14 System V Shared Memory

14.1 Introduction

System V shared memory is similar in concept to Posix shared memory. Instead of calling shm_open followed by mmap, we call shmget followed by shmat.

For every shared memory segment, the kernel maintains the following structure of information, defined by including <sys/shm.h>:

```
struct shmid_ds {
  struct ipc_perm shm_perm;     /* operation permission struct */
  size_t          shm_segsz;    /* segment size */
  pid_t           shm_lpid;     /* pid of last operation */
  pid_t           shm_cpid;     /* creator pid */
  shmatt_t        shm_nattch;   /* current # attached */
  shmat_t         shm_cnattch;  /* in-core # attached */
  time_t          shm_atime;    /* last attach time */
  time_t          shm_dtime;    /* last detach time */
  time_t          shm_ctime;    /* last change time of this structure */
};
```

We described the ipc_perm structure in Section 3.3, and it contains the access permissions for the shared memory segment.

14.2 shmget Function

A shared memory segment is created, or an existing one is accessed, by the shmget function.

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

int shmget(key_t key, size_t size, int oflag);
```

Returns: shared memory identifier if OK, −1 on error

The return value is an integer called the *shared memory identifier* that is used with the three other shm*XXX* functions to refer to this segment.

key can be either a value returned by `ftok` or the constant `IPC_PRIVATE`, as discussed in Section 3.2.

size specifies the size of the segment, in bytes. When a new shared memory segment is created, a nonzero value for *size* must be specified. If an existing shared memory segment is being referenced, *size* should be 0.

oflag is a combination of the read–write permission values shown in Figure 3.6. This can be bitwise-ORed with either `IPC_CREAT` or `IPC_CREAT | IPC_EXCL`, as discussed with Figure 3.4.

When a new shared memory segment is created, it is initialized to *size* bytes of 0.

Note that `shmget` creates or opens a shared memory segment, but does not provide access to the segment for the calling process. That is the purpose of the `shmat` function, which we describe next.

14.3 `shmat` Function

After a shared memory segment has been created or opened by `shmget`, we attach it to our address space by calling `shmat`.

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

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

Returns: starting address of mapped region if OK, −1 on error

shmid is an identifier returned by `shmget`. The return value from `shmat` is the starting address of the shared memory segment within the calling process. The rules for determining this address are as follows:

- If *shmaddr* is a null pointer, the system selects the address for the caller. This is the recommended (and most portable) method.
- If *shmaddr* is a nonnull pointer, the returned address depends on whether the caller specifies the `SHM_RND` value for the *flag* argument:
  - If `SHM_RND` is not specified, the shared memory segment is attached at the address specified by the *shmaddr* argument.
  - If `SHM_RND` is specified, the shared memory segment is attached at the address specified by the *shmaddr* argument, rounded down by the constant `SHMLBA`. LBA stands for "lower boundary address."

By default, the shared memory segment is attached for both reading and writing by the calling process, if the process has read-write permissions for the segment. The SHM_RDONLY value can also be specified in the *flag* argument, specifying read-only access.

14.4 shmdt Function

When a process is finished with a shared memory segment, it detaches the segment by calling shmdt.

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

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

Returns: 0 if OK, –1 on error

When a process terminates, all shared memory segments currently attached by the process are detached.

Note that this call does not delete the shared memory segment. Deletion is accomplished by calling shmctl with a command of IPC_RMID, which we describe in the next section.

14.5 shmctl Function

shmctl provides a variety of operations on a shared memory segment.

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

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

Returns: 0 if OK, –1 on error

Three commands are provided:

IPC_RMID	Remove the shared memory segment identified by *shmid* from the system and destroy the shared memory segment.
IPC_SET	Set the following three members of the shmid_ds structure for the shared memory segment from the corresponding members in the structure pointed to by the *buff* argument: shm_perm.uid, shm_perm.gid, and shm_perm.mode. The shm_ctime value is also replaced with the current time.
IPC_STAT	Return to the caller (through the *buff* argument) the current shmid_ds structure for the specified shared memory segment.

14.6 Simple Programs

We now develop some simple programs that operate on System V shared memory.

shmget Program

Our shmget program, shown in Figure 14.1, creates a shared memory segment using a specified pathname and length.

svshm/shmget.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     c, id, oflag;
 6     char   *ptr;
 7     size_t  length;

 8     oflag = SVSHM_MODE | IPC_CREAT;
 9     while ( (c = Getopt(argc, argv, "e")) != -1) {
10         switch (c) {
11         case 'e':
12             oflag |= IPC_EXCL;
13             break;
14         }
15     }
16     if (optind != argc - 2)
17         err_quit("usage: shmget [ -e ] <pathname> <length>");
18     length = atoi(argv[optind + 1]);

19     id = Shmget(Ftok(argv[optind], 0), length, oflag);
20     ptr = Shmat(id, NULL, 0);

21     exit(0);
22 }
```

svshm/shmget.c

Figure 14.1 Create a System V shared memory segment of a specified size.

19 shmget creates the shared memory segment of the specified size. The pathname passed as a command-line argument is mapped into a System V IPC key by ftok. If the -e option is specified, it is an error if the segment already exists. If we know that the segment already exists, the length on the command line should be specified as 0.

20 shmat attaches the segment into the address space of the process. The program then terminates. Since System V shared memory has at least kernel persistence, this does not remove the shared memory segment.

shmrmid Program

Figure 14.2 shows our trivial program that calls shmctl with a command of IPC_RMID to remove a shared memory segment from the system.

svshm/shmrmid.c

```
#include    "unpipc.h"

int
main(int argc, char **argv)
{
    int     id;

    if (argc != 2)
        err_quit("usage: shmrmid <pathname>");

    id = Shmget(Ftok(argv[1], 0), 0, SVSHM_MODE);
    Shmctl(id, IPC_RMID, NULL);

    exit(0);
}
```

svshm/shmrmid.c

Figure 14.2 Remove a System V shared memory segment.

shmwrite Program

Figure 14.3 is our shmwrite program, which writes a pattern of 0, 1, 2, ..., 254, 255, 0, 1, and so on, to a shared memory segment.

svshm/shmwrite.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, id;
 6     struct shmid_ds buff;
 7     unsigned char *ptr;

 8     if (argc != 2)
 9         err_quit("usage: shmwrite <pathname>");

10     id = Shmget(Ftok(argv[1], 0), 0, SVSHM_MODE);
11     ptr = Shmat(id, NULL, 0);
12     Shmctl(id, IPC_STAT, &buff);

13     /* set: ptr[0] = 0, ptr[1] = 1, etc. */
14     for (i = 0; i < buff.shm_segsz; i++)
15         *ptr++ = i % 256;

16     exit(0);
17 }
```

svshm/shmwrite.c

Figure 14.3 Open a shared memory segment and fill it with a pattern.

10-12 The shared memory segment is opened by shmget and attached by shmat. We fetch its size by calling shmctl with a command of IPC_STAT.

13-15 The pattern is written to the shared memory.

shmread Program

Our shmread program, shown in Figure 14.4, verifies the pattern that was written by shmwrite.

svshm/shmread.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, id;
 6     struct shmid_ds buff;
 7     unsigned char c, *ptr;

 8     if (argc != 2)
 9         err_quit("usage: shmread <pathname>");

10     id = Shmget(Ftok(argv[1], 0), 0, SVSHM_MODE);
11     ptr = Shmat(id, NULL, 0);
12     Shmctl(id, IPC_STAT, &buff);

13     /* check that ptr[0] = 0, ptr[1] = 1, etc. */
14     for (i = 0; i < buff.shm_segsz; i++)
15         if ( (c = *ptr++) != (i % 256))
16             err_ret("ptr[%d] = %d", i, c);

17     exit(0);
18 }
```

svshm/shmread.c

Figure 14.4 Open a shared memory segment and verify its data pattern.

10-12 The shared memory segment is opened and attached. Its size is obtained by calling shmctl with a command of IPC_STAT.

13-16 The pattern written by shmwrite is verified.

Examples

We create a shared memory segment whose length is 1234 bytes under Solaris 2.6. The pathname used to identify the segment (e.g., the pathname passed to ftok) is the pathname of our shmget executable. Using the pathname of a server's executable file often provides a unique identifier for a given application.

```
solaris % shmget shmget 1234
solaris % ipcs -bmo
IPC status from <running system> as of Thu Jan  8 13:17:06 1998
T         ID      KEY        MODE        OWNER    GROUP NATTCH      SEGSZ
Shared Memory:
m          1   0x0000f12a --rw-r--r-- rstevens   other1      0       1234
```

We run the ipcs program to verify that the segment has been created. We notice that the number of attaches (which is stored in the shm_nattch member of the shmid_ds structure) is 0, as we expect.

Next, we run our `shmwrite` program to set the contents of the shared memory segment to the pattern. We verify the shared memory segment's contents with `shmread` and then remove the identifier.

```
solaris % shmwrite shmget
solaris % shmread shmget
solaris % shmrmid shmget
solaris % ipcs -bmo
IPC status from <running system> as of Thu Jan  8 13:18:01 1998
T        ID      KEY        MODE        OWNER    GROUP NATTCH      SEGSZ
Shared Memory:
```

We run `ipcs` to verify that the shared memory segment has been removed.

When the name of the server executable is used as an argument to `ftok` to identify some form of System V IPC, the absolute pathname would normally be specified, such as `/usr/bin/myserverd`, and not a relative pathname as we have used (`shmget`). We have been able to use a relative pathname for the examples in this section because all of the programs have been run from the directory containing the server executable. Realize that `ftok` uses the i-node of the file to form the IPC identifier (e.g., Figure 3.2), and whether a given file is referenced by an absolute pathname or by a relative pathname has no effect on the i-node.

14.7 Shared Memory Limits

As with System V message queues and System V semaphores, certain system limits exist on System V shared memory (Section 3.8). Figure 14.5 shows the values for some different implementations. The first column is the traditional System V name for the kernel variable that contains this limit.

Name	Description	DUnix 4.0B	Solaris 2.6
shmmax	max #bytes for a shared memory segment	4,194,304	1,048,576
shmmnb	min #bytes for a shared memory segment	1	1
shmmni	max #shared memory identifiers, systemwide	128	100
shmseg	max #shared memory segments attached per process	32	6

Figure 14.5 Typical system limits for System V shared memory.

Example

The program in Figure 14.6 determines the four limits shown in Figure 14.5.

svshm/limits.c

```
#include    "unpipc.h"

#define MAX_NIDS    4096

int
main(int argc, char **argv)
```

```
{
    int     i, j, shmid[MAX_NIDS];
    void   *addr[MAX_NIDS];
    unsigned long size;

        /* see how many identifiers we can "open" */
    for (i = 0; i <= MAX_NIDS; i++) {
        shmid[i] = shmget(IPC_PRIVATE, 1024, SVSHM_MODE | IPC_CREAT);
        if (shmid[i] == -1) {
            printf("%d identifiers open at once\n", i);
            break;
        }
    }
    for (j = 0; j < i; j++)
        Shmctl(shmid[j], IPC_RMID, NULL);

        /* now see how many we can "attach" */
    for (i = 0; i <= MAX_NIDS; i++) {
        shmid[i] = Shmget(IPC_PRIVATE, 1024, SVSHM_MODE | IPC_CREAT);
        addr[i] = shmat(shmid[i], NULL, 0);
        if (addr[i] == (void *) -1) {
            printf("%d shared memory segments attached at once\n", i);
            Shmctl(shmid[i], IPC_RMID, NULL);   /* the one that failed */
            break;
        }
    }
    for (j = 0; j < i; j++) {
        Shmdt(addr[j]);
        Shmctl(shmid[j], IPC_RMID, NULL);
    }

        /* see how small a shared memory segment we can create */
    for (size = 1;; size++) {
        shmid[0] = shmget(IPC_PRIVATE, size, SVSHM_MODE | IPC_CREAT);
        if (shmid[0] != -1) {   /* stop on first success */
            printf("minimum size of shared memory segment = %lu\n", size);
            Shmctl(shmid[0], IPC_RMID, NULL);
            break;
        }
    }

        /* see how large a shared memory segment we can create */
    for (size = 65536;; size += 4096) {
        shmid[0] = shmget(IPC_PRIVATE, size, SVSHM_MODE | IPC_CREAT);
        if (shmid[0] == -1) {   /* stop on first failure */
            printf("maximum size of shared memory segment = %lu\n", size - 4096);
            break;
        }
        Shmctl(shmid[0], IPC_RMID, NULL);
    }

    exit(0);
}
```

svshm/limits.c

Figure 14.6 Determine the system limits on shared memory.

We run this program under Digital Unix 4.0B.

```
alpha % limits
127 identifiers open at once
32 shared memory segments attached at once
minimum size of shared memory segment = 1
maximum size of shared memory segment = 4194304
```

The reason that Figure 14.5 shows 128 identifiers but our program can create only 127 identifiers is that one shared memory segment has already been created by a system daemon.

14.8 Summary

System V shared memory is similar in concept to Posix shared memory. The most common function calls are

- `shmget` to obtain an identifier,
- `shmat` to attach the shared memory segment to the address space of the process,
- `shmctl` with a command of `IPC_STAT` to fetch the size of an existing shared memory segment, and
- `shmctl` with a command of `IPC_RMID` to remove a shared memory object.

One difference is that the size of a Posix shared memory object can be changed at any time by calling `ftruncate` (as we demonstrated in Exercise 13.1), whereas the size of a System V shared memory object is fixed by `shmget`.

Exercises

14.1 Figure 6.8 was a modification to Figure 6.6 that accepted an identifier instead of a pathname to specify the queue. We showed that the identifier is all we need to know to access a System V message queue (assuming we have adequate permission). Make similar modifications to Figure 14.4 and show that the same feature applies to System V shared memory.