# System V Inter Process Communication

krishna balasubramanian,

# 1 System V IPC.

These facilities are provided to maintain compatibility with programs developed on system V unix systems and others that rely on these system V mechanisms to accomplish inter process communication (IPC).

The specifics described here are applicable to the Linux implementation. Other implementations may do things slightly differently.

## 1.1 Overview

System V IPC consists of three mechanisms:

- Messages : exchange messages with any process or server.
- Semaphores : allow unrelated processes to synchronize execution.
- Shared memory : allow unrelated processes to share memory.

Access to all resources is permitted on the basis of permissions set up when the resource was created.

A resource here consists of message queue, a semaphore set (array) or a shared memory segment.

A resource must first be allocated by a creator before it is used. The creator can assign a different owner. After use the resource must be explicitly destroyed by the creator or owner.

A resource is identified by a numeric *id*. Typically a creator defines a *key* that may be used to access the resource. The user process may then use this *key* in the *get* system call to obtain the *id* for the corresponding resource. This *id* is then used for all further access. A library call *ftok* is provided to translate pathnames or strings to numeric keys.

There are system and implementation defined limits on the number and sizes of resources of any given type. Some of these are imposed by the implementation and others by the system administrator when configuring the kernel (See Section 1.5.5 [msglimits], page 8, See Section 1.6.4 [semlimits], page 11, See Section 1.7.5 [shmlimits], page 14).

There is an `msqid_ds`, `semid_ds` or `shmid_ds` struct associated with each message queue, semaphore array or shared segment. Each ipc resource has an associated `ipc_perm` struct which defines the creator, owner, access perms ..etc.., for the resource. These structures are detailed in the following sections.

## 1.2 example

Here is a code fragment with pointers on how to use shared memory. The same methods are applicable to other resources.

In a typical access sequence the creator allocates a new instance of the resource with the `get` system call using the IPC_CREAT flag.

creator process:



```
#include <sys/shm.h>
int id;
key_t key;
char proc_id = 'C';
int size = 0x5000; /* 20 K */
int flags = 0664 | IPC_CREAT; /* read-only for others */

key = ftok ("~creator/ipckey", proc_id);
id = shmget (key, size, flags);
exit (0); /* quit leaving resource allocated */
```

Users then gain access to the resource using the same key.
Client process:

```
#include <sys/shm.h>
char *shmaddr;
int id;
key_t key;
char proc_id = 'C';

key = ftok ("~creator/ipckey", proc_id);

id = shmget (key, 0, 004); /* default size   */
if (id == -1)
      perror ("shmget ...");

shmaddr = shmat (id, 0, SHM_RDONLY); /* attach segment for reading */
if (shmaddr == (char *) -1)
      perror ("shmat ...");

local_var = *(shmaddr + 3);  /* read segment etc. */

shmdt (shmaddr); /* detach segment */
```

When the resource is no longer needed the creator should remove it.
Creator/owner process 2:

```
key = ftok ("~creator/ipckey", proc_id)
id = shmget (key, 0, 0);
shmctl (id, IPC_RMID, NULL);
```

## 1.3 Permissions

Each resource has an associated `ipc_perm` struct which defines the creator, owner and access perms for the resource.

```
struct ipc_perm
        key_t key;    /* set by creator */
        ushort uid;   /* owner euid and egid */
        ushort gid;
        ushort cuid;  /* creator euid and egid */
        ushort cgid;
```

```
ushort mode;  /* access modes in lower 9 bits */
ushort seq;   /* sequence number */
```

The creating process is the default owner. The owner can be reassigned by the creator and has creator perms. Only the owner, creator or super-user can delete the resource.

The lowest nine bits of the flags parameter supplied by the user to the system call are compared with the values stored in `ipc_perms.mode` to determine if the requested access is allowed. In the case that the system call creates the resource, these bits are initialized from the user supplied value.

As for files, access permissions are specified as read, write and exec for user, group or other (though the exec perms are unused). For example 0624 grants read-write to owner, write-only to group and read-only access to others.

For shared memory, note that read-write access for segments is determined by a separate flag which is not stored in the `mode` field. Shared memory segments attached with write access can be read.

The `cuid`, `cgid`, `key` and `seq` fields cannot be changed by the user.

## 1.4 IPC system calls

This section provides an overview of the IPC system calls. See the specific sections on each type of resource for details.

Each type of mechanism provides a *get*, *ctl* and one or more *op* system calls that allow the user to create or procure the resource (get), define its behaviour or destroy it (ctl) and manipulate the resources (op).

### 1.4.1 The *get* system calls

The `get` call typically takes a *key* and returns a numeric *id* that is used for further access. The *id* is an index into the resource table. A sequence number is maintained and incremented when a resource is destroyed so that acceses using an obselete *id* is likely to fail.

The user also specifies the permissions and other behaviour charecteristics for the current access. The flags are or-ed with the permissions when invoking system calls as in:

```
msgflg = IPC_CREAT | IPC_EXCL | 0666;
id = msgget (key, msgflg);
```

- `key` : IPC_PRIVATE => new instance of resource is initialized.
- `flags` :

  IPC_CREAT : resource created for *key* if it does not exist.

  IPC_CREAT | IPC_EXCL : fail if resource exists for *key*.
- returns : an identifier used for all further access to the resource.

Note that IPC_PRIVATE is not a flag but a special `key` that ensures (when the call is successful) that a new resource is created.

Use of IPC_PRIVATE does not make the resource inaccessible to other users. For this you must set the access permissions appropriately.



There is currently no way for a process to ensure exclusive access to a resource. IPC_CREAT | IPC_EXCL only ensures (on success) that a new resource was initialized. It does not imply exclusive access.

See Also : See Section 1.5.1 [msgget], page 5, See Section 1.6.1 [semget], page 9, See Section 1.7.1 [shmget], page 12.

### 1.4.2 The *ctl* system calls

Provides or alters the information stored in the structure that describes the resource indexed by *id*.

```
#include <sys/msg.h>
struct msqid_ds buf;
err = msgctl (id, IPC_STAT, &buf);
if (err)
        !$#%*
else
        printf ("creator uid = %d\n", buf.msg_perm.cuid);
        ....
```

Commands supported by all `ctl` calls:

- IPC_STAT : read info on resource specified by id into user allocated buffer. The user must have read access to the resource.
- IPC_SET : write info from buffer into resource data structure. The user must be owner creator or super-user.
- IPC_RMID : remove resource. The user must be the owner, creator or super-user.

The IPC_RMID command results in immediate removal of a message queue or semaphore array. Shared memory segments however, are only destroyed upon the last detach after IPC_RMID is executed.

The `semctl` call provides a number of command options that allow the user to determine or set the values of the semaphores in an array.

See Also: See Section 1.5.4 [msgctl], page 7, See Section 1.6.3 [semctl], page 10, See Section 1.7.4 [shmctl], page 14.

### 1.4.3 The *op* system calls

Used to send or receive messages, read or alter semaphore values, attach or detach shared memory segments. The IPC_NOWAIT flag will cause the operation to fail with error EAGAIN if the process has to wait on the call.

`flags` : IPC_NOWAIT => return with error if a wait is required.

See Also: See Section 1.5.2 [msgsnd], page 6,See Section 1.5.3 [msgrcv], page 6,See Section 1.6.2 [semop], page 9,See Section 1.7.2 [shmat], page 13, See Section 1.7.3 [shmdt], page 14.

## 1.5 Messages

A message resource is described by a struct `msqid_ds` which is allocated and initialized when the resource is created. Some fields in `msqid_ds` can then be altered (if desired) by invoking `msgctl`. The memory used by the resource is released when it is destroyed by a `msgctl` call.

```
struct msqid_ds
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue (internal) */
    struct msg *msg_last;   /* last message in queue (internal) */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait; /* writers waiting (internal) */
    struct wait_queue *rwait; /* readers waiting (internal) */
    ushort msg_cbytes;      /* number of bytes used on queue */
    ushort msg_qnum;        /* number of messages in queue */
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* pid of last msgrcv */
```

To send or receive a message the user allocates a structure that looks like a `msgbuf` but with an array `mtext` of the required size. Messages have a type (positive integer) associated with them so that (for example) a listener can choose to receive only messages of a given type.

```
struct msgbuf
    long mtype;      type of message (See Section 1.5.3 [msgrcv], page 6).
    char mtext[1];   message text .. why is this not a ptr?
```

The user must have write permissions to send and read permissions to receive messages on a queue.

When `msgsnd` is invoked, the user's message is copied into an internal struct `msg` and added to the queue. A `msgrcv` will then read this message and free the associated struct `msg`.

### 1.5.1 msgget

A message queue is allocated by a msgget system call :

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

- `key`: an integer usually got from `ftok()` or IPC_PRIVATE.
- `msgflg`:

    IPC_CREAT : used to create a new resource if it does not already exist.

    IPC_EXCL | IPC_CREAT : used to ensure failure of the call if the resource already exists.

    rwxrwxrwx : access permissions.
- returns: msqid (an integer used for all further access) on success. -1 on failure.



A message queue is allocated if there is no resource corresponding to the given key. The access permissions specified are then copied into the `msg_perm` struct and the fields in `msqid_ds` initialized. The user must use the IPC_CREAT flag or key = IPC_PRIVATE, if a new instance is to be allocated. If a resource corresponding to *key* already exists, the access permissions are verified.

Errors:
EACCES : (procure) Do not have permission for requested access.
EEXIST : (allocate) IPC_CREAT | IPC_EXCL specified and resource exists.
EIDRM : (procure) The resource was removed.
ENOSPC : All id's are taken (max of MSGMNI id's system-wide).
ENOENT : Resource does not exist and IPC_CREAT not specified.
ENOMEM : A new `msqid_ds` was to be created but ... nomem.

### 1.5.2 msgsnd

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

- `msqid` : id obtained by a call to msgget.
- `msgsz` : size of msg text (`mtext`) in bytes.
- `msgp` : message to be sent. (msgp->mtype must be positive).
- `msgflg` : IPC_NOWAIT.
- returns : msgsz on success. -1 on error.

The message text and type are stored in the internal `msg` structure. `msg_cbytes`, `msg_qnum`, `msg_lspid`, and `msg_stime` fields are updated. Readers waiting on the queue are awakened.

Errors:
EACCES : Do not have write permission on queue.
EAGAIN : IPC_NOWAIT specified and queue is full.
EFAULT : msgp not accessible.
EIDRM : The message queue was removed.
EINTR : Full queue ... would have slept but ... was interrupted.
EINVAL : mtype < 1, msgsz > MSGMAX, msgsz < 0, msqid < 0 or unused.
ENOMEM : Could not allocate space for header and text.

### 1.5.3 msgrcv

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

- msqid : id obtained by a call to msgget.
- msgsz : maximum size of message to receive.
- msgp : allocated by user to store the message in.
- msgtyp :

    0 => get first message on queue.

    > 0 => get first message of matching type.

< 0 => get message with least type which is <= abs(msgtyp).

- msgflg :

  IPC_NOWAIT : Return immediately if message not found.

  MSG_NOERROR : The message is truncated if it is larger than msgsz.

  MSG_EXCEPT : Used with msgtyp > 0 to receive any msg except of specified type.

- returns : size of message if found. -1 on error.

The first message that meets the `msgtyp` specification is identified. For msgtyp < 0, the entire queue is searched for the message with the smallest type.

If its length is smaller than msgsz or if the user specified the MSG_NOERROR flag, its text and type are copied to msgp->mtext and msgp->mtype, and it is taken off the queue.

The `msg_cbytes`, `msg_qnum`, `msg_lrpid`, and `msg_rtime` fields are updated. Writers waiting on the queue are awakened.

Errors:
E2BIG : msg bigger than msgsz and MSG_NOERROR not specified.
EACCES : Do not have permission for reading the queue.
EFAULT : msgp not accessible.
EIDRM : msg queue was removed.
EINTR : msg not found ... would have slept but ... was interrupted.
EINVAL : msgsz > msgmax or msgsz < 0, msqid < 0 or unused.
ENOMSG : msg of requested type not found and IPC_NOWAIT specified.

### 1.5.4 msgctl

```
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

- msqid : id obtained by a call to msgget.
- buf : allocated by user for reading/writing info.
- cmd : IPC_STAT, IPC_SET, IPC_RMID (See Section 1.4 [syscalls], page 3).

IPC_STAT results in the copy of the queue data structure into the user supplied buffer.

In the case of IPC_SET, the queue size (`msg_qbytes`) and the `uid`, `gid`, `mode` (low 9 bits) fields of the `msg_perm` struct are set from the user supplied values. `msg_ctime` is updated.

Note that only the super user may increase the limit on the size of a message queue beyond MSGMNB.

When the queue is destroyed (IPC_RMID), the sequence number is incremented and all waiting readers and writers are awakened. These processes will then return with `errno` set to EIDRM.

Errors: EPERM : Insufficient privilege to increase the size of the queue (IPC_SET) or remove it (IPC_RMID).
EACCES : Do not have permission for reading the queue (IPC_STAT).
EFAULT : buf not accessible (IPC_STAT, IPC_SET).
EIDRM : msg queue was removed.
EINVAL : invalid cmd, msqid < 0 or unused.



### 1.5.5 Limis on Message Resources

Sizeof various structures:

msqid_ds 52 /* 1 per message queue .. dynamic */
msg 16 /* 1 for each message in system .. dynamic */
msgbuf 8 /* allocated by user */

Limits

- MSGMNI : number of message queue identifiers ... policy.
- MSGMAX : max size of message. Header and message space allocated on one page. MSGMAX = (PAGE_SIZE - sizeof(struct msg)). Implementation maximum MSGMAX = 4080.
- MSGMNB : default max size of a message queue ... policy. The super-user can increase the size of a queue beyond MSGMNB by a `msgctl` call.

Unused or unimplemented:
MSGTQL max number of message headers system-wide.
MSGPOOL total size in bytes of msg pool.

## 1.6 Semaphores

Each semaphore has a value >= 0. An id provides access to an array of `nsems` semaphores. Operations such as read, increment or decrement semaphores in a set are performed by the `semop` call which processes `nsops` operations at a time. Each operation is specified in a struct `sembuf` described below. The operations are applied only if all of them succeed.

If you do not have a need for such arrays, you are probably better off using the `test_bit`, `set_bit` and `clear_bit` bit-operations defined in <asm/bitops.h>.

Semaphore operations may also be qualified by a SEM_UNDO flag which results in the operation being undone when the process exits.

If a decrement cannot go through, a process will be put to sleep on a queue waiting for the `semval` to increase unless it specifies IPC_NOWAIT. A read operation can similarly result in a sleep on a queue waiting for `semval` to become 0. (Actually there are two queues per semaphore array).

A semaphore array is described by:

```
struct semid_ds
  struct ipc_perm sem_perm;
  time_t          sem_otime;      /* last semop time */
  time_t          sem_ctime;      /* last change time */
  struct wait_queue *eventn;   /* wait for a semval to increase */
  struct wait_queue *eventz;      /* wait for a semval to become 0 */
  struct sem_undo  *undo;         /* undo entries */
  ushort          sem_nsems;      /* no. of semaphores in array */
```

Each semaphore is described internally by :

```
struct sem
  short   sempid;         /* pid of last semop() */
```

```
    ushort  semval;        /* current value */
    ushort  semncnt;       /* num procs awaiting increase in semval */
    ushort  semzcnt;       /* num procs awaiting semval = 0 */
```

### 1.6.1 semget

A semaphore array is allocated by a semget system call:

`semid = semget (key_t key, int nsems, int semflg);`

- `key` : an integer usually got from `ftok` or IPC_PRIVATE
- `nsems` :

  # of semaphores in array (0 <= nsems <= SEMMSL <= SEMMNS)

  0 => dont care can be used when not creating the resource. If successful you always get access to the entire array anyway.
- semflg :

  IPC_CREAT used to create a new resource

  IPC_EXCL used with IPC_CREAT to ensure failure if the resource exists.

  rwxrwxrwx access permissions.
- returns : semid on success. -1 on failure.

An array of nsems semaphores is allocated if there is no resource corresponding to the given key. The access permissions specified are then copied into the `sem_perm` struct for the array along with the user-id etc. The user must use the IPC_CREAT flag or key = IPC_PRIVATE if a new resource is to be created.

Errors:
EINVAL : nsems not in above range (allocate).
nsems greater than number in array (procure).
EEXIST : (allocate) IPC_CREAT | IPC_EXCL specified and resource exists.
EIDRM : (procure) The resource was removed.
ENOMEM : could not allocate space for semaphore array.
ENOSPC : No arrays available (SEMMNI), too few semaphores available (SEMMNS).
ENOENT : Resource does not exist and IPC_CREAT not specified.
EACCES : (procure) do not have permission for specified access.

### 1.6.2 semop

Operations on semaphore arrays are performed by calling semop :

`int semop (int semid, struct sembuf *sops, unsigned nsops);`

- semid : id obtained by a call to semget.
- sops : array of semaphore operations.
- nsops : number of operations in array (0 < nsops < SEMOPM).
- returns : semval for last operation. -1 on failure.

Operations are described by a structure sembuf:



```
struct sembuf
    ushort  sem_num;       /* semaphore index in array */
    short   sem_op;        /* semaphore operation */
    short   sem_flg;       /* operation flags */
```

The value `sem_op` is to be added (signed) to the current value semval of the semaphore with index sem_num (0 .. nsems -1) in the set. Flags recognized in sem_flg are IPC_NOWAIT and SEM_UNDO.

Two kinds of operations can result in wait:

1. If sem_op is 0 (read operation) and semval is non-zero, the process sleeps on a queue waiting for semval to become zero or returns with error EAGAIN if (IPC_NOWAIT | sem_flg) is true.
2. If (sem_op < 0) and (semval + sem_op < 0), the process either sleeps on a queue waiting for semval to increase or returns with error EAGAIN if (sem_flg & IPC_NOWAIT) is true.

The array sops is first read in and preliminary checks performed on the arguments. The operations are parsed to determine if any of them needs write permissions or requests an undo operation.

The operations are then tried and the process sleeps if any operation that does not specify IPC_NOWAIT cannot go through. If a process sleeps it repeats these checks on waking up. If any operation that requests IPC_NOWAIT, cannot go through at any stage, the call returns with errno set to EAGAIN.

Finally, operations are committed when all go through without an intervening sleep. Processes waiting on the zero_queue or increment_queue are awakened if any of the semval's becomes zero or is incremented respectively.

Errors:
E2BIG : nsops > SEMOPM.
EACCES : Do not have permission for requested (read/alter) access.
EAGAIN : An operation with IPC_NOWAIT specified could not go through.
EFAULT : The array sops is not accessible.
EFBIG : An operation had semnum >= nsems.
EIDRM : The resource was removed.
EINTR : The process was interrupted on its way to a wait queue.
EINVAL : nsops is 0, semid < 0 or unused.
ENOMEM : SEM_UNDO requested. Could not allocate space for undo structure.
ERANGE : sem_op + semval > SEMVMX for some operation.

### 1.6.3 semctl

`int semctl (int semid, int semnum, int cmd, union semun arg);`

- semid : id obtained by a call to semget.
- cmd :

  GETPID return pid for the process that executed the last semop.

  GETVAL return semval of semaphore with index semnum.

  GETNCNT return number of processes waiting for semval to increase.

GETZCNT return number of processes waiting for semval to become 0

SETVAL set semval = arg.val.

GETALL read all semval's into arg.array.

SETALL set all semval's with values given in arg.array.

- returns : 0 on success or as given above. -1 on failure.

The first 4 operate on the semaphore with index semnum in the set. The last two operate on all semaphores in the set.

`arg` is a union :

```
union semun
    int val;                value for SETVAL.
    struct semid_ds *buf;   buffer for IPC_STAT and IPC_SET.
    ushort *array;          array for GETALL and SETALL
```

- IPC_SET, SETVAL, SETALL : sem_ctime is updated.
- SETVAL, SETALL : Undo entries are cleared for altered semaphores in all processes. Processes sleeping on the wait queues are awakened if a semval becomes 0 or increases.
- IPC_SET : sem_perm.uid, sem_perm.gid, sem_perm.mode are updated from user supplied values.

Errors: EACCES : do not have permission for specified access.
EFAULT : arg is not accessible.
EIDRM : The resource was removed.
EINVAL : semid < 0 or semnum < 0 or semnum >= nsems.
EPERM : IPC_RMID, IPC_SET ... not creator, owner or super-user.
ERANGE : arg.array[i].semval > SEMVMX or < 0 for some i.

### 1.6.4 Limits on Semaphore Resources

Sizeof various structures:

```
semid_ds    44   /* 1 per semaphore array .. dynamic */
sem          8   /* 1 for each semaphore in system .. dynamic */
sembuf       6   /* allocated by user */
sem_undo    20   /* 1 for each undo request .. dynamic */
```

Limits :

- SEMVMX 32767 semaphore maximum value (short).
- SEMMNI number of semaphore identifiers (or arrays) system wide...policy.
- SEMMSL maximum number of semaphores per id. 1 semid_ds per array, 1 struct sem per semaphore => SEMMSL = (PAGE_SIZE - sizeof(semid_ds)) / sizeof(sem). Implementation maximum SEMMSL = 500.
- SEMMNS maximum number of semaphores system wide ... policy. Setting SEMMNS >= SEMMSL*SEMMNI makes it irrelevent.
- SEMOPM Maximum number of operations in one semop call...policy.



Unused or unimplemented:
SEMAEM adjust on exit max value.
SEMMNU number of undo structures system-wide.
SEMUME maximum number of undo entries per process.

## 1.7 Shared Memory

Shared memory is distinct from the sharing of read-only code pages or the sharing of unaltered data pages that is available due to the copy-on-write mechanism. The essential difference is that the shared pages are dirty (in the case of Shared memory) and can be made to appear at a convenient location in the process' address space.

A shared segment is described by :

```
struct shmid_ds
    struct  ipc_perm shm_perm;
    int     shm_segsz;              /* size of segment (bytes) */
    time_t  shm_atime;              /* last attach time */
    time_t  shm_dtime;              /* last detach time */
    time_t  shm_ctime;              /* last change time */
    ulong   *shm_pages;             /* internal page table */
    ushort  shm_cpid;               /* pid, creator */
    ushort  shm_lpid;               /* pid, last operation */
    short   shm_nattch;             /* no. of current attaches */
```

A shmget allocates a shmid_ds and an internal page table. A shmat maps the segment into the process' address space with pointers into the internal page table and the actual pages are faulted in as needed. The memory associated with the segment must be explicitly destroyed by calling shmctl with IPC_RMID.

### 1.7.1 shmget

A shared memory segment is allocated by a shmget system call:

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

- key : an integer usually got from `ftok` or IPC_PRIVATE
- size : size of the segment in bytes (SHMMIN <= size <= SHMMAX).
- shmflg :

  IPC_CREAT used to create a new resource

  IPC_EXCL used with IPC_CREAT to ensure failure if the resource exists.

  rwxrwxrwx access permissions.
- returns : shmid on success. -1 on failure.

A descriptor for a shared memory segment is allocated if there isn't one corresponding to the given key. The access permissions specified are then copied into the `shm_perm` struct for the segment along with the user-id etc. The user must use the IPC_CREAT flag or key = IPC_PRIVATE to allocate a new segment.

If the segment already exists, the access permissions are verified, and a check is made to see that it is not marked for destruction.

`size` is effectively rounded up to a multiple of PAGE_SIZE as shared memory is allocated in pages.

Errors:
EINVAL : (allocate) Size not in range specified above.
(procure) Size greater than size of segment.
EEXIST : (allocate) IPC_CREAT | IPC_EXCL specified and resource exists.
EIDRM : (procure) The resource is marked destroyed or was removed.
ENOSPC : (allocate) All id's are taken (max of SHMMNI id's system-wide). Allocating a segment of the requested size would exceed the system wide limit on total shared memory (SHMALL).
ENOENT : (procure) Resource does not exist and IPC_CREAT not specified.
EACCES : (procure) Do not have permission for specified access.
ENOMEM : (allocate) Could not allocate memory for shmid_ds or pg_table.

### 1.7.2 shmat

Maps a shared segment into the process' address space.

```
char *virt_addr;
virt_addr =  shmat (int shmid, char *shmaddr, int shmflg);
```

- shmid : id got from call to shmget.
- shmaddr : requested attach address.
  If shmaddr is 0 the system finds an unmapped region.
  If a non-zero value is indicated the value must be page aligned or the user must specify the SHM_RND flag.
- shmflg :
  SHM_RDONLY : request read-only attach.
  SHM_RND : attach address is rounded DOWN to a multiple of SHMLBA.
- returns: virtual address of attached segment. -1 on failure.

When shmaddr is 0, the attach address is determined by finding an unmapped region in the address range 1G to 1.5G, starting at 1.5G and coming down from there. The algorithm is very simple so you are encouraged to avoid non-specific attaches.

Algorithm:
Determine attach address as described above.
Check region (shmaddr, shmaddr + size) is not mapped and allocate
page tables (undocumented SHM_REMAP flag!).
Map the region by setting up pointers into the internal page table.
Add a descriptor for the attach to the task struct for the process.
`shm_nattch`, `shm_lpid`, `shm_atime` are updated.

Notes:
The `brk` value is not altered. The segment is automatically detached when the process exits. The same segment may be attached as read-only or read-write and more than once in the process' address space. A shmat can succeed on a segment marked for destruction. The request for a particular type of attach is made using the SHM_RDONLY flag. There is no notion of a write-only attach. The requested attach permissions must fall within those allowed by `shm_perm.mode`.



Errors:
EACCES : Do not have permission for requested access.
EINVAL : shmid < 0 or unused, shmaddr not aligned, attach at brk failed.
EIDRM : resource was removed.
ENOMEM : Could not allocate memory for descriptor or page tables.

### 1.7.3 shmdt

```
int shmdt (char *shmaddr);
```

- shmaddr : attach address of segment (returned by shmat).
- returns : 0 on success. -1 on failure.

An attached segment is detached and `shm_nattch` decremented. The occupied region in user space is unmapped. The segment is destroyed if it is marked for destruction and `shm_nattch` is 0. `shm_lpid` and `shm_dtime` are updated.

Errors:
EINVAL : No shared memory segment attached at shmaddr.

### 1.7.4 shmctl

Destroys allocated segments. Reads/Writes the control structures.

```
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

- shmid : id got from call to shmget.
- cmd : IPC_STAT, IPC_SET, IPC_RMID (See Section 1.4 [syscalls], page 3).
  - IPC_SET : Used to set the owner uid, gid, and shm_perms.mode field.
  - IPC_RMID : The segment is marked destroyed. It is only destroyed on the last detach.
  - IPC_STAT : The shmid_ds structure is copied into the user allocated buffer.
- buf : used to read (IPC_STAT) or write (IPC_SET) information.
- returns : 0 on success, -1 on failure.

The user must execute an IPC_RMID shmctl call to free the memory allocated by the shared segment. Otherwise all the pages faulted in will continue to live in memory or swap.

Errors:
EACCES : Do not have permission for requested access.
EFAULT : buf is not accessible.
EINVAL : shmid < 0 or unused.
EIDRM : identifier destroyed.
EPERM : not creator, owner or super-user (IPC_SET, IPC_RMID).

### 1.7.5 Limits on Shared Memory Resources

Limits:

- SHMMNI max num of shared segments system wide ... 4096.
- SHMMAX max shared memory segment size (bytes) ... 4M

- SHMMIN min shared memory segment size (bytes). 1 byte (though PAGE_SIZE is the effective minimum size).
- SHMALL max shared mem system wide (in pages) ... policy.
- SHMLBA segment low boundary address multiple. Must be page aligned. SHMLBA = PAGE_SIZE.

Unused or unimplemented:
SHMSEG : maximum number of shared segments per process.

## 1.8 Miscellaneous Notes

The system calls are mapped into one – `sys_ipc`. This should be transparent to the user.

### 1.8.1 Semaphore `undo` requests

There is one sem_undo structure associated with a process for each semaphore which was altered (with an undo request) by the process. `sem_undo` structures are freed only when the process exits.

One major cause for unhappiness with the undo mechanism is that it does not fit in with the notion of having an atomic set of operations on an array. The undo requests for an array and each semaphore therein may have been accumulated over many `semop` calls. Thus use the undo mechanism with private semaphores only.

Should the process sleep in `exit` or should all undo operations be applied with the IPC_NOWAIT flag in effect? Currently those undo operations which go through immediately are applied and those that require a wait are ignored silently.

### 1.8.2 Shared memory, `malloc` and the `brk`.

Note that since this section was written the implementation was changed so that non-specific attaches are done in the region 1G - 1.5G. However much of the following is still worth thinking about so I left it in.

On many systems, the shared memory is allocated in a special region of the address space ... way up somewhere. As mentioned earlier, this implementation attaches shared segments at the lowest possible address. Thus if you plan to use `malloc`, it is wise to malloc a large space and then proceed to attach the shared segments. This way malloc sets the brk sufficiently above the region it will use.

Alternatively you can use `sbrk` to adjust the `brk` value as you make shared memory attaches. The implementation is not very smart about selecting attach addresses. Using the system default addresses will result in fragmentation if detaches do not occur in the reverse sequence as attaches.

Taking control of the matter is probably best. The rule applied is that attaches are allowed in unmapped regions other than in the text space (see <a.out.h>). Also remember that attach addresses and segment sizes are multiples of PAGE_SIZE.

One more trap (I quote Bruno on this). If you use malloc() to get space for your shared memory (ie. to fix the `brk`), you must ensure you get an unmapped address range. This means you must mallocate more memory than you had ever allocated before. Memory returned by malloc(), used, then freed by free() and then again returned by malloc is no good. Neither is calloced memory.



Note that a shared memory region remains a shared memory region until you unmap it. Attaching a segment at the `brk` and calling malloc after that will result in an overlap of what malloc thinks is its space with what is really a shared memory region. For example in the case of a read-only attach, you will not be able to write to the overlapped portion.

### 1.8.3 Fork, exec and exit

On a fork, the child inherits attached shared memory segments but not the semaphore undo information.

In the case of an exec, the attached shared segments are detached. The sem undo information however remains intact.

Upon exit, all attached shared memory segments are detached. The adjust values in the undo structures are added to the relevant semvals if the operations are permitted. Disallowed operations are ignored.

### 1.8.4 Other Features

These features of the current implementation are likely to be modified in the future.

The SHM_LOCK and SHM_UNLOCK flag are available (super-user) for use with the `shmctl` call to prevent swapping of a shared segment. The user must fault in any pages that are required to be present after locking is enabled.

The IPC_INFO, MSG_STAT, MSG_INFO, SHM_STAT, SHM_INFO, SEM_STAT, SEMINFO `ctl` calls are used by the `ipcs` program to provide information on allocated resources. These can be modified as needed or moved to a proc file system interface.

Thanks to Ove Ewerlid, Bruno Haible, Ulrich Pegelow and Linus Torvalds for ideas, tutorials, bug reports and fixes, and merriment. And more thanks to Bruno.

# Table of Contents