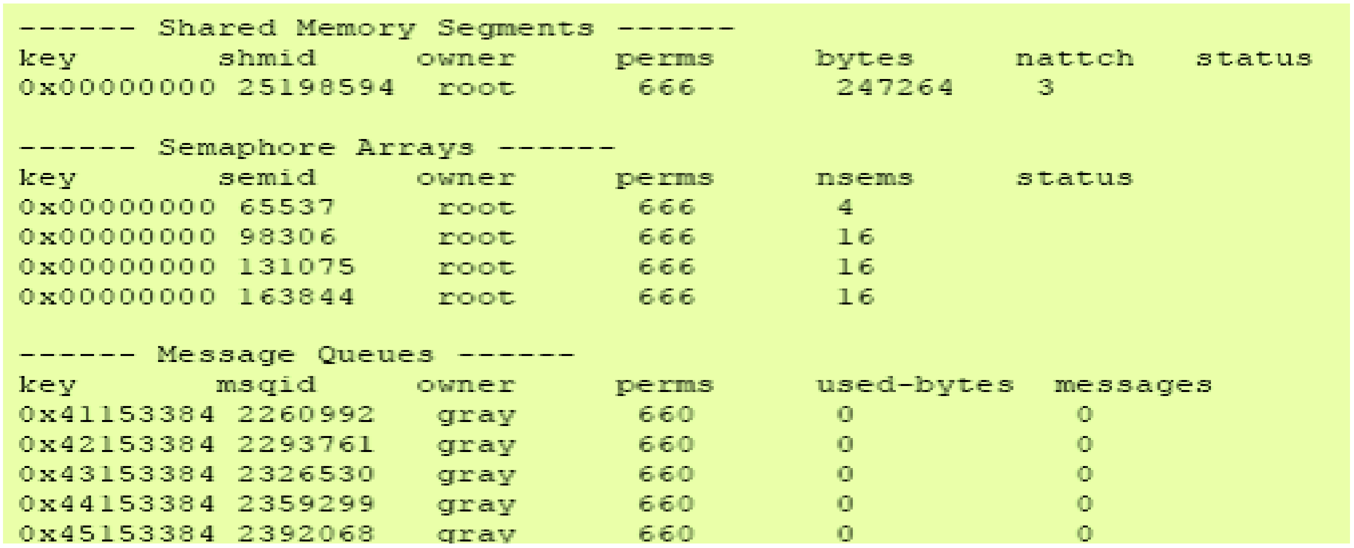
 Message Queues— Information to be communicated is placed in a predefined message structure. The process generating the message specifies its type and places the message in a system-maintained Message Queue. Processes accessing the Message Queue can use the message type to selectively read messages of specific types in a first-in-first-out (FIFO) manner. Message queues provide the user with a means of asynchronously multiplexing data from multiple processes.

- In LINUX only 16 Message Queues are available
- Kernel will create a structure called **msqid_ds** for one Queue,
- When you request to kernel to create a Message Queue, then Kernel will create a structure **msqid_ds** and that structure address will return back by converting type of integer.
- Message queues allow one or more processes to write messages that will be read by one or more reading processes.
- Linux maintains a list of Message Queues, the msgque vector: each element of which points to an **msqid_ds** data structure that fully describes the Message Queue.

- When Message Queues are created, a new **msqid_ds** data structure is allocated from system memory and inserted into the vector.
- So Message Queues are also examples for sharing information between two or more processes.



```
linux$ ipcs
```

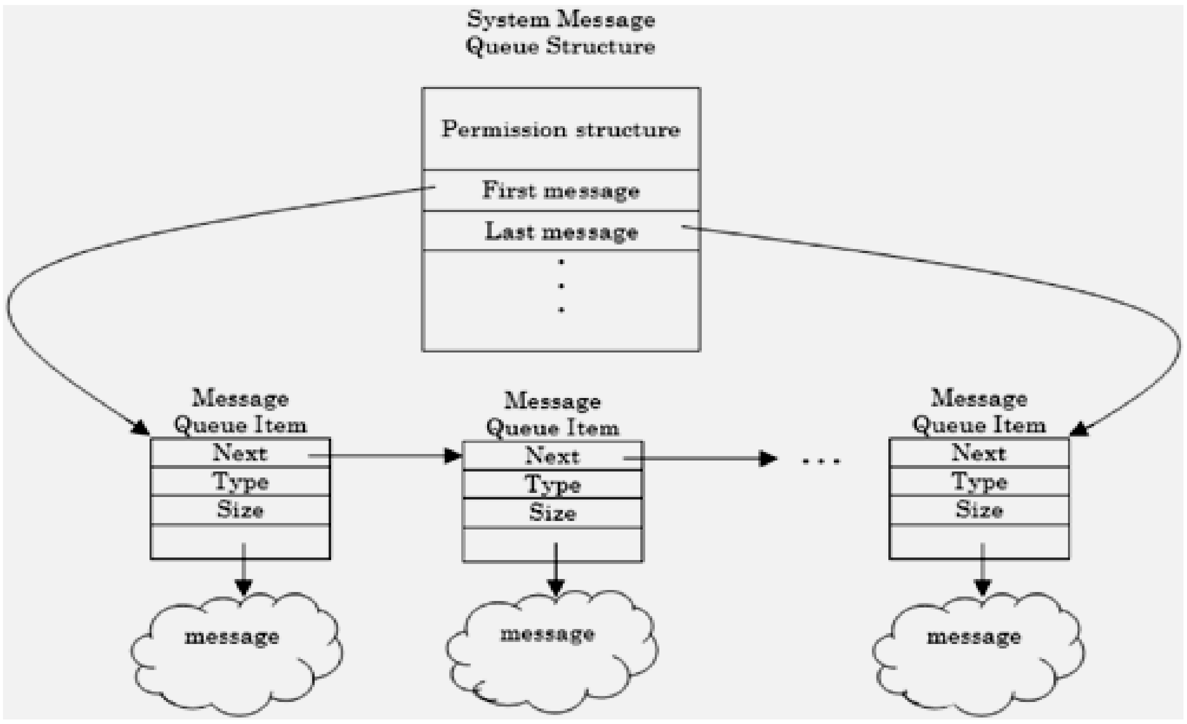


```
------ Shared Memory Segments ------
key        shmid      owner      perms      bytes      nattch    status
0x00000000 25198594   root        666        247264     3

------ Semaphore Arrays ------
key        semid      owner      perms      nsems      status
0x00000000 65537      root        666        4
0x00000000 98306      root        666        16
0x00000000 131075     root        666        16
0x00000000 163844     root        666        16

------ Message Queues ------
key        msqid      owner      perms      used-bytes  messages
0x41153384 2260992    gray        660        0           0
0x42153384 2293761    gray        660        0           0
0x43153384 2326530    gray        660        0           0
0x44153384 2359299    gray        660        0           0
0x45153384 2392068    gray        660        0           0
```

Along with `ipcs` command additionally, -s, -q, or -m can be used to indicate semaphore, Message Queue, or shared memory, and can be followed by –i and a valid decimal ID to display additional information about a specific IPC resource.

Some IPC resources exist and maintain their contents even after the process that created them has terminated. An IPC resource can be removed by its owner, using the appropriate system call within a program or by using the system-level command ipcrm. The Message Queue, shown in the output of the previous ipcs command, could be removed by its owner issuing the command.

msqid_ds and ipc_perm Structures:



msqid_ds:

The kernel creates, stores, and maintains an instance of this structure for every Message Queue created on the system. It is defined in `linux/msg.h`. One msqid structure for each queue on the system is as follows:

```
struct msqid_ds {
     struct ipc_perm msg_perm;
     struct msg *msg_first;  /* first message on queue */
     struct msg *msg_last;   /* last message in queue */
     time_t msg_stime;       /* last msgsnd time */
     time_t msg_rtime;       /* last msgrcv time */
     time_t msg_ctime;       /* last change time */
     struct wait_queue *wwait;
     struct wait_queue *rwait;
     ushort msg_cbytes;
```

```
          ushort msg_qnum;
          ushort  msg_qbytes;         /*  max  number  of  bytes  on
queue */
          ushort msg_lspid;       /* pid of last msgsnd */
          ushort msg_lrpid;       /* last receive pid */
};
```

`msg_perm.cuid` and **`msg_perm.uid`** are set to the effective user ID of the calling process.

`msg_perm.cgid` and **`msg_perm.gid`** are set to the effective group ID of the calling process.

The least significant 9 bits of **`msg_perm.mode`** are set to the least significant 9 bits of msgflg.

`msg_qnum`, **`msg_lspid`**, **`msg_lrpid`**, **`msg_stime`** and **`msg_rtime`** are set to 0.

`msg_ctime` is set to the current time.

`msg_qbytes` is set to the system limit MSGMNB.

In the above structure:

`msg_perm`:

An instance of the `ipc_perm` structure, which is defined for us in `linux/ipc.h`. This holds the permission information for the Message Queue, including the access permissions, and information about the creator of the queue (uid, etc). The ipc_perm structure is defined in <`sys/ipc.h`> as follows:

```
     struct ipc_perm {
              ushort uid;         // owner user id
              ushort gid;         // owner group id
              ushort cuid;        // creators user id
              ushort cgid;        // creators group id
              short mode;         // access modes
              key_t key;          // key value
     };
```

`msg_first` - Link to the first message in the queue (the head of the list).

`msg_last` - Link to the last message in the queue (the tail of the list).

`msg_stime` - Timestamp (`time_t`) of the last message that was sent to the queue.

`msg_rtime` - Timestamp of the last message retrieved from the queue.

`msg_ctime` - Timestamp of the last ``change'' made to the queue (more on this later).

`Wwait and rwait` - Pointers into the kernel's wait queue. They are used when an operation on a Message Queue deems the process go into a sleep state (i.e. queue is full and the process is waiting for an opening).

`msg_cbytes` - Total number of bytes residing on the queue (sum of the sizes of all messages).

`msg_qnum` - Number of messages currently in the queue.

`msg_qbytes` - Maximum number of bytes on the queue.

`msg_lspid` - The PID of the process who has sent the last message.

`msg_lrpid` - The PID of the process who retrieved the last message.

System Calls related to Message Queues

Functionality	**System Call**
Allocate an IPC resource; gain access to an existing IPC resource.	msgget
Control an IPC resource: obtain/modify status information, remove the resource.	msgctl
IPC operations: send/receive messages, perform semaphore operations, attach/free a shared memory	msgsnd msgrcv

<u>Creating Message Queue:</u>

Message Queue can be created by using **msgget()** system call. This function declaration is given below:

Summary of msgget() system call

Include Files	<sys/types.h> <sys/ipc.h> <sys/msg.h>	Manual section	2
Summary	**int msgget (key_t key, int msgflag);**		
Return	Success	Failure	
	Non-negative message queue identifier associated with key	-1	

The value for the argument key can be specified directly by the user or generated using the `ftok` library function. The value assigned to key is used by the operating system to produce a unique message queue identifier. The low-order bits of the msgflg argument are used to determine the access permissions for the message queue. Additional flags (e.g., `IPC_CREAT`, `IPC_EXCL`) may be `ORed` with the permission value to indicate special creation conditions.

Ex:- /****** WAP to create a Message Queue *************/

```
Header files
int main()  {
     int  msgid;
     msgid = msgget( 555, IPC_CREAT | 0777);
     if(msgid == -1) {
          printf(“error in creating Message Queue:\n”);
          exit(0);
     }
     printf(“Message  Queue  is  created  successfully  with  id:  %d
     and 555 as key value\n”, msgid);
```

```
    return 0;
}
```

Sending a message to Queue:

Messages are placed in the message queue (sent) using the system call `msgsnd.`

Summary of `msgsnd()` system call

Include Files	<sys/types.h> <sys/ipc.h> <sys/msg.h>	Manual section	2
Summary	**`int msgsnd( int msgid, struct msgbuf *msgp, size_t msgsize, int msgflag);`**		
Return	Success	Failure	
	0	-1	

Here:

– msgid -> return value of **`msgget`**()

– ptr -> pointer to structure msgbuf

```
struct msgbuf {
        long int type;
        char data[256];
 };
```

– msgsize -> length of the message

– msgflag -> IPC_NOWAIT or ZERO

Note: Before sending message to Queue first we must declare the msgbuf structure with member **`long int type`** followed by your message members. Here **type** member is compulsory.

Ex:- /*************** WAP to send a message to Queue *************/

```
Header files
int main()  {
     struct msgbuf  {
          long int type;
          char text[256];
     }msg;
     int  msgid, var;
     msgid = msgget( 555, IPC_CREAT | 0777); //  creating  new  MQ
                                             with 555 key
     if(msgid == -1) {
          printf("error in creating Message Queue:\n");
          exit(0);
     }
     printf("Message  Queue  is  created  successfully  with  id:  %d
     and 555 as key value\n", msgid);
     printf("initialize the members in msgbuf structure:\n");
     printf("enter message type: ");
     scanf("%d", &msg.type);
     printf("\nenter your message: ");
     scanf("%s", msg.text);
     var = msgsnd(msgid, &msg, sizeof(msg), 0);
     if(var == -1) {
          printf("\nerror in sending message\n");
          exit(1);
     }
     printf("\nmessage sent successfully\n");
     return 0;
}
```

Reading a message from Queue:

Messages are retrieved from the message queue using the system call `msgrcv`.

Summary of `msgrcv()` system call

Include Files	<sys/types.h> <sys/ipc.h> <sys/msg.h>	Manual section	2
Summary	**`int msgrcv( int msgid, struct msgbuf  *msgp, size_t msgsize,long msgtyp int msgflag);`**		
Return	Success	Failure	
	Number of bytes received	-1	

Here:

- `msgid` -> return value of **`msgget`**`()`
- `msgp` -> pointer to structure msgbuf

```
Struct msgbuf
{
     long int type;
     char data[256];
};
```

- `msgsize` -> length of the message
- `msgtyp` -> type of the message
- `msgflag` -> `IPC_NOWAIT` or `ZERO`

Ex:- /********* WAP to receive a message from Queue ************/

```
Header files
int main()  {
     struct msgbuf  {
          long int type;
          char text[256];
     }msg;
     int  msgid;
     msgid = msgget( 555, 0777);   //  Opening  MQ  with  555  key
wich is created
```

```
     if(msgid == -1) {
          printf("error in creating Message Queue:\n");
          exit(0);
     }
     printf("key 555 Message Queue opened in this process with
     id: %d\n", msgid);
     var = msgrcv(msgid, &msg, sizeof(msg), 6, 0); // receiving
type 6 message from Queue.
     If(var == -1) {
          Printf("error in receiving type 6 message from
Queue\n");
          exit(1);
     }
     printf("the received message is: %s\n", msg.text);
     return 0;
}
```

<u>Controlling Message Queue:</u>

The ownership and access permissions, established when the message queue was created, can be examined and modified using the `msgctl` system call

The `msgctl` system call references the message queue indicated by the `msqid` argument. The value of the `cmd` argument is used to indicate the action that `msgctl` should take. The following defined constants/actions can be specified:

Summary of `msgctl()` system call

Include Files	<sys/types.h> <sys/ipc.h> <sys/msg.h>	Manual section	2
Summary	**`int msgctl( int msgid,int cmd,struct msqid_ds *buf);`**		
Return	Success	Failure	
		-1	

Ex:- /************* WAP to delete a Message Queue ***********/

```
int main()  {
     int  msgid, var;
     msgid = msgget(555, 0777);
     if(msgid == -1) {
          printf(“error in creating Message Queue\n”);
          exit(0);
     }
     var = msgctl(msgid, IPC_RMID, 0);
     if(var == -1) {
          printf(“Fails to delete Queue\n”);
          exit(1);
     }
     printf(“Queue deleted successfully\n”);
     return 0;
}
```

Ex:- /************* WAP to print uid and gid of a Message Queue ***********/

```
int main()
{
     int  msgid, var;
     msgid = msgget(555, 0777);
     if(msgid == -1)
     {
          printf(“error in creating Message Queue\n”);
          exit(0);
     }
     var = msgctl(msgid, IPC_STAT, &my_msqid_ds); //    to    get
msqid_ds

structure
     if(var == -1)
     {
          printf("Fails to get msqid_ds structure of  Queue\n");
          exit(1);
     }
     printf("Owner id: %d\n", my_msqid_ds.msg_perm.uid);
     printf("Group id: %d\n", my_msqid_ds.msg_perm.gid);

     msgctl(msgid, IPC_RMID, 0);   // to delete Queue
     return 0;
}
```

Advantages and Limitations:

- We can read the message in any order from the Message Queue,
- We can store up to 2G strands,
- Less number of system calls used.
- The only one limitation of Message Queues is data transmission delay between two processes.

Key learning:

1. What are the advantages of Message Queue over FIFO?
2. The command to see IPC's is ________.
3. After msgget ,command to check whether massage queue is created or not is __________.
4. What are the data structures maintained by the kernel for Message Queue?
5. WAP to get and set message q attributes.

Introduction to Shared Memory

Shared memory allows multiple processes to share virtual memory space.

This is the fastest but not necessarily the easiest (synchronization-wise) way for processes to communicate with one another.
In general, one process creates or allocates the Shared Memory segment. The size and access permissions for the segment are set when it is created.

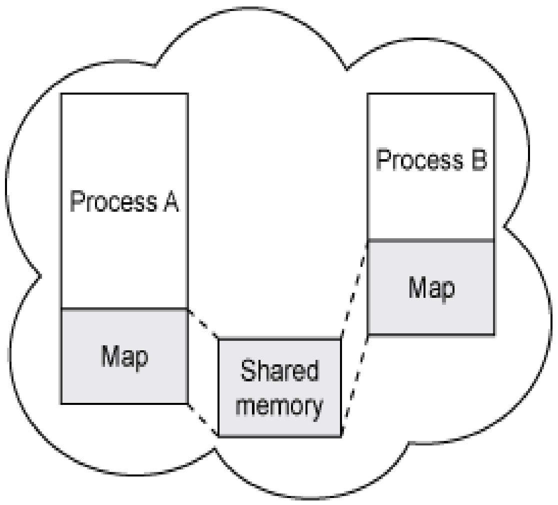
The process then attaches the shared segment, causing it to be mapped into its current data space. If needed, the creating process then initializes the shared memory.

Once created, and if permissions permit, other processes can gain access to the Shared Memory segment and map it into their data space. Each process accesses the Shared Memory relative to its attachment address.

While the data that these processes are referencing is in common, each process uses different attachment address values. For each process involved, the mapped memory appears to be no different from any other of its memory addresses.

When a process is finished with the Shared Memory segment, it can detach from it. Additionally, the creator of the segment may grant ownership of the segment to another process. When all processes are finished with the Shared Memory segment, the process that created the segment is usually responsible for removing it.

The actual mapping of the segment to virtual address space is dependent upon the memory management (MMU) hardware for the system.



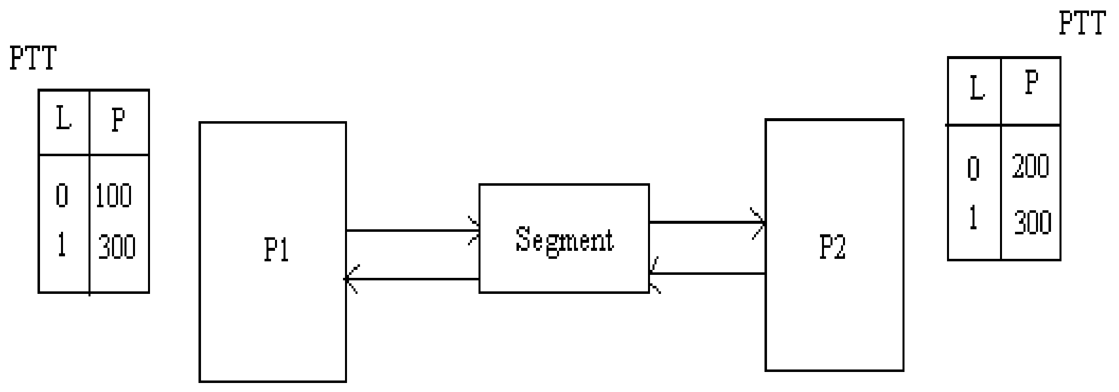
A piece of kernel memory is shared between the processes. Shared Memory is allocated from kernel memory. This memory is used like a global array by all attached processes.

- To overcome the data transmission delay in Message Queues, Shared Memory concept was implemented,
- For every process one page transmission table will be created by MMU.

PTT

Logical Page	Physical Page
0	100
1	200
2	300
3	400

- **PTT:** mapping of physical address for corresponding logical address.
- If process-1 wants to communicate with process-2, any one process will identify the one free page/segment and attach to it's PTT.
- Now another process will attach the same segment to it's PTT by using key value.
- For example, in the diagram shown below both processes P1 and P2 are sharing same memory segment having address as 300.



- Now both processes will share the same segment.
- Here communication b/w the processes will happen in USER SPACE.
- So data transmission delay is not there in Shared Memory hence, **Shared Memory is one of the fastest IPC mechanism in LINUX.**

- When you request to get one free page segment, Kernel will create one structure called **shmid_ds**.

shmid_ds and ipc_perm Structures:

As with Message Queues, the kernel maintains a special internal data structure for each Shared Memory segment which exists within its addressing space. This structure is shmid_ds.

The data structure for each Shared Memory segment in the system is shown below:

```
struct shmid_ds {
        struct ipc_perm shm_perm;        /* operation perms */
        int     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 */
        unsigned short  shm_cpid;        /* pid of creator */
        unsigned short  shm_lpid;        /* pid of last operator */
         short   shm_nattch;             /* no. of current attaches */
                                  /* the following are private */
         unsigned short   shm_npages;    /* size of segment (pages) */
         unsigned long   *shm_pages;     /* array of ptrs to frames -> SHMMAX */
         struct vm_area_struct *attaches; /* descriptors for attaches */
};
```

Operations on this structure are performed by a special system call, and should not be tinkered with directly. Here are descriptions of the more pertinent fields:

shm_perm

This is an instance of the ipc_perm structure, which is defined for us in linux/ipc.h. This holds the permission information for the segment, including the access permissions, and information about the creator of the segment (uid, etc).

shm_segsz

Size of the segment (measured in bytes).

shm_atime

Time the last process attached the segment.

shm_dtime

Time the last process detached the segment.

shm_ctime

Time of the last change to this structure (mode change, etc).

shm_cpid

The PID of the creating process.

shm_lpid

The PID of the last process to operate on the segment.

shm_nattch

Number of processes currently attached to the segment.

System Calls related to Shared Memory

The `shmget` system call is used to create the shared memory segment and generate the associated system data structure or to gain access to an existing segment. The shared memory segment and the system data structure are identified by a unique shared memory identifier that the `shmget` system call returns.

Summary of `shmget()` system call

Include Files	<sys/ipc.h> <sys/shm.h>	Manual section	2
Summary	**`int shmget ( key_t key, int size,int`**		
Return	Success	Failure	
		-1	

To identify/get free memory segment:

To identify the free memory segment, we have to use the following system call.

```
int shmget ( key_t key, size_t size, int flag);
```

- key -> unique key value
- size -> size of the segment in bytes
- flag -> IPC_CREAT|0766

Return Value:

On success, returns the id of shmid_ds structure,

On failure, returns -1.

Ex:- /*******************WAP to get 512 bytes of Shared Memory *******************/

Header files

```
int main() {
     int shid;
     shid = shmget(111, 512, IPC_CREAT | 0777);
     if(shid == -1) {
          printf("error in identifying free segment of size 512 bytes\n");
          exit(0);
     }
     printf("Success in identifying the free memory of 512 bytes\n");
     printf("the id of Shared Memory is: %d\n", shid);
     return 0;
}
```

<u>Read/Write from/to Shared Memory:</u>

Prior to read from or write to Shared Memory, `shmat()` should be done. This system is used to attach the segment address to process PTT and it returns a pointer to Shared Memory which can be used for accessing shared memory. The function declaration of `shmat()` is given below:

Summary of `shmat()` system call

Include Files	<sys/types.h> <sys/shm.h>	Manual section	2
Summary	**`void *shmat ( int shmid, const void *shmaddr,int shmflg);`**		
Return	Success	Failure	
	Reference to the data segment	-1	

Here :

shmid	->	which is returned by shmget()	
*shmaddr	->	0	- system choose the address and returns
		non zero	- Address to be specified
shmflg	->	SHM_RDONLY	- Read only
		Zero	- Read and Write.

Ex:- /*********** WAP to get 10 bytes of Shared Memory and write “hello” to it ***********/

Header files

```
int main()  {
     int shid; char *ptr;
     shid  = shmget(222, 10, IPC_CREAT | 0777);
     if(shid  < 0) {
          printf(“shmget error\n”);
          exit(0);
     }
     ptr = shmat(shid, 0, 0);
     if(ptr == NULL) {
          printf("error in shmat\n");
          exit(1);
     }
     memcpy(ptr, "hello", 6); // to copy 6 bytes of data from
"hello" address to ptr.
     return 0;
}
```

Note:- In above program, to use `memcpy`() function we have to include `string.h` header file. This is similar to `write` function. In case of shared memory, to read from or to write to Shared Memory we can use `printf`() and `scanf`() functions respectively.

To detach the Shared Memory:

To detach the Shared Memory from process PTT we have to use `shmdt`() system call. This function declaration is given below:

Summary of `shmdt()` system call

Include Files	<sys/types.h> <sys/shm.h>	Manual section	2
Summary	**void *shmdt (const void *shmaddr);**		
Return	Success	Failure	
	0	-1	

Here:

shmaddr is pointer address returned by shmat().

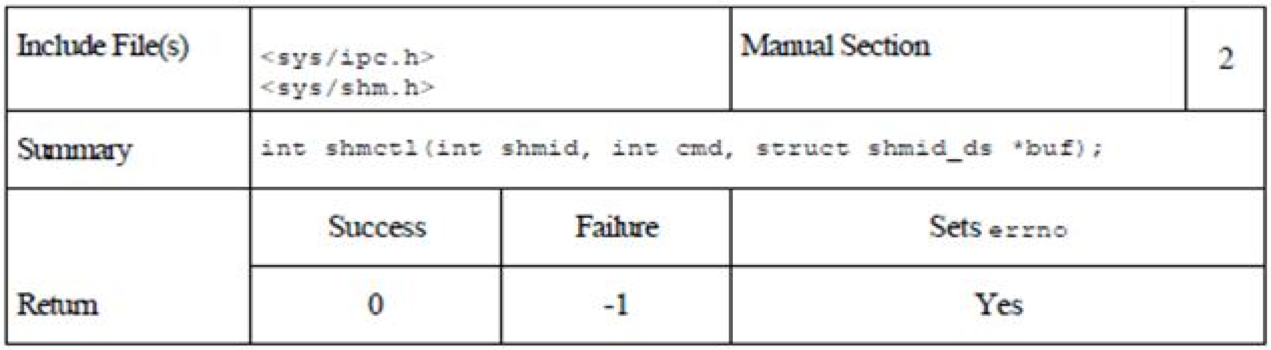
Ex:- /*************** WAP to detach the Shared Memory *******************/

Header files

```
int main()  {
     int shid; char *ptr;
     shid  = shmget(222, 1024, IPC_CREAT | 0777);
     if(shid  < 0) {
          printf(“shmget error\n”);
          exit(0);
     }
     ptr = shmat(shid, 0, 0);
     if(ptr == NULL) {
          printf(“error in shmat\n”);
          exit(1);
     }
     // Do read and write operation
     shmdt(ptr);
     return 0;
}
```

Controlling Shared Memory:

Shared Memory Controlling can be done by using shmctl() function. Here using this system call you can get the shmid_ds structure also. This system call decleration is given below.



Include File(s)	`<sys/ipc.h>` `<sys/shm.h>`		Manual Section	2
Summary	`int shmctl(int shmid, int cmd, struct shmid_ds *buf);`			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Here:

- shmid -> return value of shmget()
- cmd -> IPC_RMID, IPC_STAT, IPC_SET etc
- arg -> Zero if no arguments

Ex:- /*********WAP to delete Shared Memory segment from kernel memory*******/

Header files

```
int main()  {
     int shid; int var;
     shid  = shmget(222, 1024, 0777);   //    getting    already
existing Shared memory
     if(shid  < 0) {
          printf("shmget error\n");
          exit(0);
     }
     var = shmctl(shid, IPC_RMID, 0);
     if(var == -1) {
          printf("error in deleting segment\n");
          exit(1);
     }
     printf("Shared Memory deleted succesfully\n");
     return 0;
}
```

Note:- Usage of `shmctl` system call is same as `msgctl()` call.

Advantages:

- Efficieny (elimination of unnecessary data copy).
- Reduced complexity (single step update rather than read, write, modify buffer).

Introduction to Semaphores

Conceptually, a semaphore is a data structure that is shared by several processes. Semaphores are most often used to synchronize operations when multiple processes access a common, non-shareable resource. By using semaphores, we attempt to avoid starvation (which occurs when a process is habitually denied access to a resource it needs) and deadlock (which occurs when two or more processes each hold a resource that the other needs while waiting for the other process to release its resource). When used to synchronize the access to a resource, a semaphore is initially set to the number of available resources. Each time a process wants to obtain the resource, the associated semaphore is tested. A positive, nonzero semaphore value indicates the resource is available. To indicate it has gained access to the resource, the process decrements the semaphore. For events to progress correctly, the test and decrement operation on the semaphore must be atomic (i.e., non-interruptible/indivisible). If the tested semaphore is zero, indicating the resource is not available, the requesting process must wait. When a process is finished with a semaphore-associated resource, the process indicates the return of the resource by incrementing the semaphore. Once a resource is returned, other processes that have been waiting for the resource are notified by the system. Semaphores that control access to a single resource, taking the value of 0 (resource is in use) or 1 (resource is available), are often called binary semaphores.

- To solve the synchronization problem in Shared Memory, Locking mechanism was implemented.
- Semaphore is a tool used for locking mechanism.
- It is an integer value
- Lock - increment 0 to 1,
- Unlock - decrement 1 to 0.
- Semaphores can be thought of as simple counters that indicate the status of a resource.

- This counter is a protected variable and cannot be accessed by the user directly. The shield to this variable is provided by none other than the kernel.
- The usage of this semaphore variable is simple. If counter is 0, then the resource is available, and if the counter is greater than 0, then that resource is busy or being used by someone else.
- Semaphores can also be used to protect Critical regions.

semid_ds and sem Structures:

As with message queues, the kernel maintains a special internal data structure for each semaphore set which exists within its addressing space. This structure is of type semid_ds, and One semid_ds data structure for each set of semaphores in the system is defined in linux/sem.h as follows:

```
struct semid_ds {
            struct ipc_perm sem_perm;          /* permissions .. see ipc.h */
            time_t        sem_otime;           /* last semop time */
            time_t        sem_ctime;           /* last change time */
            struct sem     *sem_base;          /* ptr to first semaphore in array */
            struct wait_queue *eventn;
            struct wait_queue *eventz;
            struct sem_undo  *undo;            /* undo requests on this array */
            ushort        sem_nsems;           /* no. of semaphores in array */
     };
```

As with message queues, operations on this structure are performed by a special system call, and should not be tinkered with directly. Here are descriptions of the more pertinent fields:

sem_perm

This is an instance of the ipc_perm structure, which is defined for us in linux/ipc.h. This holds the permission information for the semaphore set, including the access permissions, and information about the creator of the set (uid, etc).

sem_otime

Time of the last semop() operation (more on this in a moment)

sem_ctime

Time of the last change to this structure (mode change, etc)

sem_base

Pointer to the first semaphore in the array (see next structure)

sem_undo

Number of undo requests in this array (more on this in a moment)

sem_nsems

Number of semaphores in the semaphore set (the array)

Each semaphore in a semaphore set has the following associated values in sem structure:

```
Struct sem {
        ushort semval;        // Semaphore value, non negative
        short semid;          // id of the process that did last operation on the semaphore
                              value
        ushort semncnt;               // A count, that the number of processes are waiting
                                      to increment the semaphore value
        ushort semzcnt;               // A count, that the number of processes are waiting
                                      for semaphore value to become zero.
};
```

System Calls related to Semaphores

Three major system calls are used by developers to create and manage Semaphores between two or more processes. To use those system calls the required header files are:

Getting set of Semaphores:

Summary of `semget()` system call

Include Files	`<sys/types.h>`
	`<sys/ipc.h>`
	`<sys/sem.h>`
		Manual section	2

Summary	**`int semget (key_t key, int nsems, int semflg);`**		
Return	Success	Failure	
	Semaphore identifier	-1	

To get one set of Semaphores, we have to use the semget() system call. This function declaration is given below.

The semget() system call returns the semaphore set identifier associated with the argument **key**. A new set of **nsems** semaphores is created if key has the value IPC_PRIVATE or if no existing semaphore set is associated with key and IPC_CREAT is specified in **semflg**.

When creating a new semaphore set, semget() initializes the set associated data structure, semid_ds as follows:

- **sem_perm.cuid** and sem_perm.uid are set to the effective user ID of the calling process.
- **sem_perm.cgid** and sem_perm.gid are set to the effective group ID of the calling process.
- The least significant 9 bits of **sem_perm.mode** are set to the least significant 9 bits of semflg.
- **sem_nsems** is set to the value of nsems.
- **sem_otime** is set to 0.
- **sem_ctime** is set to the current time.

The argument nsems can be 0, when a semaphore set is not being created. Otherwise nsems must be greater than 0 and less than or equal to the maximum number of semaphores per semaphore set. If the semaphore set already exists, the permissions are verified.

RETURN VALUE:

If successful, the return value will be the semaphore set identifier, otherwise -1 is returned, with errno indicating the error.

EX:- /************** WAP to create a semaphore set with 5 semaphores **************/

```
Header files
int main() {
        int semid;
        semid = semget(222, 5, IPC_CREAT | 0777);
        if(semid == -1) {
                printf("error in creating semaphore set\n"); exit(0);
        }
        printf("Semaphore set created successfully\n");
        printf("key value: 222, and id: %d\n", semid);
        return 0;
}
```

Working with Semaphores:

Working with semaphores including two operations semaphore lock and semaphore unlock. These two operations can be done by using only one system call semop(). This function declaration is given below:

Summary of `semop()` system call

Include Files	`<sys/types.h>` `<sys/ipc.h>` `<sys/sem.h>`	Manual section	2
Summary	**`Int semop(int semid, struct sembuf *sops, unsigned nsops);`**		
Return	Success	Failure	
	0	-1	

Here:

- semid -> return value of semget()
- *sops -> pointer to sembuf structure
- struct sembuf {

```
        ushort sem_num;     // semaphore number
        short sem_op;       // semaphore operation
        short sem_flg;      // operation flag
  };
```

- nops -> number of operations

semop() performs operations on selected semaphores in the set indicated by semid. Each of the **nsops** elements in the array pointed to by **sops** specifies an operation to be performed on a single semaphore. The elements of this structure are of type struct sembuf, containing the following members:

```
unsigned short sem_num;     /* semaphore number */
short     sem_op;           /* semaphore operation */
short     sem_flg;          /* operation flags */
```

Flags recognized in sem_flg are IPC_NOWAIT and SEM_UNDO. If an operation specifies SEM_UNDO, it will be automatically undone when the process terminates.

Ex:- Write two different functions to show lock and unlock operation on Semaphores.

Declarations and definitions:

```
Static struct sembuf op_lock[2];
op_lock [2] = {
            {0, 0, 0}, // Wait for semaphore zero to become 0.
            {0, 1, 0} // Increment semaphore zero by 1.
          };
Static struct sembuf op_unlock[1];
op_unlock[1] = { 0, -1, IPC_NOWAIT };
```

Lock Function:

```
    my_lock() {
        if ( (semop( semid, &op_lock[0], 2)) < 0)
            perror( “\n locking error \n”);
    }
```

Unlock Function:

```
    my_unlock() {
        if ( semop( semid, &op_unlock[0], 1)) < 0)
            perror (“\n unlock error \n”);
    }
```

Ex:- /****** WAP to show lock and unlock operation on one semaphore ********/

```
Header files
void my_lock();
void my_unlock();
Int main() {
    // write above sembuf declarations and definitions here
    Int semid, var;
    semid = semget(222, 1, IPC_CREAT | 0777); // remove IPC_CREAT in p2.c
    if(semid == -1) {
        printf(“error in creating semaphore set\n”); exit(0);
    }
    printf(“Semaphore set created successfully\n”);
    printf(“key value: 222, and id: %d\n”, semid);
    my_lock();
    printf(“enter the value of var: ”); scanf(“%d, &var”);
    printf(“\nvar = %d\n”, var);
    my_unlock();
    return 0;
}
```

Important Notes:

- By executing the above program, you will not see the lock and unlock operation on semaphore. To analyse clearly how semaphores are useful to synchronize the two or more processes, we have to create two processes which are doing lock and unlock operations on same semaphore.
- Save above program as p1.c and copy the same contents in another program called p2.c
- Create two separate executable codes as server and client using below commands:
  - gcc –o server p1.c
  - gcc –o client p2.c
- Open another terminal and go the directory where you are working
- Then in terminal-1 execute server as ./server it will execute the highlighted printf statement and waits for input
- But don't enter the value now, then go to terminal-2 and execute client as ./client. Here it will not execute the highlighted printf statement. Because it is waiting for semaphore to lock.
- Now enter the value in terminal-1 and monitor the terminal-2 window.
- Again execute server in terminal-1 before entering the value in terminal-2, then see the difference.

Controlling the semaphore set:

Controlling Semaphores can be done by using semctl() function. Here using this system call you can get the semid_ds structure also. This system call declaration is given below.

Summary of `semctl()` system call

Include Files	`<sys/types.h>` `<sys/ipc.h>` `<sys/sem.h>`	Manual section	2
Summary	**`int semctl (int semid, int semnum, int cmd,union semun arg);`**		
Return	Success	Failure	
	0 or value requested	-1	

Here:

- semid -> return value of semget()
- cmd -> IPC_RMID, IPC_STAT, IPC_SET etc
- arg -> Zero if no arguments

Note: - Usage of semctl() system call is same as msgctl() and shmctl() functions.

Important note:- Creating multiple message Queues or Shared memory segments or Semaphores with same key value is not allowed in Linux. To avoid such error make a habit of deleting after use.

Commands:

- To check how many ipc's are created in system use **ipcs** command
- To delete particular ipc from system use **ipcrm** command.
- Read manual pages of these two commands before using.

Key learning:

1. Explain briefly about shared memory?
2. What are the advantages of shared memory?
3. List the header files associated with shared memory?
4. List the system calls associated with shared memory?
5. WAP to get and set Shared Memory attributes.