

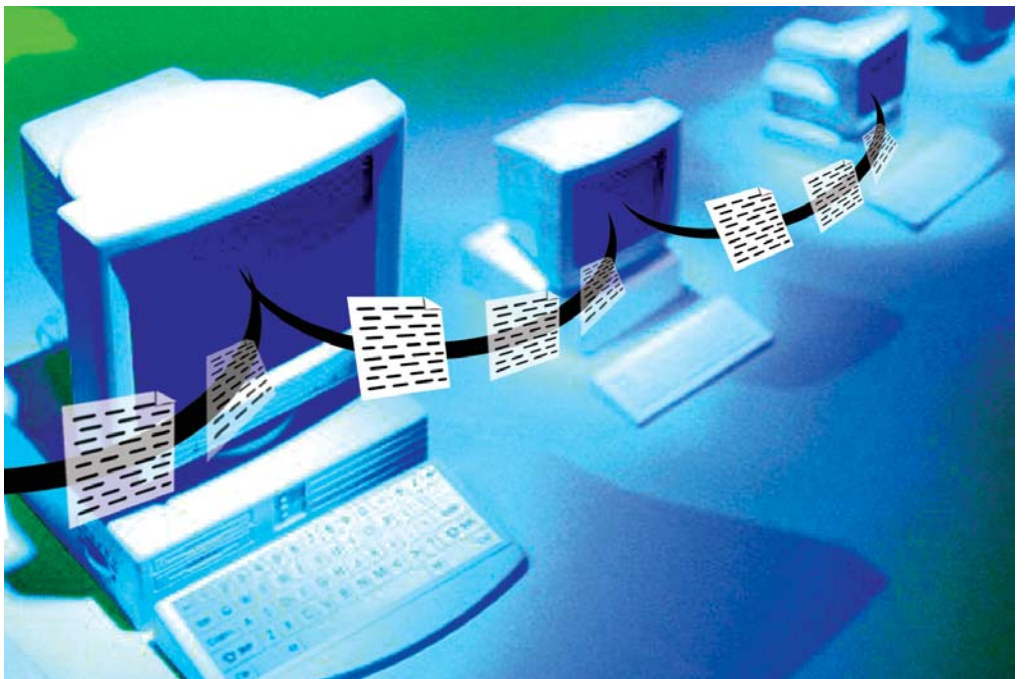
Sharing Data with Shared Memory

Shared memory, an important IPC mechanism, saves memory by sharing data between different processes. It's also a fast communication mechanism, since it incurs no overhead in moving data from the user to the kernel buffer, and vice versa.

Shared memory is an important IPC mechanism. It is useful since it saves memory when accessing data—by sharing it between different processes. This minimises the overhead of memory consumption and saves processor time. This is the fastest communication mechanism in System V IPC, since the shared memory is directly mapped into the calling process' address space. Shared memory is used to provide access to global variables, shared

libraries, HTTP daemons and other programs written in languages like Perl, C, etc. It also helps in providing access to executable programs containing text and data segments. Text segments contain the executable code of a program. All processes of the program share this portion of the memory. For example, if you initialise your program ten times, the text segment for the program needs to be loaded only once, but the data segment is unique for each process.

This article first explains the relevant data



structures used by shared memory. A shared memory segment is created by the *shmget* system call. Then the segment is attached by using the *shmat* system call. The system calls used for shared memory are discussed in a later section of this article, along with a demo program. The internals of the system calls, in this article, have been explained with the help of source code of the kernel version 2.6.14. To minimise complexity, the whole code is not discussed. This should help you understand shared memory implementation in some detail.

Shared memory data structures

The data structures used in shared memory are *shmid_ds*, *ipc_perm*, *shminfo*, *shm_info* and *shmid_kernel*. These data structures are located in the kernel. These structures are defined in the `<linux/shm.h>` file. Listing 1 shows the members of these structures. Since the *ipc_perm* structure is just similar to the one used in message queues, it is not taken up for discussion. The details of two other structures—*shmid_ds* and *shm_info*—should be clear from Listing 1. As in the case of message queues, the kernel uses the *shmid_ds* structure to maintain an internal data structure for every shared memory segment created. The *shminfo* structure has five members—*shmmax*, which specifies the maximum shared memory segment size in bytes; *shmin*, which specifies the minimum shared memory segment size in bytes; *shmmni*, which specifies the maximum number of shared memory identifiers; *shmseg*, which specifies the maximum shared memory segments per process and *shmalls*, which specifies the maximum amount of shared memory in pages. The

```
File Edit View Terminal Go Help
struct shmid_ds
{
    struct ipc_perm shm_perm;
    size_t shm_segsz; /* size of segment in bytes */
    __time_t shm_atime; /* time of last shmat() */
    __time_t shm_dtime; /* time of last shmdt() */
    __time_t shm_ctime; /* time of last change by shmctl() */
    __pid_t shm_cpid; /* pid of creator */
    __pid_t shm_lpid; /* pid of last shmop */
    shmatt_t shm_nattch; /* number of current attaches */
};
struct shminfo
{
    unsigned long int shmmax;
    unsigned long int shmin;
    unsigned long int shmmni;
    unsigned long int shmseg;
    unsigned long int shmalls;
};
struct shm_info
{
    int used_ids;
    unsigned long int shm_tot; /* total allocated shm */
    unsigned long int shm_rss; /* total resident shm */
    unsigned long int shm_swp; /* total swapped shm */
    unsigned long int swap_attempts;
    unsigned long int swap_successes;
};
5,1 All
```

Listing 1: The *shmid_ds*, *shminfo* and *shm_info* structures

```
File Edit View Terminal Go Help
1 #include <sys/shm.h>
2 main () {
3     int key, shmid, choice;
4     char *ptr;
5
6     key = ftok(".", 'a');
7     shmid = shmget (key, 1024, IPC_CREAT|0744);
8     ptr = shmat(shmid, (void *)0, 0);
9
10    printf (" \n\n \n \t\t Menu driven program for shared memory \n");
11    printf (" \n\n \t\t\t Reading: 1\n \t\t\t writing: 2\n");
12
13    printf ("Enter your choice: ");
14    scanf ("%d", &choice);
15
16    switch (choice)
17    {
18        case 1:
19            printf (" Reading shared memory segment: %s\n", ptr);
20            break;
21
22        case 2:
23            printf (" Enter: ");
24            scanf ("%[^\n]", ptr);
25            break;
26    }
27 }
21,1 All
```

Listing 2: C program to create and access a shared memory segment

shmid_kernel structure is used to maintain kernel mappings for memory segments. It holds the members of the *kern_ipc_perm* structure, the file structure, ID, the number of attachments, the segment size, the last attach time, the last detach time, the last change time, the pid of the creator, the pid of the last user and the *user_struct* structure.

System calls related to shared memory

Some calls, which are of great importance in a shared memory environment, are listed below.

shmget: If one program can create a shared memory portion, then other programs having adequate access permissions can access this shared memory segment. The *shmget* system call is used to create a shared memory segment. The syntax of the system call is shown below:

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

It accepts three arguments—key, size and *shmflg*. The key and *shmflg* are same as the ones used in the *msgget* system call. The size argument is used to specify the size of the shared memory. On success, the system call returns the shared memory ID, or else it returns -1.

The *shmget* system call internally calls the *sys_shmget* function, which resides in `/usr/src/linux-2.6.14/ipc/shm.c` source file. It first checks the key to ascertain whether it has been declared as `IPC_PRIVATE`. If so, it calls the *newseg* function.

```
asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
{
    struct shmid_kernel *shp;
    .....
    if (key == IPC_PRIVATE) {
        err = newseg(key, shmflg, size);
```

```

    } else if ((id = ipc_findkey(&shm_ids, key)) == -1) {
        if (!(shmflg & IPC_CREAT))
            err = -ENOENT;
        else
            err = newseg(key, shmflg, size);
    } else if ((shmflg & IPC_CREAT) && (shmflg &
IPC_EXCL)) {
        err = -EEXIST;
        .....
        if (shp->shm_segsz < size)
            err = -EINVAL;
        else if (ipcperms(&shp->shm_perm, shmflg))
            err = -EACCES;
        else {
            int shmid = shm_buildid(id,
shp->shm_perm.seq);
            .....
            err = shmid;
            return err;
        }
    }
}

```

The *newseg* function is used to create and initialise a new, shared memory segment. If *IPC_PRIVATE* is not declared as a key, then it checks whether the key already exists. If not, then it will check *shmflg* and, according to the flag, take necessary action. As the next step, it will check the size of the shared memory segment and the access permission. If everything goes well, it will create a shared memory segment by calling the *shm_buildid* function. On success, the function returns the shared memory ID.

If a process has a shared memory identifier, then the next step is to attach (or map) the shared memory segment onto its own address space.

shmop: There are two system calls that are used for shared memory operations—*shmat* and *shmdt*. The *shmat* system call is used to attach the created shared memory segment onto a process address space. A pointer is returned on the successful execution of the system call and the process can read or write to the segment using the pointer. The syntax of this system call is as follows:

```
void *shmat (int shmid, const void *shmaddr, int shmflg );
```

shmat attaches the given memory segment identified by *shmid* to the address space of the calling process. If the second argument *shmaddr* is 0, then the kernel will allocate an unmapped memory region in the physical memory. The last argument, *shmflg*, is used to specify whether the shared memory segment is used for read-only or for both reading and writing. If it has read-write access, then we need to pass 0 as the value—otherwise *SHM_RDONLY* for the shared memory segment has to be marked as read-only. The return value of *shmat* is *void *pointer*—the pointer to void is actually part of the ANSI-C standard; and it can be assigned the value of any pointer

type. Using the *shmat* system call, many processes can attach a shared segment of physical memory to their virtual address space.

shmat calls the *sys_shmat* function, which, in turn, calls the *do_shmat* function.

```

asmlinkage long sys_shmat(int shmid, char __user *shmaddr, int
shmflg)
{
    unsigned long ret;
    long err;

    err = do_shmat(shmid, shmaddr, shmflg, &ret);
    if (err)
        return err;
    force_successful_syscall_return();
    return (long)ret;
}

long do_shmat(int shmid, char __user *shmaddr, int shmflg,
ulong *raddr)
{
    if (shmflg & SHM_RDONLY) {
        prot = PROT_READ;
        o_flags = O_RDONLY;
        acc_mode = S_IRUGO;
    } else {
        prot = PROT_READ | PROT_WRITE;
        o_flags = O_RDWR;
        acc_mode = S_IRUGO | S_IWUGO;
    }
    err = shm_checkid(shp, shmid);
    if (ipcperms(&shp->shm_perm, acc_mode)) {
        shp->shm_nattch++;
        user_addr = (void*) do_mmap (file, addr, size, prot, flags,
0);
        return user_addr;
    }
}

```

The *do_shmat* function checks the shared memory flag, which is passed through the *shmat* system call. If the flag is *SHM_RDONLY*, it sets the opening and access mode as read-only. If the flag value is 0, then it sets the opening and access mode as read-write. Then it validates the shared memory ID, the access permission and increments the number of attachments. The creation of a virtual memory mapping to the shared memory segment pages is done, by calling the *do_mmap* function. On successful execution of the *do_mmap* function, it returns the virtual address at which the shared memory is mapped. The *prot* argument in the *do_mmap* describes the state of memory protection; and it must agree with the accessing mode.

The detachment of an attached shared memory segment is done by *shmdt* to pass the address of the pointer as an argument.

```
int shmdt (const void *shmaddr );
```

The *shmdt* system call calls the *sys_symdt* function:

```
asmlinkage long sys_shmdt(char __user *shmaddr)
```

This function unmaps the shared memory segment by calling the *do_munmap* function. Before calling the *do_munmap* function, the *sys_shmdt* function searches the *vm_area_struct* associated with the shared memory address.

```
do_munmap(nm, vma->vm_start, vma->vm_end - vma->vm_start);
```

The arguments of the functions are the starting address and the size. This function removes the virtual address mapping for the shared memory segment and calls the *shm_close* function.

shm_close updates the *shm_lprid* and the *shm_dtim* fields, and decrements the number of attached shared memory segments.

```
static void shm_close (struct vm_area_struct *shmd)
{
    shp->shm_lprid = current->tgid;
    shp->shm_dtim = get_seconds();
    shp->shm_nattach--;
    if(shp->shm_nattach == 0 &&
        shp->shm_flags & SHM_DEST)
        shm_destroy (shp);
    .....
}
```

The SHM_DEST flag (defined in <linux/shm.h>) is described as the shared memory segment that is being destroyed on the last detach. After decrementing, if the number of attachments is 0, it calls the *shm_destroy* function to release the shared memory segment resources.

```
static void shm_destroy (struct shmid_kernel *shp)
{
    shm_tot -= (shp->shm_segsz + PAGE_SIZE - 1) >>
    PAGE_SHIFT;
    shm_rmid (shp->id);
    .....
    security_shm_free(shp);
    .....
}
```

The above function alters the total number of shared memory pages, and then removes the shared memory ID by calling the *shm_rmid* function. Finally, it frees the shared memory segment descriptor.

shmctl: To retrieve information about a shared memory segment, to modify the attributes of the segment

or to remove the allocated shared memory resources itself, we need to call the *shmctl* system call. This system call accepts three arguments—the shared memory ID, the command and the address of the *shmid_ds* structure.

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

Three commands are used in the system call—IPC_STAT, IPC_SET and IPC_RMID, which have been explained in a previous LFY article on *Message Queues*. To put it briefly, IPC_STAT copies the shared memory segment information into the buffer argument. IPC_SET sets the user given values into the system's *shmid_ds* structure. Once a shared memory segment is created, it will persist in the system till a reboot is done, or the *ipcrm* command is issued at the command line. The removal could also be effected by passing the IPC_RMID command in the *shmctl* system call. If we haven't removed the segment after its usage, it could unnecessarily create memory leaks in the system. IPC_RMID is used to mark the shared memory segment as destroyed, but the removal of shared memory segments is dependent on the value of the number of attachments. Apart from the above three generic commands, shared memory has two more specific commands—SHM_LOCK and SHM_UNLOCK. The pages in the shared memory are swappable. It may be swapped out to the swap space during periods of high memory usage; so if we specify the command SHM_LOCK, it prevents the swapping of a shared memory segment. Enabling SHM_UNLOCK makes segments swappable. Only the root user can perform this operation. To check the root privilege, the kernel source code uses a capability function.

```
if (!capable (CAP_SYS_ADMIN))
```

The *capable* function is used in kernel level programs to check the capability of the calling process. The main aim of this function is to restrict some of the privileged operations from the non-root users. If successful, the function returns 1; otherwise, it returns 0. The details of the CAP_SYS_ADMIN and the remaining list of the flags are in the <linux/capability.h> file.

shmctl calls the *sys_symctl* function and passes the shared memory ID, command and user buffer addresses.

```
asmlinkage long sys_shmctl (int shmid, int cmd, struct shmid_ds
    __user *buf)
```

The function first validates the given command and shared memory ID:

```
if (cmd < 0 || shmid < 0) {
    err = -EINVAL;
    .....
}
```


Now, depending upon the command, it will execute the corresponding case statement in the switch-case structure after checking the valid access permissions. If the command is IPC_STAT, then the shared memory information is copied into the temporary buffer and transferred into the user buffer address by calling:

```
copy_shmid_to_user (buf, &tbuf, version).
```

If the command is IPC_SET, then it updates the information from the user-given buffer structure by calling:

```
copy_shmid_from_user (&setbuf, buf, version).
```

When we pass IPC_RMID as a command in the *shmctl* function, it checks the ID and the number of attachments. If no other process is using the shared memory segment, it calls the *shm_destroy* function to remove the allocated shared memory resources. All these functions in the source code are synchronised using suitable global locking mechanisms.


```
case IPC_RMID:
{
    err = shm_checkid(shp, shmid);

    if (shp->shm_nattch){
        shp->shm_flags |= SHM_DEST;
        .....
    } else
```

```
shm_destroy (shp);
```

```
}
```

Shared memory reserves a given memory space for storing data, which can be accessed by more than one process. Among the available IPC mechanisms, shared memory is one of the fastest techniques to share data among processes in the Linux environment, since there is no overhead in moving data from the user to the kernel buffer and vice versa—when we write to or read from shared memory. After creating a shared memory segment by using the *shmget* system call, the *shmat* system call is used to attach the given shared memory segment with the physical memory. On success, the call returns a pointer.

Whenever a shared concept comes in, a race condition also strolls along. If more than one process wants to modify the shared memory, the race condition comes into force. To synchronise shared memory, some form of a synchronisation tool must be used as protection from the race condition. One of the System V IPC mechanisms is a semaphore. A semaphore is a synchronisation tool, which is used to protect shared data, shared resources or shared memory from concurrent access. We shall discuss the intricacies of System V semaphores in a future article. **END** 

By: Dr B. Thangaraju. The author is working as a technical lead, Talent Transformation, Wipro Technologies, Bangalore. He can be reached at balat.raju@wipro.com.
Acknowledgement: The author would like to thank Dr N. Kaulgud for the critical reviewing of this manuscript.

**Some of you will spend
THOUSANDS to learn OLD technologies**

**The smarter ones will
grab every opportunity to master the future**



Read LINUX For You

For more info, log on to:
www.linuxforu.com



ASIA'S FIRST
LINUX
MAGAZINE

LINUX
THE COMPLETE MAGAZINE ON OPEN SOURCE
ForYou