Unix System V Inter Process Communication

Timeline of UNIX:

1965 : AT&T Bell Lab, GE & Project MAC of MIT develop a new operating system called Multics.

1969 : Due to failure of achievement as the goal & due to over-budget, Bell Lab withdrew its participation. But some of the members of the above project from the Bell like Ken Thompson & Dennis Ritchie took the idea Multics and did a paper design of new operating system.

1970 : They implemented this design on DEC PDP-7 machine and was named as UNICS ,Where “MULTI” is replaced by “UNI” by another member Brian Kernighan.

1971 : UNIX was tried on more powerful machine PDP-11.

1972 : Dennis Ritchie developed ‘C’ language

1973 : The whole UNIX operating system was re-written in C

1977 : UNIX was first ported on Non-PDP machine called Interdata 8/32. Realising its commercial value, AT&T released UNIX commercially in the world with its source code.

1977 to 1982 : AT&T itself created UNIX System III, UNIX System IV

1983 : The UNIX System V. This last version i.e. System V was sold commercially and become the most popular one.

About Unix System V:

UNIX System V is one of the first commercial versions of the Unix operating system.
It was originally developed by AT&T and first released in 1983.
Four major versions of System V were released, numbered 1, 2, 3, and 4.
System V Release 4, or SVR4, was commercially the most successful version.

SVR1 : Realised in 1983
It added support for inter-process communication using messages, semaphores, and shared memory, developed earlier for the Bell-internal CB UNIX.

SVR2 : Realised in 1984
It added demand paging, copy-on-write, shared memory, and record and file locking.
Maurice J. Bach's book, The Design of the UNIX Operating System, is the definitive description of the SVR2 kernel.

SVR3 : Released in 1987
SVR3 included STREAMS, Remote File Sharing (RFS), the File System Switch (FSS) virtual file system mechanism, a restricted form of shared libraries, and the Transport Layer Interface (TLI) network API.

SVR4 : Released in 1988

SVR4 release of Unix gives multiple things to technical world as:

- From BSD: TCP/IP support, sockets, UFS, support for multiple groups, C shell.
- From SunOS: the Virtual File System interface new virtual memory system including support for memory mapped files
- Better support for Inter Process Communication (IPC).
- Korn shell (Type of shell)
- An application binary interface (ABI) based on Executable and Linkable Format (ELF).
- Support for standards such as POSIX and X/Open.

About the Process :

- A process is an instance of an executing program.
- A program is a file containing a range of information that describes how to con- struct a process at run time.
- A program is an executable file and a process is an instance of the "program in execution". Many processes can run simultaneously on UNIX system. This feature is called as Multi-tasking or Multi-programming. Also many instances of one program also can run simultaneously on UNIX system. There are system calls to create a new process, terminate an old process.
- Note that different processes or different instances of same process get execute independently of each other.
- The major 4 system calls for process management are fork ( ), exec ( ), wait ( ) and exit ( ).
    1. fork ( ) is used to create a new process
    2. exec ( ) is used to execute the new process
    3. wait ( ) is used to allow the old process to wait for the completion of the execution of new process
    4. exit ( ) is used for exiting the program.

- As stated before, process is the execution state of the program. Many processes may execute simultaneously in UNIX like multi-processing system. And also there may be many processes of a single program. For every process separate memory gets allocated by the kernel which is called as Process Address space.
- Every process can access the data from its address space only due to the protected mode operating system.
- A process can read its data and Stack section in memory, but can not read or write to another process's data and Stack section.
- Processes can communicate with each other by using system calls, this method of communication is called as Inter-Process Communication.
- IPC mechanism allows arbitrary processes to exchange data and synchronize execution.
- Every multitasking OS provides a set of facilities that allows processes to communication with each other.
- Information is exchanged between the OS and the process and/or between one running process with the another.

Inter Process Communication :

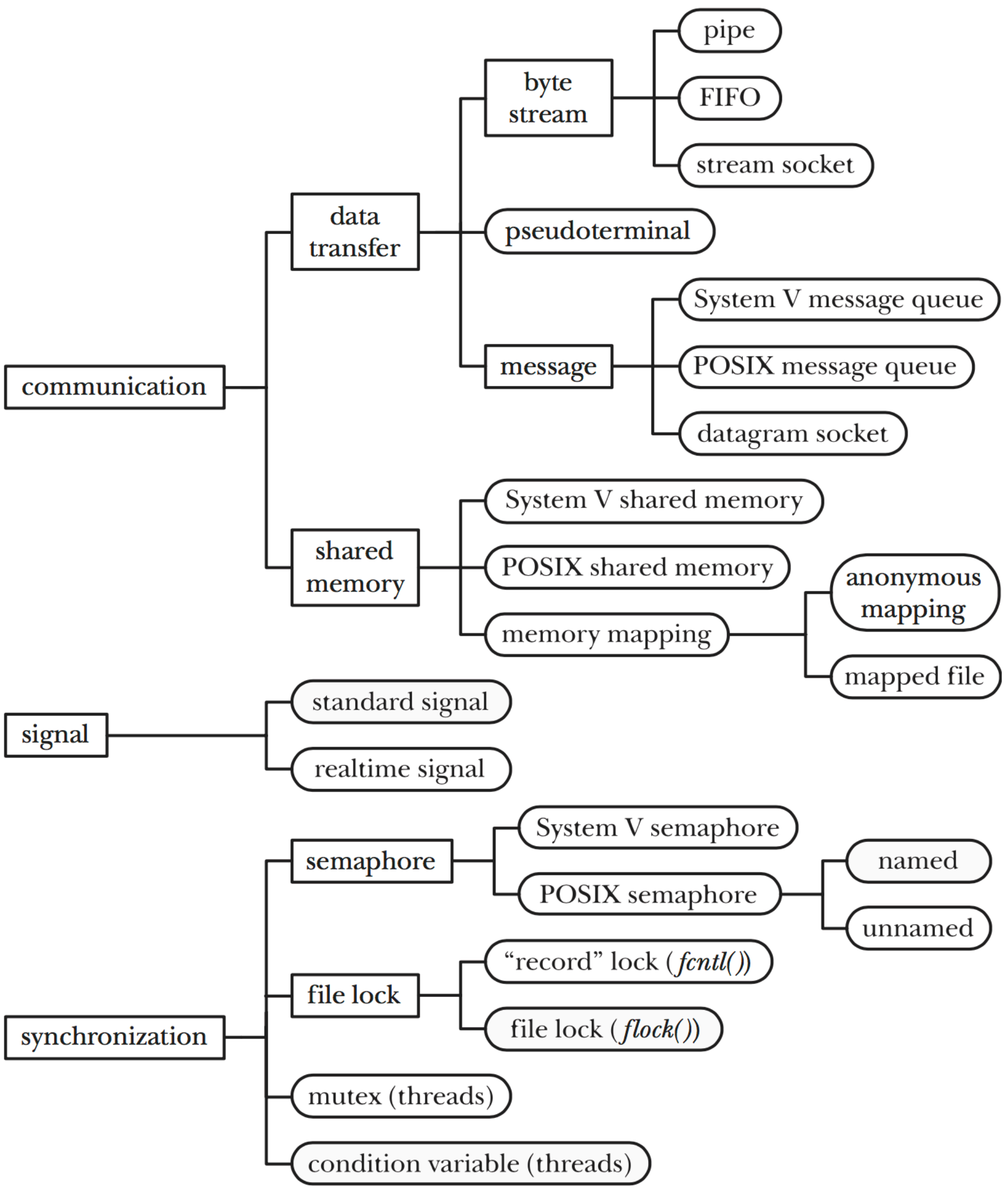
IPC mechanism allows arbitrary processes to exchange data and synchronize execution.

- Every multitasking OS provides a set of facilities that allows processes to communication with each other.
- Information is exchanged between the OS and the process and/or between one running process with the another.

There are several forms of IPC as:

• Pipes	
• Named pipes	(FIFO)
• Signals	
• Message queue	(System V IPC)
• Shared memory	(System V IPC)
• Semaphores	(System V IPC)
• Sockets	(Free BSD)

Classification of IPC mechanisms



About Unix System V Inter Process Communication:

- These System V IPC mechanisms all share common authentication methods. 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.
- The object's reference identifier is used by each mechanism as an index into a table of resources. It is not a straight forward index but requires some manipulation to generate the index.
- Interprocess communication (IPC) includes thread synchronization and data exchange between threads beyond the process boundaries.
- If threads belong to the same process, they execute in the same address space, i.e. they can access global (static) data or heap directly, without the help of the operating system.
- However, if threads belong to different processes, they cannot access each others address spaces without the help of the operating system.

There are two fundamentally different approaches in IPC:

1. Processes are residing on the same computer
2. Processes are residing on different computers.

The first case is easier to implement because processes can share memory either in the user space or in the system space. This is equally true for uniprocessors and multiprocessors.
In the second case the computers do not share physical memory, they are connected via I/O devices(for example serial communication or Ethernet).
Therefore the processes residing in different computers cannot use memory as a means for communication.

Creating and opening a System V IPC object :

Each System V IPC mechanism has an associated get system call (msgget(), semget(), or shmget()), which is analogous to the open() system call used for files.
Given an integer key (analogous to a filename), the get call either:

- Creates a new IPC object with the given key and returns a unique identifier for that object; or
- Returns the identifier of an existing IPC object with the given key.

Generating unique IPC Keys using ftok() function:

System V IPC keys are integer values represented using the data type key_t.
The IPC get calls translate a key into the corresponding integer IPC identifier.
Instead of taking hardcoded key we can use ftok() function to generate unique IPC keys.

The ftok() (file to key) function returns a key value suitable for use in a subsequent call to System V IPC get system calls.

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

key_t ftok(char *pathname, int proj); // Returns integer key on success, or –1 on error

ftok() uses the i-node number rather than the name of the file to generate the unique key value.
The ftok() function uses the identity of the file named by the given *pathname* (which must refer to an existing, accessible file) and the least significant 8 bits of *proj_id* (which must be nonzero) to generate a *key_t* type System V IPC key, suitable for use with msgget, semget, or shmget.

All process that wants to communicate with each other can call ftok() function with same parameters to generate key.

Associated IPC Data Structures:

The kernel maintains an associated data structure for each instance of a System V IPC object.
The form of this data structure varies according to the IPC mechanism (message queue, semaphore, or shared memory) and is defined in the corresponding header file for the IPC mechanism (sys/msg.h or sys/sem.h or sys/shm.h).
The associated data structure for an IPC object is initialized when the object is created via the appropriate get system call (msgget() /shmget() / semget()).

IPC objects Permissions:

All three objects of IPC mechanisms includes a substructure, ipc_perm, that holds information used to determine permissions granted on the object

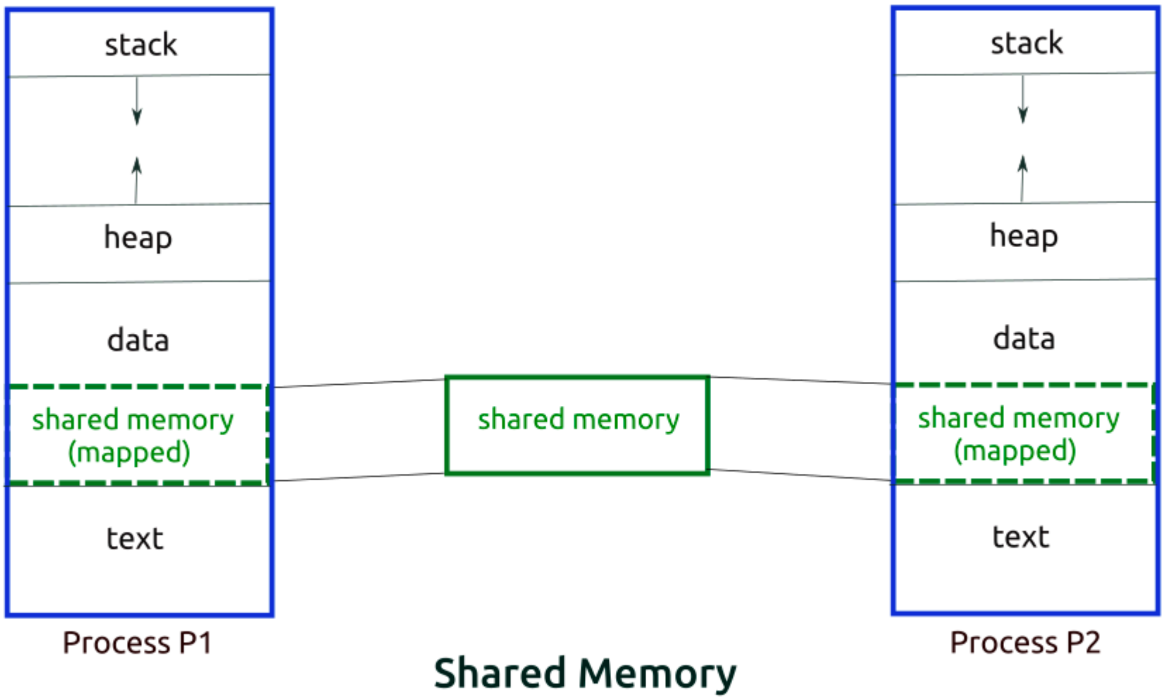
```
struct ipc_perm
{
        key_t __key;              // Key, as supplied to 'get' call
        uid_t uid;                // Owner's user ID
        gid_t gid;                // Owner's group ID
        uid_t cuid;               // Creator's user ID
        gid_t cgid;               // Creator's group ID
        unsigned short mode;      // Permissions
        unsigned short __seq;     // Sequence number
};
```

IPC Client-Server Architecture:

In client-server applications, the server typically creates the System V IPC objects, while the client simply accesses them.
In other words, the server performs an IPC get call specifying the flag IPC_CREAT, while the client omits this flag in its get call.

Shared Memory

- Shared memory allows two or more processes to share the same region (usually referred to as a segment) of physical memory.
- Since a shared memory segment becomes part of a process's user-space memory, no kernel intervention is required for IPC. All that is required is that one process copies data into the shared memory; that data is immediately available to all other processes sharing the same segment.
- This provides fast IPC by comparison with techniques such as pipes or message queues, where the sending process copies data from a buffer in user space into kernel memory and the receiving process copies in the reverse direction



- Shared Memory has certain size and some physical address.
  Processes that wants to communicate with each other attaches this segment
- to their address space. Thus the virtual addresses of a process are now pointing to the SM segment.
- Normally text , data , and stack segments are NOT PERSISTANT in the memory. That is, they do not stay in the memory of no process is accessing them.
- Whereas SM segment are persistent in memory until
  - SM identifier is removed.
  - There are no move processes attached to it.
- - Or system reboots.

  The advantage of using SM for IPC is SPEED. It is very quick to exchange between 2 processes.
  Database systems uses SM to exchange queries and results between the data base client U1 and the data base server.
- Disadvantage is that synchronization is must. I.e. they must ensure the synchronize access to this memory.

Initialization Shared Memory shmget() :

```
shmid = shmget(key , size , flag);
```

The shmget() system call converts this IPC key to SM identifies.
key -> Specifies the IPC key used to identify the SM segment.

size -> Specifies how large the SM segment must be.
If shmget() references an existing SM segment and size specified in the system call must be greater than the existing size, the system call this and returns an error EINVAL.

Flag -> Contains the access mode and mode bits.
Access mode -> Define which processes are allowed. Mode bits -> IPC_PRIVATE and IPC_CREATE.
If the process has neither RD or WR permission, it will not be able to attach SM. If process has RDONLY permission and tries to write it, it will be killed with SIGSEGV.

Each shared memory segment is described by shmid_ds struct...

```
struct shmid_ds
{
        struct ipc_perm shm_perm;
        int shm_segsz;
        struct anon_map *shm_perm;
        ushort_t shm_lkent;
        pid_t shm_lpid;
        pid_t shm_cpid;
        long shm_nattach;
        time_t shm_atime;
        time_t shm_dtime;
        time shm_ctime;
};
```

Members of above structure are :

- shm_perm : IPC permission struct.
- shm_segsz : size of the segment. This can be any value, up to max shared memory segment.
- shm_amp : ptr to a non map for this struct. Ptr to region table entry. Kernel uses anonymous pages for shared memory segment.
- shm_lkent : Count of times this segment is locked.
- shm_lpid : ID of last process to perform shmop.
- shm_cpid : Creator’s process id.
- shm_nattch : Count of proc attached to this process.
- shm_cnattch : Holds the same value as shm_nattch.
- shm_atime : Time last shmat was done.
- shm_dtime : Time last shmdt was done.
- Shm_ctime : Time of last IPC_SET shmctl command.

Attaching the Shared Memory segment shmat():

The kernel searched the shared memory table on given key (i.e. search linked list of shmid_ds).
If it finds the entry and the permission modes are acceptable , then get the id of that entry.
If not, and IPC_CREAT flag is set, then create a new entry in the table.
The kernel verifies the size is between SHMMIN and SHMMAX , then it allocates a region using allocreg(). The fields (first 3) permission modes, size and the ptr to region table entry is set.
It also sets the flag indicating that no memory is allocated for this share region and so this region entry. So the member of region (ptr to page table is still null. It allocates page table only when a process attaches a region to its address space.
The kernel also sets a flag on the region table entry to indicate that region should not be freed even if last process attached to it exists.
Thus data in shared memory remains intact, even if no process include it as a part of virtual address space.
Finally the remaining fields of shmid_ds are initialized.

Vaddr = shmat(id , addr , flags);

Returns address at which shared memory is attached on success, or (void *) –1 on error

Algorithm : shmat

```
input :     (1) Shared memory descriptor.
            (2) Virtual address to attach memory.
            (3) Flags.

    Output : Virtual address where memory was attached.
    {
            check validity of descriptor, permissions;
             if(uses specified virtual address)
            {
                    round off virtual address as specified by the
                    check legality of virtual address, size of region;
            }
            else   //uses want kernel to find good address.
            {
                    kernel picks up virtual address;
                    error if not available;
            }
            attach region to process address space (Algorithm Attachreg)
            if(region being attached for 1st time)
                    allocate page tables and memory for region (Algorithm growreg)
            return(virtual address where attached);
    }
```

Explanation :
id - return by shmget(), identifies the entry in shared memory region table.
Flag : (1) Whether region is read only.
(2) Whether kernel should round off user specified virtual address.
Return virtual address : may not be same as that of requested virtual address.

(iv) Check validity of descriptor and permission. While executing the kernel checks that the process has necessary permissions to access this region, defined in shmid_ds. Also check the validity of descriptor.

Virtual address specified by user. If virtual address is specified by the user, check if(virtual address is already mapped or not in user virtual address space) then return EINVAL error. Else if round off flag is specified then kernel should round virtual address off.

If virtual address = 0 kernel chooses if conveniently. The shared memory must not overlap other regions in ProcVASpace.

e.g. (a) kernel should not attach it close to data region because brk system call.

(b) Also do not put it at the top of the stack.

Best place to put in case of stack, if stack grows upward then (higher addresses) at the start of stack.

Page tables : The kernel does all necessary checking and attaches the region to procedure. Virtual address space using attachreg(). If the process is first to attach this shared memory region, then allocate necessary tables using growreg(). Fill all pages with zeroes.

Time info : Adjust the table entry for shared memory fields shmid_ds for access time information.

Also set the no. of process attached to this shared memory.

Finally return the virtual address.

Detaching a shared memory shmdt():

It used to remove mapping of shared memory with our process.

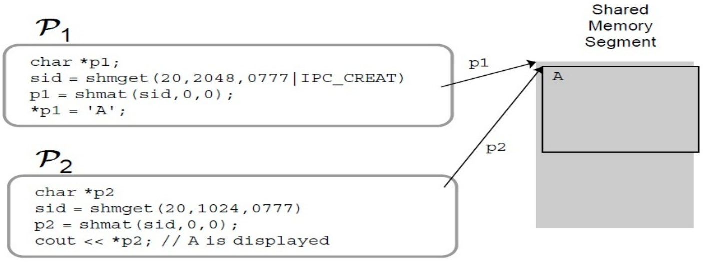
```
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

shmaddr : Virtual address in the process for this shared memory region.

Check the address supplied is the shared memory segment address in that process.

The kernel must keep track of shared memory segments attached to a process so that it can update reference count and use it for such shared memory segment address validation.

Returns 0 on success, or –1 on error



Application Program for shared memory

Server Application

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include<unistd.h>
#include<stdlib.h>

#define SHMSZ 30

int main()
{
	char c;
	int shmid;
	char *shm, *s;

	printf("Demo of IPC using Shared Memory\n");

	key_t key = ftok(".",'a');					// Generate key

	shmid = shmget(key, SHMSZ, IPC_CREAT | 0666);	// Create the segment

	shm = shmat(shmid, NULL, 0);			// Attach segment to our data space

	s = shm;						// base address of shared memory

	for (c = 'a'; c <= 'z'; c++)
	{
		*s = c;
		s++;
	}

	printf("Data is written in Shared Memory\n");

	*s = '\0';

	//Wait until other process changes the first character of our memory to '*' indicating  that it has read what we put there.

	while (*shm != '*')
	{
		sleep(1);
	}

	printf("Data is Successfully fetched by client\n");

	printf("Terminating server\n");
	exit(0);

}
```

Output of above application

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ gcc IPC_SharedMemo
ry_Server.c -o server
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ./server
Demo of IPC using Shared Memory
Data is written in Shared Memory
Data is Successfully fetched by client
Terminating server
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Client Application

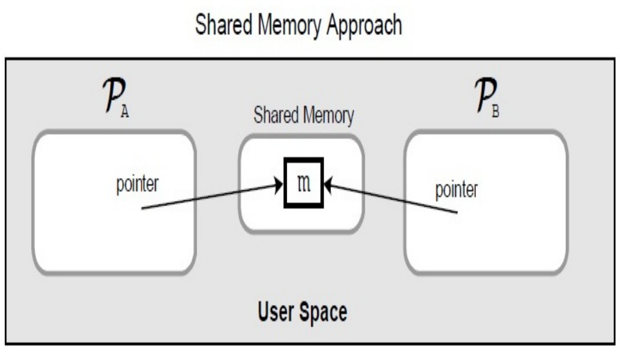
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include<stdlib.h>

#define SHMSZ 30

int main()
{
	int shmid;
	key_t key;
	char *shm, *s;

	printf("Demo of IPC using Shared Memory\n");

	printf("Client is running\n");

	key = ftok(".",'a');

	shmid = shmget(key, SHMSZ, 0666);

	shm = shmat(shmid, NULL, 0);

	printf("Data received from Server\n");

	for (s = shm; *s != '\0'; s++)// Now read what the server put in the memory.
	{
		printf("%c",*s);
	}

	*shm = '*'; // change the first character of the segment to '*', indicating we have
read the segment.

	printf("\nTerminating the Client\n");

	exit(0);
}
```

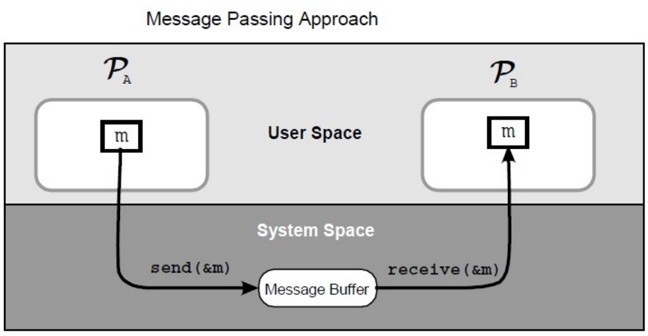
Output of above application

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ gcc IPC_SharedMemo
ry_Client.c -o client
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ./client
Demo of IPC using Shared Memory
Client is running
Data received from Server
abcdefghijklmnopqrstuvwxyz
Terminating the Client
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

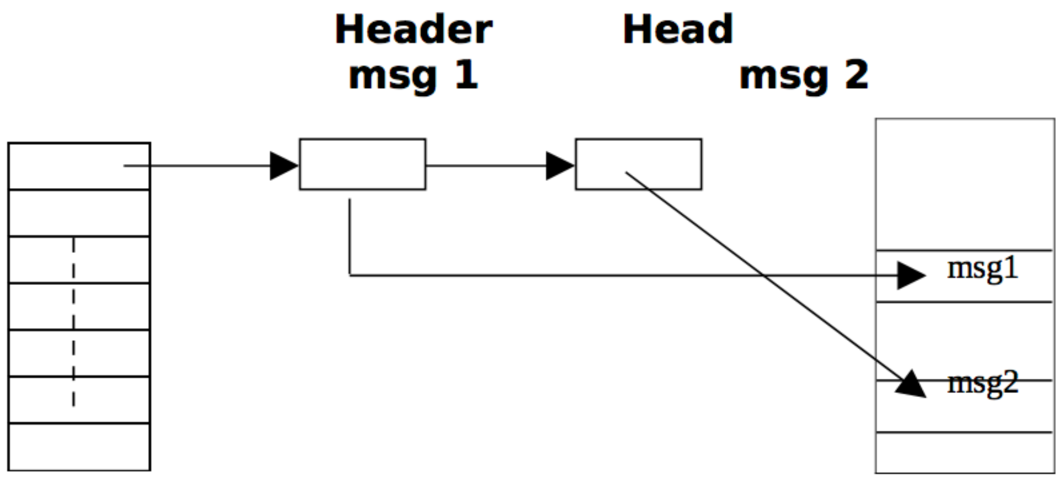
Messages Queues

- Message queues allow processes to exchange data in the form of messages.
- A data structure (message m) is copied from the space of the sender process into a message buffer in system space, and then copied again from the buffer in system space to the structure in space of the receiving process.
- In order to make messages work across the process boundary, the message buffers have to be named: each process which creates a message buffer has to refer to the same message buffer name (kernel will either create a new message object and return a handle, or will just return the handle if the message buffer has already been created by another process.)
- The handles returned by the kernel are local for each process.





- The message transfer is normally synchronized, i.e. the receiving process is blocked if the message buffer is empty, and the sending process is blocked if the message buffer is full. The messages can be of fixed or variable size.
- In the latter case, the message buffer is organized as a queue of message headers, while the space for the actual message is dynamically allocated.
- Messages are less efficient than shared memory (require buffering and synchronization), but sometimes are more suitable due to the built-in synchronization.



Creating or Opening a Message Queue msgget():

The msgget() system call creates a new message queue or obtains the identifier of an existing queue.

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

Returns message queue identifier on success, or –1 on error

- The entry in the message queue is allocated with msgget() system call.
- When user calls msgget() to create a new descriptor – msggid, the kernel reaches msg queue, if one exists in it. If there is no entry for the specific key, the kernel allocates a new queue structure, (header), initializes it and returns an identifier (msggid) to the user.
- Depending on the flag value... IPC_EXEL | IPC_CREAT then if it finds an empty for the key, then msgget() returns an error.
- If calls ipc_get() to allocate msqid_ds from message queue array. The fields of this struct (msg que header) are initialized and identifier is returned.

```
struct msqid_ds
{
        struct ipc_perm msg_perm;
        struct msg * msg_first;
        struct msg * msg_last;
        unsigned long msg_cbytes;
        unsigned long msg_qnum;
        unsigned long msg_qbytes;
        pid_t msg_lspid;
        pid_t msg_lrpid;
        time_t msg_stime;
        time_t msg_rtime;
        time_t msg_ctime;
}
```

Members of above structure:

- msg_perm : IPC permissions.
- msg_first : ptr to the first msg on this queue.
- msg_last : ptr to the last msg on this queue.
- msg_cbytes : no. of bytes in the queue.
- msg_qnum : no. of msgs in the queue.
- msg_qbytes : max no. of bytes that can be queued.
- msg_lspid : pid or proc that performed last msgsnd().
- msg_lrpid : pid of proc that performed msgrcv().
- msg_stime : Time at which last msg was send.
- msg_rtime : Time at which last msg was recv.
- msg_ctime : Time at which msqid_ds was last changed.

```
struct msg
{
        struct msg * msg_next;
        long msg_type;
        unshort msg_ts;
        short msg_spot;
}
```

Members of above structure :

- msg_next : ptr to next msg in the queue.
- msg_type : message type.
- msg_ts : message size.
- msg_spot : message map address.

Exchanging Messages msgsnd() and msgrcv():

The msgsnd() and msgrcv() system calls perform I/O on message queues. The first argument to both system calls (msqid) is a message queue identifier. The second argument, msgp, is a pointer to a programmer-defined structure used to hold the message being sent or received. This structure has the following general form:

```
struct mymsg
{
	long mtype;				// Message type
	char mtext[];				// Message body
}
```

Sending Messages msgsnd():

The msgsnd() system call writes a message to a message queue.

```
	#include <sys/msg.h>
	int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Returns 0 on success, or –1 on error

Algorithm : msgsnd

Input : (1) msg queue descriptor.
(2) addr. of msg struct.
(3) size of msg.
(4) flags.

Output : no. of bytes sent.

```
{
	check legality of descriptor : permissions.
	while(not enough space to store msg)
	{
		if(flags specify not to wait)
			return;
		sleep(until event enough space available);
	}
	get message headers;
	read msg text from user to kernel space; Adjust data structures...
	- en queue message header.
	- msg hdr pts to data.
	Wake up all processes waiting to read message from queue.
}
```

Explanation :

msgsnd(msqid , msg , count , flag)

The parameters in the system call are validated.
Sending process has the write "permission" for the msg descriptor.
msg length does not exceed the system limit. (MSGMAX).
Msg queue does not contain too many bytes.
msg type is "+ve int".
If any of there test fails, the system call returns an error.

The size of the queue is checked. If there is no space on the queue for the message being added, action depends on the flag IPC_NOWAIT.
If this flag is set, the system call returns immediately with the error code EAGAIN. If it is clear process sleeps until there is enough space in the queue for the msg.

The function then attempts to allocate a message header. If there are no message header available, it sleeps or returns depending on IPC_NOWAIT.

Next step is to allocate a buffer from message buffers pool which is large enough to hold space, depending on IPC_NOWAIT, sleep as return.

At this point fun has message data from user space to message buffer. It links the buffer to msg header. Link the header at the end of message queue.

If any processes are sleeping waiting for the data to appears on the queue, those processes are waken up by wakeprocs().

Finally, the last access time and other accounting fields are updated in msquid_ds and system call returns.

Receiving Messages msgrcv():

The msgrcv() system call reads (and removes) a message from a message queue, and copies its contents into the buffer pointed to by msgp.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t maxmsgsz, long msgtyp, int msgflg);
```

Returns number of bytes copied into mtext field, or –1 on error

Algorithm : msgrcv

input : (1) msg descriptor
(2) Addr. of data array for incoming msg.
(3) Size of data array.
(4) flag.

Output : no. of bytes in returned message.

```
{
    check permissions;
    loop :
        check legality of msg descriptor; /* Find msg to return to user */
        if(requested msg type == 0)
```

```
            consider 1st msg in the queue; else
        if(req. msg type > 0)
            consider 1st msg on queue with req = type;
        else /* type < 0*/
            consider lowest msg type that is found first (whose absolute value is less than or equal to requested)

        if(there is a message)
        {
            adjust msg size or error;
            copy msg type and text from kernel to user space;
            unlink msg from queues;
            return;
        }
        /* no message */
        if(flag specify not to sleep)
            return error;
    sleep(event : msg arrives on queue)
    goto loop;
}
```

Explanation :

Validate the parameters in system call. Check if user process (receiving) has RD access permission for msg descriptor. On failure, sys call exits and returns error. If there are no messages to receive, the process sleeps or returns depending on IPC_NOWAIT.

The processing of message queue depends on the type of argument supplied in the system call.

== 0 Selects the 1st message on message queue to read.

> 0 Performs linear search and selects the first msg whose msg type = type in system call.

< 0 Performs the linear search and select the 1st msg whose type is less than equal to the absolute value of type specified in system call. Once again IPC_NOWAIT check.

If msg is found, msg size is checked. If the message data size is greater than that of supplied in system call, return error.

If buffer size <= size in system call, copy the data to user data structure, return the data buffer to data buffer pool.

Also unlink the msg header from linked list and return it ti msg header pool.

Update the message queue header struct.

- declare cnt of msg on the queue.
- declare no. of bytes in the queue.
- Set last recv time and recv pid.

Wake up all the processes that are waiting for getting room on this list.

Message Queue Associated Data Structure

Each message queue has an associated msqid_ds data structure of the following form:

```
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;		// Number of bytes in queue
	msgqnum_t msg_qnum;			// Number of messages in queue
	msglen_t msg_qbytes;			// Maximum bytes in queue
	pid_t msg_lspid;			// PID of last msgsnd()
	pid_t msg_lrpid;			// PID of last msgrcv()
};
```

Application program of Message Queue

Server Application

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/msg.h>
#include<stdlib.h>

#define  MAX_TEXT 512

struct my_msg_st
{
	long int my_msg_type;
	char some_text[MAX_TEXT];
};

int main()
{
	int running = 1, msgid;
	struct my_msg_st some_data;
	char buffer[BUFSIZ];

	printf("Demonstartion of IPC using Message Queue\n");

	msgid = msgget( (key_t)1234, 0666 | IPC_CREAT);
	if (msgid == -1)
	{
		printf("failed to create:\n");
		exit(EXIT_FAILURE);
	}

	printf("Message Queue is created successfully\n");
```

```
	while(running)
	{
		printf("Enter Some message : ");
		fgets(buffer, BUFSIZ, stdin);
		some_data.my_msg_type = 1;
		strcpy(some_data.some_text, buffer);

		if(msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1)
		{
			printf("msgsnd failed\n");
			exit(EXIT_FAILURE);
		}

		if(strncmp(buffer, "end", 3) == 0)
		{
			running = 0;
		}
	}

	printf("Terminating the server process\n");

	exit(EXIT_SUCCESS);
}
```

Output of above application

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx: ~/Desktop
File Edit View Search Terminal Help
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ gcc IPC_Message_Se
rver.c -o server
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ./server
Demonstartion of IPC using Message Queue
Message Queue is created succesfully
Enter Some message : First message
Enter Some message : Second message
Enter Some message : Third Message
Enter Some message : Bye
Enter Some message : end
Terminanting the server process
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Client Application

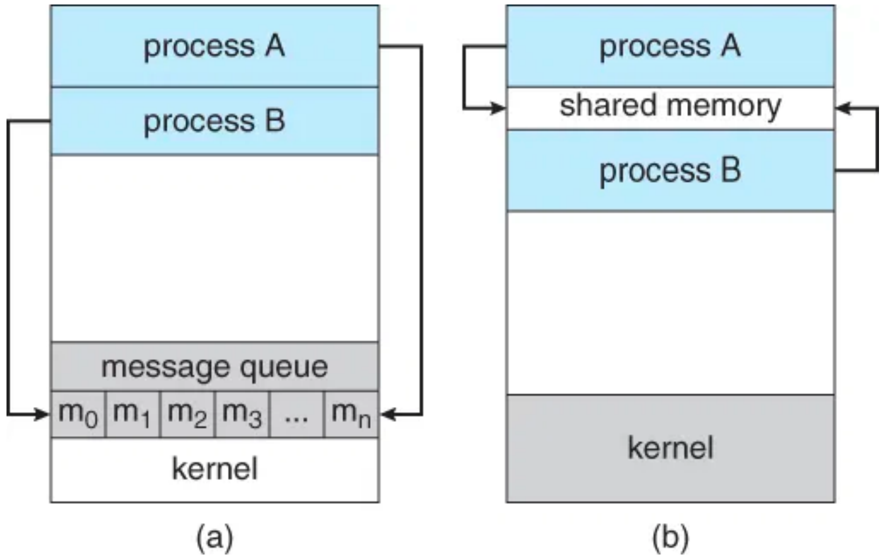
```
#include<stdio.h>
#include<string.h>
#include<errno.h>
#include<unistd.h>
#include<sys/msg.h>
#include<stdlib.h>

struct my_msg_st
{
	long int my_msg_type;
```

```
	char some_text[BUFSIZ];
};

int main()
{
	printf("Demonstration of IPC using Message Queue\n");

	printf("Client process is running\n");

	int running = 1;
	int msgid;
	struct my_msg_st some_data;
	long int msg_to_receive = 0;

	msgid = msgget( (key_t)1234,0666);

	printf("Fetching the messages from message queue\n");

	while (running)
	{
		msgrcv(msgid, (void*)&some_data,BUFSIZ,msg_to_receive,0);

		printf("Received Message: %s\n",  some_data.some_text);
		if(strncmp(some_data.some_text, "end", 3)== 0)
		{
			running = 0;
		}
	}

	printf("Terminating the client process\n");

	exit(EXIT_SUCCESS);
}
```

Output of above application

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx: ~/Desktop
File  Edit  View  Search  Terminal  Help
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ gcc IPC_Message_Cl
ient.c -o client
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ./client
Demonstration of IPC using Message Queue
Client process is running
Fetching the messages from message queue
Received Message: First message

Received Message: Second message

Received Message: Third Message

Received Message: Bye

Received Message: end

Terminating the client process
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Message Queue vs Shared Memory



Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.
An advantage of message passing model is that it is easier to build parallel hardware. This is because message passing model is quite tolerant of higher communication latencies. It is also much easier to implement than the shared memory model.

However, the message passing model has slower communication than the shared memory model because the kernel interaction.

The shared memory in the shared memory model is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other.

An advantage of shared memory model is that memory communication is faster as compared to the message passing model on the same machine.

However, shared memory model may create problems such as synchronization and memory protection that need to be addressed.

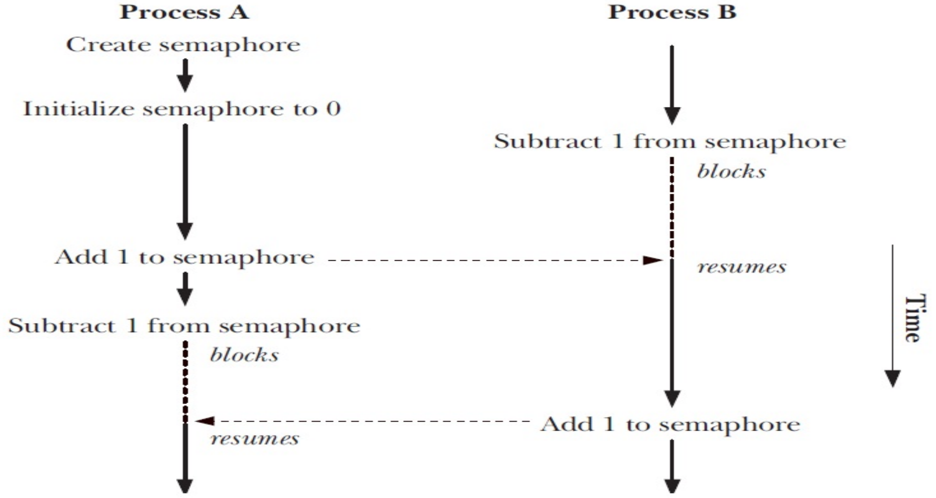
Semaphores

- Unlike the other IPC as message queue and shared memory semaphores are not used to transfer data between processes. Instead, they allow processes to synchronize their actions. One common use of a semaphore is to synchronize access to a block of shared memory, in order to prevent one process from accessing the shared memory at the same time as another process is updating it.
- A semaphore is a kernel-maintained integer whose value is restricted to being greater than or equal to 0. Various operations (i.e., system calls) can be performed on a semaphore, including the following:
  - setting the semaphore to an absolute value;
  - adding a number to the current value of the semaphore;
  - subtracting a number from the current value of the semaphore; and
  - waiting for the semaphore value to be equal to 0.

The general steps for using a System V semaphore are the following:

- Create or open a semaphore set using semget().
- Initialize the semaphores in the set using the semctl() SETVAL or SETALL operation. (Only one process should do this.)
- Perform operations on semaphore values using semop(). The processes using the semaphore typically use these operations to indicate acquisition and release of a shared resource.
- When all processes have finished using the semaphore set, remove the set using the semctl() IPC_RMID operation. (Only one process should do this.)

The semaphore system call allow processes to synchronize execution by doing set of operations automatically on a set of semaphores.
Before the implementation of semaphore, a process would create a lock file.
If create() system call fails, process assumes that some other process had already locked the resource.
A semaphore has integer value associated with it and 2 operations.
P : If signal is red, wait for the signal to go grenn, pass the signal and signal goes to red.
e.g. Before entering the critical section P operation must be performed.
V : Set the signal to green.
e.g. While leaving the critical section, V operation is performed.
P and V operations must be automatic from user's point of view. That is no INTR should occur while they run or processes would find semaphore values in inconsistent state.



Creating or Opening a Semaphore Set
The semget() system call creates a new semaphore set or obtains the identifier of an existing set.

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

Returns semaphore set identifier on success, or –1 on error

Semaphore Operations
The semop() system call performs one or more operations on the semaphores in the semaphore set identified by semid.

```
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

Returns 0 on success, or –1 on error

Unnamed pipe

Application program for Unnamed pipe

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define ReadEnd  0
#define WriteEnd 1

int main()
{
    int pipeFDs[2];
    char buf;
    const char* msg = "Marvellous Infosystems\n";

    printf("Demonstration of IPC using Unnamed Pipe\n");

    if (pipe(pipeFDs) < 0)
    {
        perror("Pipe Creation Failed");
        exit(-1);
    }

    pid_t cpid = fork();
    if (cpid < 0)
    {
        perror("Fork failed");
        exit(-1);
    }

    if (0 == cpid)
    {
         printf("Child process is running\n");

        close(pipeFDs[WriteEnd]);

         printf("Data received from parent process\n");

        while (read(pipeFDs[ReadEnd], &buf, 1) > 0)
        {
            write(STDOUT_FILENO, &buf, sizeof(buf));
        }

        close(pipeFDs[ReadEnd]);

        _exit(0);
    }
```

```c
    else
    {
        printf("Parent process is running\n");

        close(pipeFDs[ReadEnd]);

        write(pipeFDs[WriteEnd], msg, strlen(msg));

        close(pipeFDs[WriteEnd]);

        printf("Data is successfully written into pipe by parent process\n");

        wait(NULL);

        exit(0);
    }

    return 0;
}
```

Output of above application

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ gcc IPC_UnnamedPip
e.c -o unnamed
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ./unnamed
Demonstration of IPC using Unnamed Pipe
Parent process is running
Data is successfully written into pipe by parent process
Child process is running
Data received from parent process
Marvellous Infosystems
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Named Pipe (FIFO)

Application program for Named pipe

Server Application

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include<stdio.h>
#include<string.h>

int main()
{
	int fd;
	char * myfifo = "/tmp/myfifo";

	printf("Demonstration of IPC using Named Pipe\n");

	printf("Server is running\n");

	// create the FIFO (named pipe)
	mkfifo(myfifo, 0666);

	// write data to the FIFO
	fd = open(myfifo, O_WRONLY);
	write(fd, "Marvellous Message", strlen("Marvellous Message")+1);
	close(fd);

	printf("Data successfully written in named pipe\n");

	return 0;
}
```

Output of above application

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx: ~/Desktop
File Edit View Search Terminal Help
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ gcc IPC_NamedPipe_
Server.c -o server
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ./server
Demonstration of IPC using Named Pipe
Server is running
Data successfully written in named pipe
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Client Application

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

#define MAX_BUF 1024

int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";
    char buf[MAX_BUF];

    printf("Demonstration of IPC using Named Pipe\n");

    printf("Client process is running\n");

    /* open, read, and display the message from the FIFO */
    fd = open(myfifo, O_RDONLY);
    read(fd, buf, MAX_BUF);
    printf("Received data is : %s\n", buf);

    close(fd);

    return 0;
}
```

Output of above application

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx: ~/Desktop
File  Edit  View  Search  Terminal  Help
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ gcc IPC_NamedPipe_
Client.c -o client
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ./client
Demonstration of IPC using Named Pipe
Client process is running
Received data is : Marvellous Message
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Application program for signals

```
#include <stdio.h>
#include <signal.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

void sighup();
void sigint();
void sigquit();

int main()
{
    int pid;
    if ((pid = fork()) < 0)
    {
            exit(1);
    }

   if (pid == 0)    // Child process
   {
            signal(SIGHUP,sighup);
            signal(SIGINT,sigint);
            signal(SIGQUIT, sigquit);
            for(;;);
    }
    else              // Parent process
    {
            printf("\nPARENT: sending SIGHUP\n\n");
            kill(pid,SIGHUP);
            sleep(3);

            printf("\nPARENT: sending SIGINT\n\n");
            kill(pid,SIGINT);
            sleep(3);

            printf("\nPARENT: sending SIGQUIT\n\n");
            kill(pid,SIGQUIT);
            sleep(3);
    }
}

void sighup()
{
        signal(SIGHUP,sighup);
        printf("CHILD: I have received a SIGHUP\n");
}

void sigint()
```

```
{
        signal(SIGINT,sigint);
        printf("CHILD: I have received a SIGINT\n");
}

void sigquit()
{
        printf("Parent process kill child\n");
        exit(0);
}
```

Output of above application

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx: ~/Desktop
File Edit View Search Terminal Help
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ gcc IPC_Signals.c
-o myexe
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ./myexe

PARENT: sending SIGHUP

CHILD: I have received a SIGHUP

PARENT: sending SIGINT

CHILD: I have received a SIGINT

PARENT: sending SIGQUIT

Parent process kill child
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Important points about Inter Process Communication:

- Using ipcs command we can obtain information about IPC objects on the system. By default, ipcs displays all IPC objects (Message queue, Shared memory, Semaphore).

Show inter-process communication

The below commands show inter-process communication facilities status.

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x000004d2 0          marvellous 666        1024         2

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 8          marvellous 600        67108864   2          dest
0x00000000 32778      marvellous 600        524288     2          dest
0x00000000 11         marvellous 600        524288     2          dest
0x00000000 32780      marvellous 600        4194304    2          dest
0x00000000 14         marvellous 600        524288     2          dest
0x00000000 18         marvellous 600        524288     2          dest
0x00000000 32801      marvellous 600        524288     2          dest
0x00000000 32802      marvellous 600        524288     2          dest
0x00000000 65574      marvellous 600        1048576    2          dest
0x61070053 32818      marvellous 666        30         0
0x00000000 32821      marvellous 600        524288     2          dest
0x00000000 32822      marvellous 600        16777216   2          dest
0x00000000 55         marvellous 600        524288     2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Active semaphore sets

Print information about active semaphore sets.

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Shared memory segments

Print information about active shared memory segments.

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 8          marvellous 600        67108864   2          dest
0x00000000 32778      marvellous 600        524288     2          dest
0x00000000 11         marvellous 600        524288     2          dest
0x00000000 32780      marvellous 600        4194304    2          dest
0x00000000 14         marvellous 600        524288     2          dest
0x00000000 18         marvellous 600        524288     2          dest
0x00000000 32801      marvellous 600        524288     2          dest
0x00000000 32802      marvellous 600        524288     2          dest
0x00000000 65574      marvellous 600        1048576    2          dest
0x61070053 32818      marvellous 666        30         0
0x00000000 32821      marvellous 600        524288     2          dest
0x00000000 32822      marvellous 600        16777216   2          dest
0x00000000 55         marvellous 600        524288     2          dest

marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Shows limits

The IPCS –l shows limits of shared memory, semaphores, and messages.

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ipcs -l

------ Messages Limits --------
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398509481980
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767

marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

Usage of IPC facilities

In the below option ,'u' displays current usage for all the IPC facilities.

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ ipcs -u

------ Messages Status --------
allocated queues = 1
used headers = 2
used space = 1024 bytes

------ Shared Memory Status --------
segments allocated 13
pages allocated 22785
pages resident  5195
pages swapped   0
Swap performance: 0 attempts     0 successes

------ Semaphore Status --------
used arrays = 0
allocated semaphores = 0

marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

There are Three read-only files in the /proc/sysvipc directory provide the same information as can be obtained via ipcs command :

/proc/sysvipc/msg lists all messages queues and their attributes.

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx: ~/Desktop
File Edit View Search Terminal Help
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ cat /proc/sysvipc/msg
       key      msqid perms      cbytes       qnum lspid lrpid   uid   gid  cuid  cgid      stime
   rtime      ctime
      1234          0   666        1024          2 28859 27766  1000  1000  1000  1000 1588168237 15
88167176 1588166456
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

/proc/sysvipc/sem lists all semaphore sets and their attributes.

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx: ~/Desktop
File Edit View Search Terminal Help
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ cat /proc/sysvipc/sem
       key      semid perms      nsems   uid   gid  cuid  cgid      otime      ctime
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$
```

/proc/sysvipc/shm lists all shared memory segments and their attributes.

```
marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx: ~/Desktop
File Edit View Search Terminal Help

marvellous@marvellous-HP-ENVY-x360-Convertible-15m-bp1xx:~/Desktop$ cat /proc/sysvipc/shm
       key      shmid perms                  size  cpid  lpid nattch   uid   gid  cuid  cgid      at
ime      dtime      ctime                   rss                   swap
         0          8  1600              67108864  1715 25992      2  1000  1000  1000  1000 1588163
418 1588163418 1588177424              16588800                      0
         0      32778  1600                524288  4298 25980      2  1000  1000  1000  1000 1588163
405 1588163405 1588161649                188416                      0
         0         11  1600                524288  1830  1370      2  1000  1000  1000  1000 1588177
485          0 1588177485                512000                      0
         0      32780  1600               4194304  4298 25980      2  1000  1000  1000  1000 1588163
405 1588163405 1588161712                557056                      0
         0         14  1600                524288  1827 28215      2  1000  1000  1000  1000 1588167
523 1588167523 1588177488                 81920                      0
         0         18  1600                524288  2174 30524      2  1000  1000  1000  1000 1588178
263 1588178263 1588180002                122880                      0
         0      32801  1600                524288  1830  1370      2  1000  1000  1000  1000 1588162
375          0 1588162374                524288                      0
         0      32802  1600                524288  1830  1370      2  1000  1000  1000  1000 1588162
406          0 1588162406                294912                      0
         0      65574  1600               1048576  2174 30524      2  1000  1000  1000  1000 1588178
263 1588178263 1588169102                212992                      0
1627848787      32818   666                    30 25968 26352      0  1000  1000  1000  1000 1588164
092 1588164092 1588163399                  4096                      0
         0      32821  1600                524288 26396 28781      2  1000  1000  1000  1000 1588168
```

Good books to refer for Inter Process Communication:

- The Linux Programming interface by Michael KerrisK. (Chapter no 43,44,45,46,47,48)
- Linux System Programming by Robert Love. (Chapter no 5 & 6)
- Advanced Programming in the UNIX Environment by W. Richard Stevens (Chapter no 15)
- The design of the Unix operating system by Maurice J. Bach (Chapter no 11)

For any queries feel free to ask
Piyush Manohar Khairnar
WhatApp : 7020713938
Mail : marvellousinfosystem@gmail.com