System V IPC

- Linux supports three types of interprocess communication mechanisms which first appeared in Unix System V.
- These are message queues, semaphores and shared memory.
- 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.

System V 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.

System V 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 and private.
- Private means that it may be accessed only by the process that created it, or by child processes of this process.
- Public 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 System V IPC objects.

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

- The 'ipcrm' command accepts a resource type ('shm', 'msg' or 'sem') and a resource ID, and removes the given resource from the system.

```
ipcrm <msg | sem | shm> <IPC ID>
```

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

Message Queues

- 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.
- 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.

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.

Creating A Message Queue - msgget()

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

  ```
  int msgget ( key_t key, int msgflg );
  ```

- RETURNS: message queue identifier on success -1 on error:
- This system call accepts two parameters - a queue key, and flags.
- The key may be one of:
  - IPC_PRIVATE - used to create a private message queue.
  - a positive integer - used to create (or access) a publicly-accessible message queue.

Creating A Message Queue - msgget()

- The second parameter contains flags that control how the system call is to be processed.
- **IPC_CREAT**
  - Create the queue if it doesn't already exist in the kernel.
- If IPC_CREAT is used alone, msgget() 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.

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 );
```

  The first argument to msgsnd is our queue identifier, returned by a previous call to msgget.
- The second argument, msgp, is a pointer to our redeclared and loaded message buffer.

```
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()

- The msgsz argument contains the size of the message in bytes, excluding the length of the message type (4 byte long).
- The msgflg argument can be set to 0 (ignored), or:
- **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.
  - If not specified, then the calling process will suspend (block) until the message can be written.
- RETURNS: 0 on success -1 on error.

Reading A Message From The Queue - msgrcv()

- int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);
- This system call accepts the following list of parameters:
  - <u>int msqid</u> - id of the queue, as returned from msgget().
  - <u>struct msgbuf* msg</u> - a pointer to a pre-allocated msgbuf structure. It should generally be large enough to contain a message with some arbitrary data (see more below).
  - <u>int msgsz</u> - 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'.
  - <u>int msgtyp</u> - Type of message we wish to read. may be one of:
    - <u>0</u> - The first message on the queue will be returned.

Reading A Message From The Queue - msgrcv()

    - 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.

- int msgflg - a logical 'or' combination of many flags:
    - IPC_NOWAIT - if there is no message on the queue matching what we want to read, return '-1',

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);

Parameters:

- 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.

Shared memory

- Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process.
- This is by far the fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue, etc).
- Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process.
- A segment can be created by one process, and subsequently written to and read from by any number of processes.
- As with message queues and semaphore sets, the kernel maintains a special internal data structure for each shared memory segment which exists within its addressing space. This structure is of type shmid_ds, and is defined in linux/shm.h

SYSTEM CALL: shmget()

- In order to create a new memory segment, or access an existing segment, the shmget() system call is used.

  PROTOTYPE: int shmget (key_t key, int size, int shmflg);
- The first argument to shmget() is the key value.
- This key value is then compared to existing key values that exist within the kernel for other shared memory segments.
- At that point, the open or access operation is dependent upon the contents of the shmflg argument.

- **IPC_CREAT**
  - Create the segment if it doesn't already exist in the kernel.
- **IPC_EXCL**
  - When used with IPC_CREAT, fail if segment already exists.

  RETURNS:
  - shared memory segment identifier on success
  - -1 on error: errno = EINVAL (Invalid segment size specified)

SYSTEM CALL: shmat()

- Once a process has a valid IPC identifier for a given segment, the next step is for the process to attach or map the segment into its own addressing space.

  int shmat (int shmid, char *shmaddr, int shmflg);

- RETURNS:
  - address at which segment was attached to the process or
  - -1 on error.
- If the addr argument is zero (0), the kernel tries to find an unmapped region. This is the recommended method.
- An address can be specified to resolve conflicts with other apps.
- The SHM_RND flag can be OR'd into the flag argument to force a passed address to be page aligned (rounds down to the nearest page size).

SYSTEM CALL: shmat()

- In addition, if the SHM_RDONLY flag is OR'd in with the flag argument, then the shared memory segment will be mapped in, but marked as readonly.
- Once a segment has been properly attached, and a process has a pointer to the start of that segment, reading and writing to the segment become as easy as simply referencing or dereferencing the pointer.
- Be careful not to lose the value of the original pointer, If this happens, you will have no way of accessing the base (start) of the segment.

SYSTEM CALL: shmctl()

int shmctl (int shmqid, int cmd, struct shmid_ds *buf);

- RETURNS:
  - 0 on success
  - -1 on error
- This particular call is modeled directly after the *msgctl* call for message queues. In light of this fact, it won't be discussed in too much detail. Valid command values are:
- **IPC_STAT**
  - Retrieves the shmid_ds structure for a segment, and stores it in the address of the buf argument
- **IPC_RMID**
  - Marks a segment for removal.
- The IPC_RMID command doesn't actually remove a segment from the kernel. Rather, it marks the segment for removal. The actual removal itself occurs when the last process currently attached to the segment has properly detached it. Of course, if no processes are currently attached to the segment, the removal seems immediate.

SYSTEM CALL: shmdt()

- To properly detach a shared memory segment, a process calls the *shmdt* system call.

  int shmdt (char *shmaddr);
- RETURNS:
  - 0 on success
  - -1 on error

EXAMPLE

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

int main()
    {
    int x,y,z;
    char *a; x=shmget(0x30,1024,
    IPC_CREAT|0666);
    a=shmat(x,NULL,0);
    strcpy(a,"hello");
    sleep(5);
    write(1,8,a);
    shmdt(a);
    shmctl(x,IPC_RMID);
    }
```

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

int main()
    {
    int x,y,z;
    char *a;
    x=shmget(0x30,1024,IPC_CREA
    T|0666);
    a=shmat(x,NULL,0);
    sleep(5);
    printf("\n%s\n",a);
    strcpy(a+6,"bye");
    shmdt(a);
    }
```