MODULE 15 :
Inter Process communication

IPC

- Linux supports three types of interprocess communication mechanisms which first appeared in Unix System V.

  1. Message queues,
  2. Shared memory and
  3. Semaphores.

- These System V IPC mechanisms all share common authentication methods.

- Each IPC object has a unique IPC identifier associated with it.

- When we say ``IPC object'', we are speaking of a single message queue, semaphore set, or shared memory segment.

- This identifier is used within the kernel to uniquely identify an IPC object.

IPC

- Processes may access these resources only by passing a unique reference identifier to the kernel via system calls.

- Access to these System V IPC objects is checked using access permissions, much like accesses to files are checked.

- The access rights to the System V IPC object is set by the creator of the object via system calls.

- All Linux data structures representing System V IPC objects in the system include an ipc_perm structure which contains the owner and creator processes user and group identifiers, the access mode for this object (owner, group and other) and the IPC object's key.

IPC

- To obtain a unique ID, a key must be used. The key must be mutually agreed upon by both client and server processes.
- Two sets of key are supported:

  Public key

  Private key.
- **Private Key** means that it may be accessed only by the process that created it, or by child processes of this process.
- **Public Key** means that it may be potentially accessed by any process in the system, except when access permission modes state otherwise.

The ipcs Command

- The ipcs command can be used to obtain the status of all IPC objects.

- ipcs

Shared memory

Key id owner perm size nattch status

Semaphore

Key id owner perm size nattch status

Message queue

Key id owner perm no.messg status

The ipcs Command

Command	Option	Action
ipcs		to check usage of all IPC's
ipcs	-q:	Show only message queues
ipcs	-s:	Show only semaphores
ipcs	-m:	Show only shared memory
ipcs	--help:	Additional arguments

The ipcs Command

- The 'ipcrm' command is used to delete message queue or shared memory or semaphore.

- ipcrm –q ID

  This command takes Message Queue ID as argument and deletes message Queue

- ipcrm –m ID

  This command takes Shared memory ID as argument and deletes Shared memory

- ipcrm –s ID

  This command takes Semaphore ID as argument and deletes Semaphore

  We need to have the proper permissions in order to delete a resource

The ipcs Command

•The 'ipcrm' command is used to delete message queue or shared memory or semaphore.

•ipcrm –Q KEY

This command takes Message Queue Key as argument and deletes message Queue

•ipcrm –M KEY

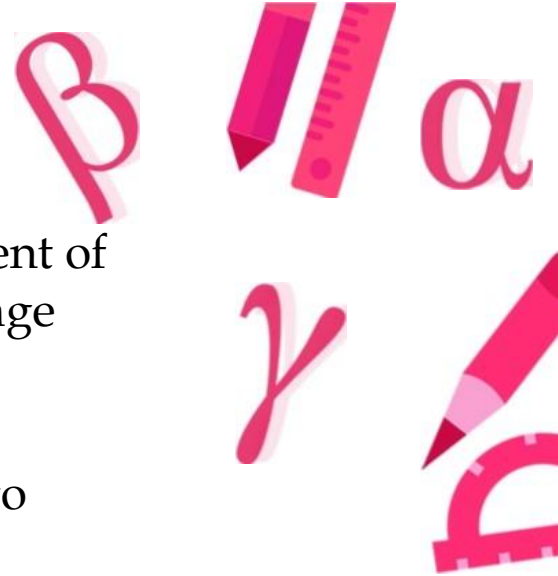
This command takes Shared memory Key as argument and deletes Shared memory

•ipcrm –S KEY

This command takes Semaphore Key as argument and deletes Semaphore

We need to have the proper permissions in order to delete a resource

Message Queues



- Linux maintains a list of message queues, the msgque vector; each element of which points to msqid_ds data structure which fully describes the message queue.

- One of the problems with pipes is that it is up to you, as a programmer, to establish the protocol.

- With a stream taken from a pipe, it means you have to somehow parse the bytes, and separate them to packets.

- Other problem is that data sent via pipes arrives in FIFO order.

- A message queue is a queue onto which messages can be placed.

- Message queues allow one or more processes to write messages which will be read by one or more reading processes.

- Each message queue (of course) is uniquely identified by an IPC identifier.

Message Queues

- When message queues are created a new msqid_ds data structure is allocate from system memory and inserted into the vector.
- Each msqid_ds data structure contains an ipc_perm data structure and pointers to the messages entered onto this queue.
- A message is composed of a message type (which is a number), and message data.
- A message queue can be either private, or public.
- If it is private, it can be accessed only by its creating process or child processes of that creator.
- If it's public, it can be accessed by any process that knows the queue's key.
- Several processes may write messages onto a message queue, or read messages from the queue.
- Messages may be read by type, and thus not have to be read in a FIFO order as is the case with pipes.

Steps to create a message queue and writing into it.

- **Step1 :** Generate a key.
- **Step2 :** Create a message queue using msgget() function and get the message queue ID.
- **Step3 :** Write a message and message type into the message queue using msgsnd() function.

Message to be sent and message type should be defined in structure and address of the structure should be passed as arguments.

Steps to read a message from message queue and delete message queue

- Step1 : Generate a key.
- Step2 : Get the message queue ID.
- Step3 : read message and message type from the message queue using msgrcv() function. message and message type read into the structure.
- Step4 : message queue can be deleted using msgctl() function.

Generating a Key

•Key can be generated using ftok function.

key_t ftok(char *str ,char constant);

where,

char *str – file name.

char constant - a character constant.

return value of ftok is a key of type key_t

Example:

key_t key;

key=ftok(“file1” , ’s’);

Creating A Message Queue - msgget()

- The msgget() system call is used to create message queue.

int msgget (key_t key, int msgflg);

where,

key_t key – return value of ftok to create publicly- accessible message queue.

IPC_PRIVATE used to create a private message queue.

int msgflg – IPC_CREAT either returns the message queue identifier for a newly created message queue, or returns the identifier for a queue which exists with the same key value.

IPC_EXCL is used along with IPC_CREAT, then either a new queue is created, or if the queue exists, the call fails with -1.

Creating A Message Queue - msgget()

An optional octal mode may be OR'd into the mask, since each IPC object has permissions that are similar in functionality to file permissions on a UNIX file system

RETURNS : message queue identifier on success -1 on error:

Example;

```
int id;
id=msgget(key,IPC_CREAT|0766);
```

or

```
id= msgget(key,IPC_CREAT|0766);
```

Writing Messages Onto A Queue - msgsnd()

•Once we have the queue identifier, we can begin performing operations on it.

•To deliver a message to a queue, you use the msgsnd system call:

```
int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg );
```

where,

int msqid : message queue identifier, returned by msgget.

struct msgbuf : this is a pointer to our redeclared and loaded message buffer structure.

```
struct msgbuf
{
long mtype; /* message type, positive number (cannot be 0 ). */
char mtext[1]; /* message body array. larger than one byte. */
};
```

Writing Messages Onto A Queue - msgsnd()

int msgsz - this contains the size of the message in bytes, excluding the length of the message type (4 byte long).

msgflg 0 - the calling process will suspend (block) until the message can be written

IPC_NOWAIT - If the message queue is full, then the message is not written to the queue, and control is returned to the calling process.

RETURNS : 0 on success -1 on error.

Writing Messages Onto A Queue - msgsnd()

Example:

```
struct msgbuf
{
long mtype;
char mtext[20];
};
struct msgbuf s1;
s1.mtyp=20;
strcpy(s1.mtext,"hello world");
msgsnd ( id, &s1, sizeof(s1),0 );
```

Reading A Message From The Queue - msgrcv()

To read a message from message queue, msgrcv system call is used.

int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);

Where

int msqid - id of the queue, as returned from msgget().

struct msgbuf* msg - a pointer to a pre-allocated msgbuf structure. It should generally be large enough to contain a message with some arbitrary data

int msgsz - size of largest message text we wish to receive. Must NOT be larger than the amount of space we allocated for the message text in 'msg'.

Reading A Message From The Queue - msgrcv()

int msgtyp - Type of message we wish to read. may be one of:

0 - The first message on the queue will be returned.

a positive integer - the first message on the queue whose type (mtype) equals this integer (unless a certain flag is set in msgflg, see below).

a negative integer - the first message on the queue whose type is less than or equal to the absolute value of this integer.

Reading A Message From The Queue - msgrcv()

int msgflg - a logical 'or' combination of any of the following flags:

0 - the calling process will suspend (block) until the message of given message type is available

IPC_NOWAIT - if there is no message on the queue matching what we want to read, return '-1',

MSG_NOERROR - If a message with a text part larger than 'msgsz' matches what we want to read, then truncate the text when copying the message to our msgbuf structure. If this flag is not set and the message text is too large, the system call returns '-1'.

Reading A Message From The Queue - msgrcv()

Example:

```
struct msgbuf
{
long mtype;
char mtext[20];
};
struct msgbuf s1;

msgrcv ( id, &s1, sizeof(s1),20,0);
printf("read messagefrom message queue is %s\n",s1.mtext);
```

msgctl()

•To perform control operations on a message queue, you use the msgctl() system call.

int msgctl (int msgqid, int cmd, struct msqid_ds *buf);

Where

int msqid - id of the queue, as returned from msgget().

int cmd - process the following command

IPC_STAT

Retrieves the msqid_ds structure for a queue, and stores it in the address of the buf argument.

IPC_RMID

Removes the queue from the kernel.

msgctl()

- To delete a message queue from the kernel using msgctl function is
  msgctl (id,IPC_RMID,NULL);

- To get a information about message queue structure msgctl function is
  struct msgqid_ds mq1;
  msgctl(id,IPC_STAT,&mq1);

Example : message queue.

Two programs are written,
First program creates a message queue and writes a message into the message queue.
Second program reads the message from message queue.
mq1.c

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/mqueue.h>
#include<string.h>
struct msgbuf
{
long mtype;
char mtext[20]
}
```

Example : message queue.

```
int main()
{
key_t key;
int md_id;
struct msgbuf buf;
key=ftok(“abc”,’x’);
mq_id=msgget(key,IPC_CREAT|0777);
buf.mtype=20;
strcpy(buf.mtext,”hello world”);
msgsnd(mq_id,&buf,sizeof(buf),0);
Printf(“message sent to the message queue\n”);
}
```

Example: message queue

mq2.c

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/mqueue.h>
#include<string.h>
struct msgbuf
{
long mtype;
char mtext[20]
}
```

Example: message queue

```
int main()
{
key_t key;
int md_id;
struct msgbuf buf;
key=ftok("abc",'x');
mq_id=msgget(key,0);
msgrcv(mq_id,&buf,sizeof(buf),0);
printf("message read from message queue\n");
printf("read message is %s\n",buf.mtext);
msgctl(mq_id,IPC_RMID,0);
}
```

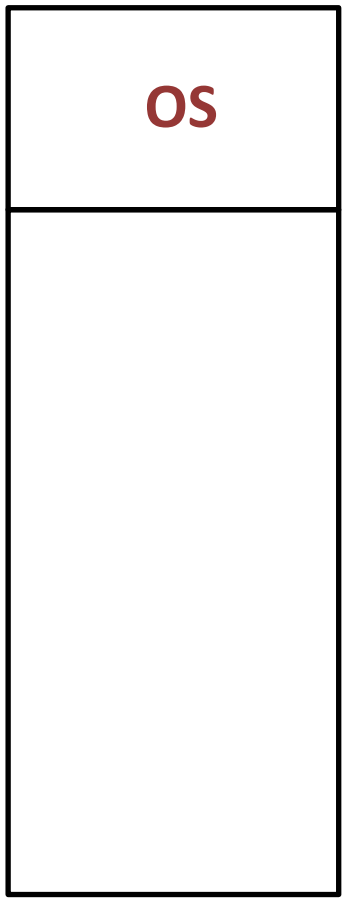
Message queue - limitations

- Message queues are effective if a small amount of data is transferred.
- Very expensive for large transfers.
- During message sending and receiving, the message is copied from user buffer into kernel buffer and vice versa
- So each message transfer involves two data copy operations, which results in poor performance of a system.
- A message in a queue can not be reused

SHARED MEMORY

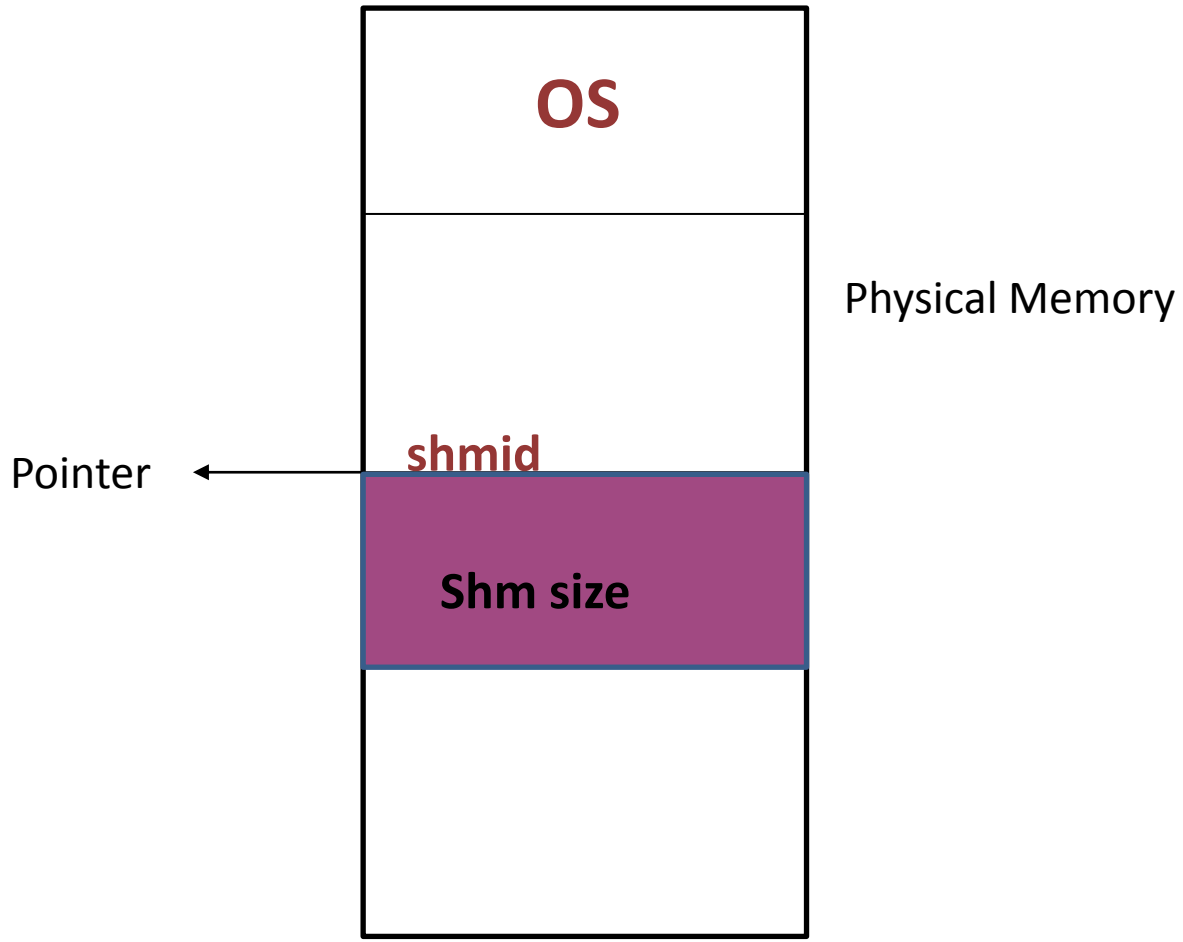
shmid

Shm size



Physical Memory

Shared Memory: shmat ( )



Steps to Access Shared Memory

The steps involved are:

Creating shared memory
Connecting to the memory & obtaining a pointer to the memory
Reading/Writing & changing access mode to the memory
Detaching from memory
Deleting the shared segment

Shmget()

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

shmget system call is used to create a shared
memory segment.
where
Key : it is the return value of ftok function.
Size : size of the shared memory to be created in bytes.
Shmflg : IPC_CRAET to create the shared memory.
0 when shared memory is already created.

Shmget()

- On success the shmget returns the shared memory ID or else it returns -1.

Example:

To create a shared memory of size 2kbyte the shmget system call will be

```
Shm_id=shmget(key,2048,IPC_CREAT|0777);
```

Shmat()

void *shmat(int shmid,void *shmaddr,int shmflg)

Used to attach the created shared memory segment

onto a process address space.

Where,

shmid : shred memory ID.

it is the return value of shmget()

shmaddr : 0 makes shared memory non swappable area

shmflg : 0 for this process can both read from shared memory and write into the shared memory

SHM_RDONLY makes shared memory read only for this process.

SHM_WRONLY makes shared memory write only for this process.

Shmat()

A pointer is returned on the successful execution of the system call and the process can read or write to the segment using the pointer.

Example:

To get the pointer to a shared memory and to use the shared memory for read and write, the shmget syatem call is

P=shmat(shm_id,0,0);

Where p is pointer

Reading / writing to Shared memory

Reading or writing to a shared memory is the easiest part.

The data is written on to the shared memory as we do it with normal memory using the pointers.

Shmdt()

int shmdt(void *p);

Shmdt is used to detach the pointer(process) from shared memory.

Where,

P : it is the pointer to the shared memory.

if shmat is called then number of attachment is increased and if shmdt is called number of attachment is decreased.

Shmctl()

int shmctl(shmid,command,struct);

It is used to remove shared memory object and can also be used to get the information about shared memory.
Where.

shmid : shared memory ID, return value of shmget.

command : IPC_STAT -copies the shared memory segment information into the buffer argument.

IPC_SET - sets the user given values into the system's shmid_ds structure.

IPC_RMID - remove the shared memory segment.

struct : address of structure of type shmid_ds

Example : Shared Memory

Two programs are written,
First program creates a shared memory of size 1024 bytes and writes a data into the shared memory.
Second program reads the data from shared memory.

shm1.c

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

Example : Shared Memory

```
int main()
{
key_t key;
int shm_id;
char *p;
key=ftok(“abc”,’x’);
shm_id=shmget(key,1024,IPC_CREAT|0777);
p=(char *)shmat(shm_id,0,0);
strcpy(p, ”india is great”);
shmdt(p);
Printf(“data is written in shared memory\n”);
}
```

Example : Shared Memory

shm1.c

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<string.h>
int main()
{  key_t key; int shm_id; char *p;
key=ftok("abc",'x');
shm_id=shmget(key,0,0);
p=(char *)shmat(shm_id,0,0);
Printf("data read from shared memory is %s\n",p);
shmdt(p);
shmctl(shm_id,IPC_RMID,NULL);
}
```

Limitation

•Data can either be read or written only. Append is not allowed.

•Race condition

- Since many processes can access the shared memory, any modification done by one process in the address space is visible to all other processes. Since the address space is a shared resource, the developer should implement a proper locking mechanism to prevent the race condition in the shared memory

Thank You!