INTERPROCESS COMMUNICATION [IPC]

- 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.

FORMS OF IPC:

Several forms of PC:

- Pipes
- Named pipes (also called as FIFO)
- Signals
- Message queue ┐
- Shared memory ├── [System V IPC]
- Semaphores ┘
- Sockets [Free BSD]

SYSTEM V IPC:-

(1) The UNIX system V IPC consists of 3 mechanisms …
   (a) Message queue.
   (b) Shared memory.
   (c) Semaphores.

(ii) Message queue, shared memory and semaphores are also referred to as IPC resources.

(iii) Message queue allows processes to send formatted data streams to arbitrary processes. Shared memory allows processes to share parts of virtual address space. Semaphore allows processes to synchronize their execution.

(iv) At the implementation level, they share common processes…
   (a) Just like process table / file table , kernel maintains the global table for each mechanism. The entry of this table is the instance of that mechanism.
   (b) Key: Each entry contains a numeric key which is its user chosen name.
      Key is analogues to file name and is used to identify exclusively an IPC resource.
      Cooperating processes wishing to access the same IPC resource must use the same key.

The key is implemented as an int so like file descriptor, processes use int to identify IPC resources, they wish to access.

(c) ID:

ID is analogues to file descriptors. When program attaches to an IPC resource, the system call returns id.

Subsequent system call use this ID to identify IPC resource being used.

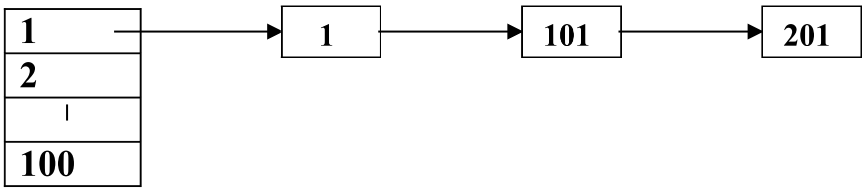
(d) get:

Each mech contains a get system call to create a new entry or to retrieve an existing one from IPC global table.

- **get() takes 2 parameters 1) key and 2) flag.**
- **get() returns ID that is described in (c)**
- **id <- get(key , flag);**
- **flag has the following values.**
- **IPC_PRIVATE , IPC_CREAT , IPC_EXCL.**
- **IPC_CREAT : To create a new entry in the global table or if exists the key entry in the table then retrieve it.**
- **IPC_CREATE | IPC_EXCL : If you want exclusive new entry, then this combination is used. If key entry is already present in the table error is returned.**

(e) Find entry in the table:

To find the index into the table from the descriptor…
Index = descriptor mod(no. of entries in table).
e.g. If table contains 100 entries then 1 , 101 , 201 …. are in the entry of 1.



Hash function of no. of entries in the table.

(f) Each entry contains the status info such as the…

- **proc ID of last process to update entry (sndmsg , rcvmsg , shmat so on).**
- **Time of last access or update.**

(g) Each mech contains "CONTROL" system call.

- **To query status of an entry.**
- **To set the status information.**
- **To remove entry from system.**

(h) IPC_PERM struct:

OS uses permissions to access IPS resource.

Each process is free to create an IPC resource, but once created the perm restrict to access it.

When an IPC resource is created it has ipc_perm struct associated with it. The access modes in this struct are similar to file permissions.

```
struct ipc_perm
{
        uid_t uid;    //uid of resource owns.
        gid_t gid;    //gid of resource owns.
        uid_t cuid;  // uid of resource creates.
        gid_t cgid;  // gid of resource creates.
        mode_t mode;       // Resource access modes.
        unsigned long seq;          // Sequence no.
        key_t key;   // global resource key.
};
```

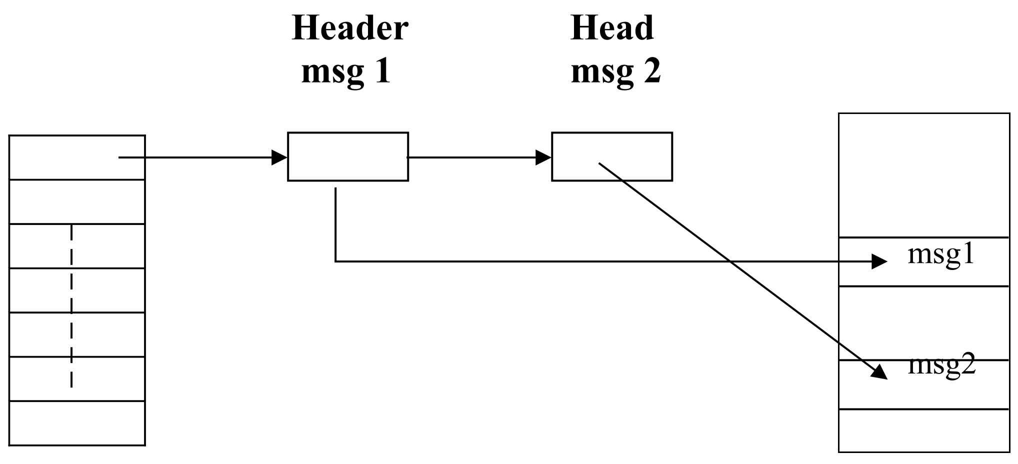
- **uid , gid : uid and gid of current owner of resource. Privilege process can change these fields with IPC_SET control option to allow non-privilege process to access resource.**
- **cuid , cgid : userid and groupid of the creates of the resource. These fields remains constant throughout the life of resource.**
- **mode : Access mode of the resource. The perm bits are same as those used for a file.**
- **seq : unique resource ID.**
- **key : key for the resource.**

MESSAGE QUEUE.

(1) Introduction: Messages are passed between the processes vid message queue, which is identified by “MSG QUE IDENTIFIES”.

All messages that are passed between communicating processes are linked into the queue.

When msg is sent, it is sent to the back of queue and when msg is read it is unlinked from front of queue.



Message queue headers **Data area**

There are 4 system calls used with msg queue…

- **msgget()**
- **msgch()**
- **msgsnd()**
- **msgrcv()**
-

(2) msgget() : Message allocation.

(a) Syntax : msggid = msgget(key , flag);

(b) The entry in the message queue is allocated with msgget() system call.

(c) 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.

(d) Depending on the flag value… IPC_EXEL | IPC_CREAT then if it finds an empty for the key, then msgget() returns an error.

(e) The system call handlers for magget is msgget();

(f) 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.

(g)
```
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;
        };
```

- 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.
-

(h)

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

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

(3) MSG_OPERATION : msgsnd()

(a) 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.
        }
```

(b) msgsnd(msqid , msg , count , flag)

(c) 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.

(d) 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.

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

(f) 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.

(g) 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.

(4) MSG OPERATION : msgrcv()

(a) 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 iis 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 queuw)
          goto loop;
  }
```

(b) count = msgrcv(id , msg , maxcnt , type , flag);

(c) 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.

(d) If there are no messages to receive, the process sleeps or returns depending on IPC_NOWAIT.

(e) 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.

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

(g) 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.

(h) 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.

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

(5) MSG CONTROL : msgctl()

(a) With msgctl()….
- A process can query the status of msg descriptor.
- Can set its status.
- Can remove msg descriptor from global table.

(b) msgctl(id , cmd , mstatbuf)

(c) id : The id for message queue.

(d) cmd : 3 commands can be specified.

(e) IPC_STAT : Copies msgid_ds held in kernel into the user data struct pointed by 3 parameters.

(f) IPC_SET : Alters the fields in msqid_ds struct. The fields uid , gid , mode and qbytes can be altered.

(g) IPC_RMID : Removes the msg queue and all msgs help on it. Super user or creator of query can only do this operation.

(6) Message queue tunable + global information

(i) Initialization of the message queue struct is done mostly by the compiler when kernel is built.

(ii) Message queue tunable parameters can be configured by the system admin. To take effect the system must be rebuilt and rebooted.

(iii) The variables of struct msginfo are same as that of tunable parameters.

struct msginfo {	Tunable param
msgmap	MSGMAP
msgmax	MAGMAX
msgmnb	MSGMNB
msgmni	MSGMNI
magssz	MSGSSZ

```
        msgtql                              MSGSEG
    }                                       MSGTQL
```

MSGMAP : Size of rmalloc() map for msg data.
MSGMAX : Largest msg size allowed.
MSGMNB : Max no. of bytes allowed on message queue.
MSGMNI : no .of message queue.
MSGSSZ : size of message allocation unit.
MSGSEG : no. of msg segments.
MSGTQL : no. of msg headers.

(iv) Message global var :

- (a) struct map magmap[] : Map used by rmalloc().
- (b) struct msquid_ds mqueue[] : Message queue array.
- (c) struct msg msgh[] : Message headers.
- (d) struct msginfo msginfo : System tunable msginfo struct.
- (e) struct msg *msgfp : Message headers freelist (pts into msgh)
- (f) paddr_t msg : Message storage buffers managed by msgma();

Shared Memory.

(a) Introduction

- (i) Shared memory (SM) is exactly what its name implies.. an area of memory that is shared between 2 or more processes.
- (ii) SM has certain size and some physical address.
- (iii) 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.
- (iv) 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.
- (v) The advantage of using SM for IPC is SPEED. It is very quick to exchange between 2 processes.
- (vi) Database systems uses SM to exchange queries and results between the *data base client* U1 and the *data base server*.
- (vii) Disadvantage is that synchronization is must. I.e. they must ensure the synchronize access to this memory.

(b) INITILIZATION shmget() :

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

(ii) The shmget() system call converts this IPC key to SM identifies.

(iii) 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.

(iv) 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;
};
```

- 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 anon 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.

(v) Search or Create :

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.

(vi) 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.

(vii) Finally the remaining fields of shmid_ds are initialized.

(c) SHARE MEMORY OPTIONS : shmat()

(i) Process attached a shared memory region to its address space using shmat system call…

```
Vaddr = shmat(id , addr , flags);
```

(ii) 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 flag;
                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);
}
```

(iii) 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.

(v) 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.

(vi) 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.

(vii) 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.

(viii) 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.

(ix) Finally return the virtual address.

(d) Shared Memory OPERATION : shmdt()

(i) shmdt(addr)
addr : Virtual address in the process for this shared memory region.

(ii) Check the address supplied is the shared memory segment address in that process.
The kernel must keep trackl 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.
One of the member in procedure structure (P – segacct) holds the linked list of segacct structure, which locate the shared memory segment for the process.

```
struct segacct
{
        struct segacct * sa_next;
        caddr_t sa_addr;
        size_t len;
        struct anon_map *sa_map;
};
```

- **sa_next : Pointer to next segacct for this process.**
- **Sa_addr : virtual address at which segment is mapped.**
- **len : size of segment.**
- **anon_map : points to anon_map.**

(iii) Although it would seem more logical to use shared memory identifies to remove…
(a) Same shared memory segment can be mapped at more than virtual address in the same process. So the id is same for all those entries. So to distinguish between the 2 instances of shared segment.
(b) The identifier may have been removed.

(iv) The kernel searches this entry in pregion (depending on virtual address) and detaches it from kernel region table using DETACHREG().
Since pregion has ptr to region table entry, region table entry the shared memory entry is search. But remembers region table doesn't have BACK ptr to shared memory table.

(v) Finally the fields in shared memory table entry (shmid_ds) like reference count, dtime (last detach time) are changed.

(vi) If reference count falls to 0, the corresponding pages are freed.

(e) Shared Memory Control : shmctl()

(i) shmctl (id,cmd,shmstatbuf);
id : ID for shared memory table entry.
Cmd : type of operation.
Shmstatbuf : used in query or set param.

It is the address of user level DS to content the resultant.

(ii) It is used for…

(a) query the status.

(b) Set param for shared memory region.

(c) Remove the entry.

(iii) IPC_RMID :

This is the valued for cmd in shmctl() to remove the shared memory segment.

The kernel frees the entry in shared memory table corresponding to given id.

If no process has region attched to its virtual address space, it freed the region table entry and all its resources using freereg().

If region is still attached to some process (ref count > 0), the kernel just clears the flag in region table which was indicating do not free the region even if last process detach the region.

Process which were using the shared memory, still continuous doing so. But new process can not attach it.

When all process detaches the region, the kernel frees the region and also are the pages.

(f) General :

(i) Shared memory is controlled by 4 tunable parameters that are held in global shminfo struct.

```
{
        shmmax;
        shmmin;
        shmmni;
        shmseg;
}
```

These are initialize from following param…

SHMMAX : Specifies largest shared memory segment allowed.

SHMMIN : min size of shared memory segment allowed.

SHMMNI : Configures no. of shared memory identifies in the system.

SHMSEG : Configure the max no. of shared memory segment that a single process can attach to.

(ii) The reference count on shared memory segment is altered by following…

- **shmget : reference count initialized to z.**
- **shmat : iNCR reference count for each attach segment.**

- fork : When parent forks(), new child proc is created and it is attached to all shared memory segment that parent is attached to. So the reference count of all shared memory is incremented by 1.
- shmdt : decrement reference count.
- exit : reference count of all shared memory segment are decremented to which this process is attached.
- Exec : The process detaches from all shared memory segment. When issued exec() so decrements.
- IPC_RMID : Reference count = 1 remove segment.

SEMAPHORE.

(a) Introduction :

(i) The semaphore system call allow processes to synchronize execution by doing set of operations automatically on a set of semaphores.

(ii) Its dictionary meaning is signaling apparatus, which has arm that moves up and down. Best example to visualize this is old fashioned railway line signals.

(iii) 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.
The major disadvantage :
(a) When to try again.
(b) Lock files may be left behind. When system crashes.

(iv) 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.

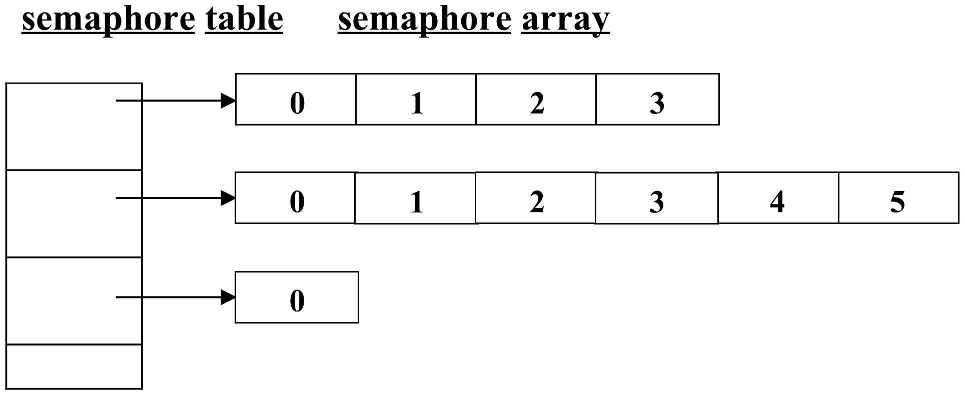
(v) P and V operations must be automatic from user's point of view. That is no INTR should occur while they run or processes would find semaphore values in inconsistent state.

(vi) The semaphore value need not be a Boolean. It can be an integer.
For P operation -> Semaphore value is decremented.
For V operation -> Semaphore value is incremented.

(vii) It may happen, process claims for a buffer, that is it performed P operation and then it dies because it receives the signal, and not perform V operation. So semaphore implementation should provide the means to undo the claim.

(viii) Semaphore may be binary or it may have int value. e.g. If it is protecting a buffer pool of 20 then its initial value can be set to 20.

(b) semget()



(i) In SVR4, semaphore identifier refers to a group of individual semaphores. A group can be of 1 or more than 1 semaphores. For simple semaphore option, uuse single semaphore as a group. If multiple semaphore claim are to be performed, group of multiple semaphores must be used.

(ii) Each group is identified by semid_ds struct whereas individual entry in semaphore is identified by sem_struct.

(iii) struct semif_ds

```
{
        struct ipc_perm sem_perm;
        struct sem *sem_base;
        unsigned short sem_nsems;
        time_t sem_otime;
        time_t sem_ctime;
};
```

- **sem_perm : Permissions.**
- **sem_base : ptr to semaphore array.**
- **sem_nsems : no. of semaphores in group.**
- **sem_otime : Time of last operation.**
- **Sem_ctime : Time of last change.**

(iv) struct sem

```
{
        unsigned short semval;
        pid_t sempid;
        unsigned short semncnt;
        unsigned short semzcnt;
};
```

- **semval : Current val of semaphore.**
- **sempid : Process that did last operation.**

- semncnt : Count of processes waiting to perform P operation.
- semzcnt : Count of processes waiting for semval to be zero.

(v) id = semget(key , count , flag);

- key : unique , used as filename i.e. for programmes purpose.
- count : The kernel allocates an entry that points to an array of semaphore struct with count element.
- flag : IPC_PRIVATE , IPC_CREAT , IPC_EXCL.

(vi) This semid must be used for subsequent system calls.

(vii) Semaphore persists until such time it is deallocated or until OS is next rebooted. Thus even if process exits without deallocating semaphores, they remain in kernel address space and OS eventually exhausts its supply of semaphores.

(c) semop :

(i) Processes manipulated semaphores with semop system call.

```
Oldval = semop(id,oplist,count);
```

id : semid in semaphore table to be operate.

oplist : A pointer to an array of semaphore operations. An array of sembuf structures that specifies which operations are to be performed on which semaphore in the group.

count : Specifies how many sembuf structures in oplist array.

(ii) Algorithm : semop

```
input : (1) sem descriptor.
        (2) Array of sem operations.
        (3) No. of elements in the array.
Output : start value of last semaphore operated on.
{
        check legality of semaphore descriptor;
        start :
        -   read array of semaphore operation. From user -> kernel space;
        -   Check permissions for all semaphore operations.
        -   For(each semaphore operation in array)
            {
                if(semaphore operation is positive)
                        {

                        }
            }
}
```

(iii) struct sembuf

```
{
        unshort sem_num;
        short sem_op;
        short sem_flg;
};
```

- sem_num : semaphore no. in a group.
- sem_op : Operation to be performed.
- sem_flg : IPC_NOWAIT , SEM_UNDO.

(iv) Permissions :

The kernel reads array of semop from user address space to kernel. It checks the semaphore nos. are legal and running process has necessary permissions to read or change semaphores.

(v) AUTOMIC OPERATION Property :

The ATOMICITY property says either perform all or nothing.

If condition arises that the kernel must sleep while it is operating on a group of semaphores, it must restore the semaphores it has already operated on to their values at the start of system call.

It sleeps till event to wakeup occurs and it RESTARTS the system call. Kernel had already saved the old semaphores in a global array. So it reads these arrays from uses space again if it is going to restart system call.

Thus ATOMICITY is maintained.

(vi) Operation value in sem_buf struct :

(a) >0

- **Used to leave critical section.**
- **For V operation , sem_op has value 1.**
- **It increments the semaphore value.**
- **In general the value of sem_op is added to the semaphore value (e.g. 20 buffers ex).**
- **Kernel awakens all the processes that are waiting for the semaphore value to become > 0.**

(b) = 0

- **If sem_op = 0, the kernel checks semaphore value.**
- **It is used to wait for the value of semaphore to fall to zero.**
- **If semaphore value = 0, then kernel continues with the next semaphore in the array else the process sleeps until it become zero.**

(c) <0

- **Is used to enter critical section.**

- **For P operation, if absolute(operational value) <= semaphore value then semaphore value = semaphore value – operational value.**
- **Now obtained semaphore value = 0, kernel awakens all the processes waiting for this semaphore to become zero.**
  (In the case (vi) (b))
- **If absolute(operational value) > semaphore value, that is, resources that process required is not available with kernel, so that the process to sleep on the event of increment semaphore value.**

Here process sleeps at the interruptible priority, hence it wakes up on the receipt of signal.

(vii) Deadlock :

In order to avoid deadlock situation, if you want to use no. of semaphores in a single process, try to group them as far as possible in the semaphore operation array.

(viii) Conditional semaphore :

While executing semop(), if situation arises when the process goes to sleep on an event to exceed part semaphore's value. In that case you can specify IPC_NOWAIT in system call, so instead the process to sleep it will return from the system call with an error condition. Thus we can make the process NOT to sleep it is unable to do ATOMIC OPERATION.

(ix) SEM_UNDO :

(a) What happens when process is operating with source semaphore and those semaphores have locked some resources and process exits without removing semaphores. Will that keep the lock on the resources till system shuts down?

(b) To avoid such problems the use of SEM_UNDO flag. When process exits either normally as abnormally, the kernel reverses the effect of every semaphore operation the process had done. This is called semaphore undo operation.

(c) A semundo operation relies on sem_undo struct…

```
struct sem_undo
{
   struct sem_undo *un_np;
   struct un_cnt;
   short un_aoe;
   unsigned short un_num;
   int un_id;
};
```

un_np : Points to the next group of undo struct.
un_cnt : no. of entries in the group currently in use.
un_aoe : shadow / adjust_on_exit value.
un_num : Semaphore number.
un_id : Semaphore identifier.

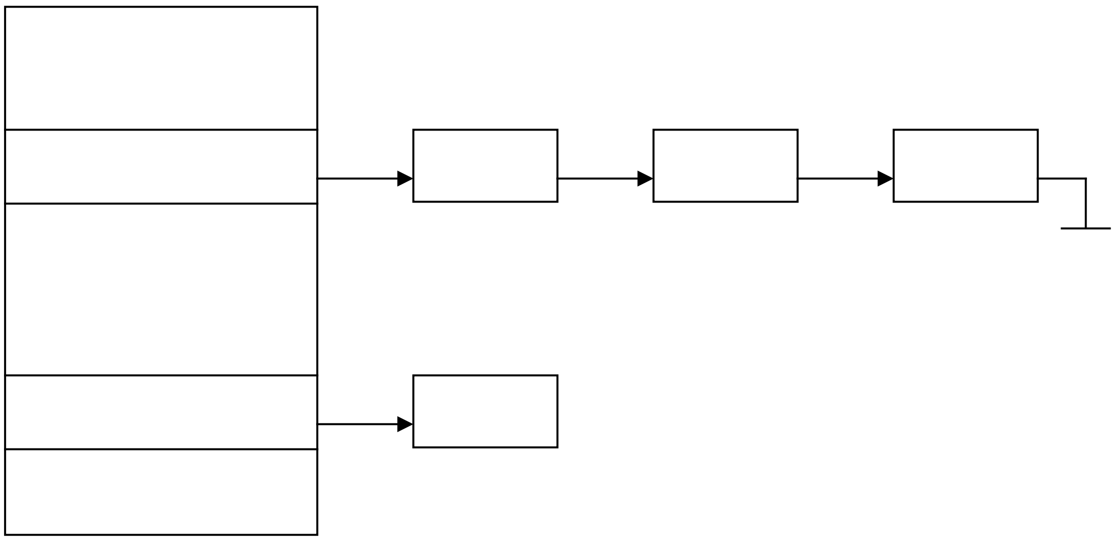
(d) Kernel maintains a table with one entry for every process in the system. Each entry points to the undo structure, one for each semaphore.

(e) The kernel allocates and deallocates the sem_undo struct dynamically when process executes semop() with SEM_UNDO flag.

(f) If process_semno combination does not exits, then one is allocates for the P operation. If one is already present, the shadow value / Adjust value is adjusted, depending on P or V operation.

(g) For P operation, the shadow /Adjust value is decremented, since semaphore value is incremented.

Per process undo headers. **Undo struct LL.**



(h) When process exits, the semaphore function semexit() is called by exit(), to perform any semaphore clanup opearion for the process. Semexit() locks all the entries for the process and address shadow value for the semaphore. The sem_undo structure are returned to the free list and the entry in the header set to zero.

(d) semctl()

(i) Performs miscellaneous operations on a group of semaphores.

(ii) In particular it can be used to do the following…

- **Get or set value of part semaphore.**
- **Query the fields in semid_ds.**
- **Set the fields in semid_ds.**
- **Delete semid_ds entry.**

(iii) semctl(id,num,amd,arg)

- **id: semaphore group id.**
- **Num : semaphore no. in above group.**

(iv) cmd : The cmd parameters specify the following values….

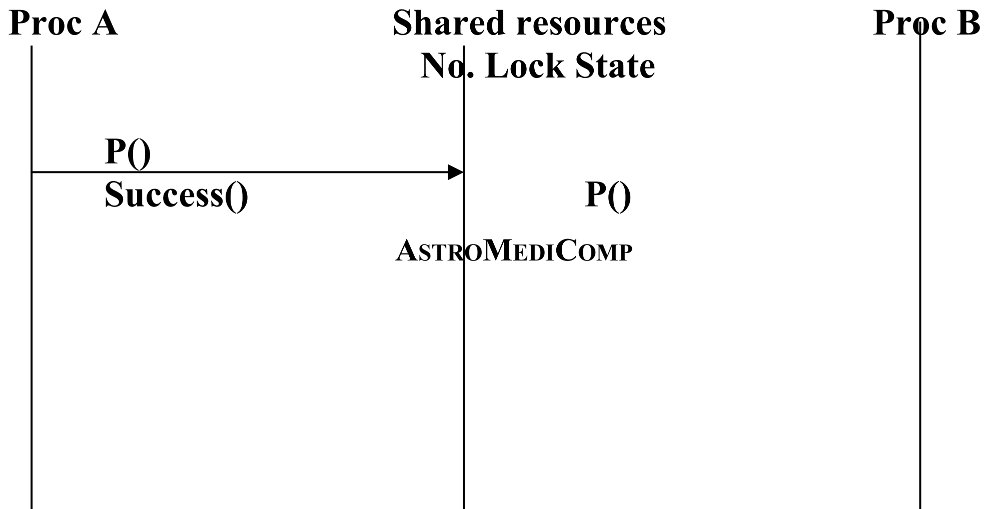
- **GETVAL : returns the value of semaphore in a group. Id_num combination identifies the semaphore.**
- **SETVAL : Set the value of semaphore to the value in agr.value.**
- **GETPID : Getpid of process who last altered the semaphores. It returns semid_ds.semid.**
- **GETNCNT , GETZCNT : semid_ds.semncnt semid_ds.semzcnt**
- **GETALL : Fills the array pointed to by arg.a with the values of all semaphores in the group specified by id.**
- **SETALL : Allows the program to set all the values of semaphore provided in arg.a and id.**
- **IPC_STAT : read semid_ds. It is put in arg.b.**
- **IPC_SET : Set uid,gid and mode fields in semid_ds with the values provided in args.b. This is privilege instruction.**
- **IPC_RMID : Removes semaphores group and all other data structures associated with it.**

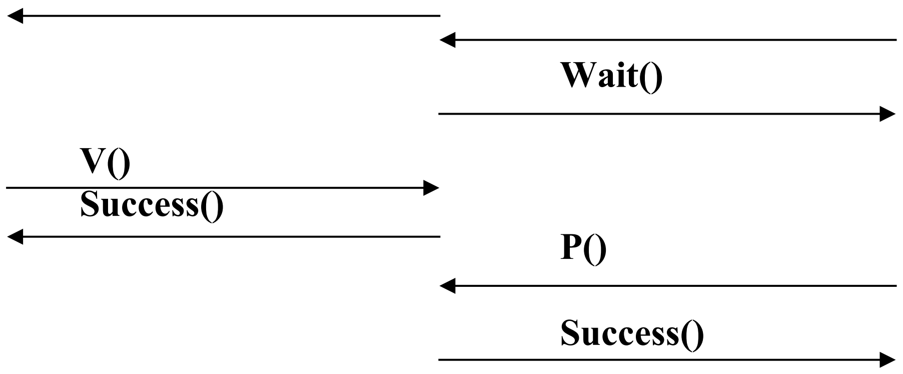
(v) arg :

```
union semun
{
        int value;
        struct semid_ds *b; // For IPC_STAT and IPC_SET.
        unsigned short *a; // For GETALL and SETALL.
};
```

(e) SEMAPHORE USES IN OS :

(i) Mutual exclusion : Prevents more than one processes accessing the same data at the same time.





(ii) Co Operating Processes :

One generates the data and others use this data.