54

POSIX SHARED MEMORY

In previous chapters, we looked at two techniques that allow unrelated processes to share memory regions in order to perform IPC: System V shared memory (Chapter 48) and shared file mappings (Section 49.4.2). Both of these techniques have potential drawbacks:

- The System V shared memory model, which uses keys and identifiers, is not consistent with the standard UNIX I/O model, which uses filenames and descriptors. This difference means that we require an entirely new set of system calls and commands for working with System V shared memory segments.
- Using a shared file mapping for IPC requires the creation of a disk file, even if we are not interested in having a persistent backing store for the shared region. Aside from the inconvenience of needing to create the file, this technique incurs some file I/O overhead.

Because of these drawbacks, POSIX.1b defined a new shared memory API: POSIX shared memory, which is the subject of this chapter.

> POSIX talks about shared memory *objects*, while System V talks about shared memory *segments*. These differences in terminology are historical—both terms are used for referring to regions of memory shared between processes.

54.1 Overview

POSIX shared memory allows to us to share a mapped region between unrelated processes without needing to create a corresponding mapped file. POSIX shared memory is supported on Linux since kernel 2.4.

SUSv3 doesn't specify any of the details of how POSIX shared memory is to be implemented. In particular, there is no requirement for the use of a (real or virtual) file system to identify shared memory objects, although many UNIX implementations do employ a file system for this purpose. Some UNIX implementations create the names for shared memory objects as files in a special location in the standard file system. Linux uses a dedicated *tmpfs* file system (Section 14.10) mounted under the directory `/dev/shm`. This file system has kernel persistence—the shared memory objects that it contains will persist even if no process currently has them open, but they will be lost if the system is shut down.

> The total amount of memory in all POSIX shared memory regions on the system is limited by the size of the underlying *tmpfs* file system. This file system is typically mounted at boot time with some default size (e.g., 256 MB). If necessary, the superuser can change the size of the file system by remounting it using the command *mount −o remount,size=<num-bytes>*.

To use a POSIX shared memory object, we perform two steps:

1. Use the *shm_open()* function to open an object with a specified name. (We described the rules governing the naming of POSIX shared memory objects in Section 51.1.) The *shm_open()* function is analogous to the *open()* system call. It either creates a new shared memory object or opens an existing object. As its function result, *shm_open()* returns a file descriptor referring to the object.
2. Pass the file descriptor obtained in the previous step in a call to *mmap()* that specifies `MAP_SHARED` in the *flags* argument. This maps the shared memory object into the process's virtual address space. As with other uses of *mmap()*, once we have mapped the object, we can close the file descriptor without affecting the mapping. However, we may need to keep the file descriptor open for subsequent use in calls to *fstat()* and *ftruncate()* (see Section 54.2).

> The relationship between *shm_open()* and *mmap()* for POSIX shared memory is analogous to that between *shmget()* and *shmat()* for System V shared memory. The origin of the two-step process (*shm_open()* plus *mmap()*) for using POSIX shared memory objects instead of the use of a single function that performs both tasks is historical. When the POSIX committee added this feature, the *mmap()* call already existed ([Stevens, 1999]). In effect, all that we are doing is replacing calls to *open()* with calls to *shm_open()*, with the difference that using *shm_open()* doesn't require the creation of a file in a disk-based file system.

Since a shared memory object is referred to using a file descriptor, we can usefully employ various file descriptor system calls already defined in the UNIX system (e.g., *ftruncate()*), rather than needing new special-purpose system calls (as is required for System V shared memory).

54.2 Creating Shared Memory Objects

The *shm_open()* function creates and opens a new shared memory object or opens an existing object. The arguments to *shm_open()* are analogous to those for *open()*.

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>        /* Defines mode constants */
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
                                Returns file descriptor on success, or –1 on error
```

The *name* argument identifies the shared memory object to be created or opened. The *oflag* argument is a mask of bits that modify the behavior of the call. The values that can be included in this mask are summarized in Table 54-1.

Table 54-1: Bit values for the *shm_open() oflag* argument

Flag	Description
O_CREAT	Create object if it doesn't already exist
O_EXCL	With O_CREAT, create object exclusively
O_RDONLY	Open for read-only access
O_RDWR	Open for read-write access
O_TRUNC	Truncate object to zero length

One of the purposes of the *oflag* argument is to determine whether we are opening an existing shared memory object or creating and opening a new object. If *oflag* doesn't include O_CREAT, we are opening an existing object. If O_CREAT is specified, then the object is created if it doesn't already exist. Specifying O_EXCL in conjunction with O_CREAT is a request to ensure that the caller is the creator of the object; if the object already exists, an error results (EEXIST).

The *oflag* argument also indicates the kind of access that the calling process will make to the shared memory object, by specifying exactly one of the values O_RDONLY or O_RDWR.

The remaining flag value, O_TRUNC, causes a successful open of an existing shared memory object to truncate the object to a length of zero.

> On Linux, truncation occurs even on a read-only open. However, SUSv3 says that results of using O_TRUNC with a read-only open is undefined, so we can't portably rely on a specific behavior in this case.

When a new shared memory object is created, its ownership and group ownership are taken from the effective user and group IDs of the process calling *shm_open()*, and the object permissions are set according to the value supplied in the *mode* bit-mask argument. The bit values for *mode* are the same as for files (Table 15-4, on page 295). As with the *open()* system call, the permissions mask in *mode* is masked

against the process umask (Section 15.4.6). Unlike *open()*, the *mode* argument is always required for a call to *shm_open()*; if we are not creating a new object, this argument should be specified as 0.

The close-on-exec flag (`FD_CLOEXEC`, Section 27.4) is set on the file descriptor returned by *shm_open()*, so that the file descriptor is automatically closed if the process performs an *exec()*. (This is consistent with the fact that mappings are unmapped when an *exec()* is performed.)

When a new shared memory object is created, it initially has zero length. This means that, after creating a new shared memory object, we normally call *ftruncate()* (Section 5.8) to set the size of the object before calling *mmap()*. Following the *mmap()* call, we may also use *ftruncate()* to expand or shrink the shared memory object as desired, bearing in mind the points discussed in Section 49.4.3.

When a shared memory object is extended, the newly added bytes are automatically initialized to 0.

At any point, we can apply *fstat()* (Section 15.1) to the file descriptor returned by *shm_open()* in order to obtain a *stat* structure whose fields contain information about the shared memory object, including its size (*st_size*), permissions (*st_mode*), owner (*st_uid*), and group (*st_gid*). (These are the only fields that SUSv3 requires *fstat()* to set in the *stat* structure, although Linux also returns meaningful information in the time fields, as well as various other less useful information in the remaining fields.)

The permissions and ownership of a shared memory object can be changed using *fchmod()* and *fchown()*, respectively.

Example program

Listing 54-1 provides a simple example of the use of *shm_open()*, *ftruncate()*, and *mmap()*. This program creates a shared memory object whose size is specified by a command-line argument, and maps the object into the process's virtual address space. (The mapping step is redundant, since we don't actually do anything with the shared memory, but it serves to demonstrate the use of *mmap()*.) The program permits the use of command-line options to select flags (`O_CREAT` and `O_EXCL`) for the *shm_open()* call.

In the following example, we use this program to create a 10,000-byte shared memory object, and then use *ls* to show this object in `/dev/shm`:

```
$ ./pshm_create -c /demo_shm 10000
$ ls -l /dev/shm
total 0
-rw-------    1 mtk      users       10000 Jun 20 11:31 demo_shm
```

Listing 54-1: Creating a POSIX shared memory object

pshm/pshm_create.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"
```

```
static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] name size [octal-perms]\n", progName);
    fprintf(stderr, "    -c   Create shared memory (O_CREAT)\n");
    fprintf(stderr, "    -x   Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt, fd;
    mode_t perms;
    size_t size;
    void *addr;

    flags = O_RDWR;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
        case 'c':   flags |= O_CREAT;           break;
        case 'x':   flags |= O_EXCL;            break;
        default:    usageError(argv[0]);
        }
    }

    if (optind + 1 >= argc)
        usageError(argv[0]);

    size = getLong(argv[optind + 1], GN_ANY_BASE, "size");
    perms = (argc <= optind + 2) ? (S_IRUSR | S_IWUSR) :
                getLong(argv[optind + 2], GN_BASE_8, "octal-perms");

    /* Create shared memory object and set its size */

    fd = shm_open(argv[optind], flags, perms);
    if (fd == -1)
        errExit("shm_open");

    if (ftruncate(fd, size) == -1)
        errExit("ftruncate");

    /* Map shared memory object */

    addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    exit(EXIT_SUCCESS);
}
```

pshm/pshm_create.c

54.3 Using Shared Memory Objects

Listing 54-2 and Listing 54-3 demonstrate the use of a shared memory object to transfer data from one process to another. The program in Listing 54-2 copies the string contained in its second command-line argument into the existing shared memory object named in its first command-line argument. Before mapping the object and performing the copy, the program uses *ftruncate()* to resize the shared memory object to be the same length as the string that is to be copied.

Listing 54-2: Copying data into a POSIX shared memory object

pshm/pshm_write.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd;
    size_t len;                 /* Size of shared memory object */
    char *addr;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name string\n", argv[0]);

    fd = shm_open(argv[1], O_RDWR, 0);      /* Open existing object */
    if (fd == -1)
        errExit("shm_open");

    len = strlen(argv[2]);
    if (ftruncate(fd, len) == -1)           /* Resize object to hold string */
        errExit("ftruncate");
    printf("Resized to %ld bytes\n", (long) len);

    addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1)
        errExit("close");                   /* 'fd' is no longer needed */

    printf("copying %ld bytes\n", (long) len);
    memcpy(addr, argv[2], len);             /* Copy string to shared memory */
    exit(EXIT_SUCCESS);
}
```

pshm/pshm_write.c

The program in Listing 54-3 displays the string in the existing shared memory object named in its command-line argument on standard output. After calling *shm_open()*, the program uses *fstat()* to determine the size of the shared memory and uses that size in the call to *mmap()* that maps the object and in the *write()* call that prints the string.

Listing 54-3: Copying data from a POSIX shared memory object

pshm/pshm_read.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd;
    char *addr;
    struct stat sb;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name\n", argv[0]);

    fd = shm_open(argv[1], O_RDONLY, 0);    /* Open existing object */
    if (fd == -1)
        errExit("shm_open");

    /* Use shared memory object size as length argument for mmap()
       and as number of bytes to write() */

    if (fstat(fd, &sb) == -1)
        errExit("fstat");

    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1);                   /* 'fd' is no longer needed */
        errExit("close");

    write(STDOUT_FILENO, addr, sb.st_size);
    printf("\n");
    exit(EXIT_SUCCESS);
}
```

pshm/pshm_read.c

The following shell session demonstrates the use of the programs in Listing 54-2 and Listing 54-3. We first create a zero-length shared memory object using the program in Listing 54-1.

```
$ ./pshm_create -c /demo_shm 0
$ ls -l /dev/shm                        Check the size of object
total 4
-rw-------    1 mtk    users    0 Jun 21 13:33 demo_shm
```

We then use the program in Listing 54-2 to copy a string into the shared memory object:

```
$ ./pshm_write /demo_shm 'hello'
$ ls -l /dev/shm                        Check that object has changed in size
total 4
-rw-------    1 mtk    users    5 Jun 21 13:33 demo_shm
```

From the output, we can see that the program resized the shared memory object so that it is large enough to hold the specified string.

Finally, we use the program in Listing 54-3 to display the string in the shared memory object:

```
$ ./pshm_read /demo_shm
hello
```

Applications must typically use some synchronization technique to allow processes to coordinate their access to shared memory. In the example shell session shown here, the coordination was provided by the user running the programs one after the other. Typically, applications would instead use a synchronization primitive (e.g., semaphores) to coordinate access to a shared memory object.

54.4 Removing Shared Memory Objects

SUSv3 requires that POSIX shared memory objects have at least kernel persistence; that is, they continue to exist until they are explicitly removed or the system is rebooted. When a shared memory object is no longer required, it should be removed using *shm_unlink()*.

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

int shm_unlink(const char *name);
```

Returns 0 on success, or −1 on error

The *shm_unlink()* function removes the shared memory object specified by *name*. Removing a shared memory object doesn't affect existing mappings of the object (which will remain in effect until the corresponding processes call *munmap()* or terminate), but prevents further *shm_open()* calls from opening the object. Once all processes have unmapped the object, the object is removed, and its contents are lost.

The program in Listing 54-4 uses *shm_unlink()* to remove the shared memory object specified in the program's command-line argument.

Listing 54-4: Using *shm_unlink()* to unlink a POSIX shared memory object

pshm/pshm_unlink.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name\n", argv[0]);
    if (shm_unlink(argv[1]) == -1)
        errExit("shm_unlink");
    exit(EXIT_SUCCESS);
}
```

pshm/pshm_unlink.c

54.5 Comparisons Between Shared Memory APIs

By now, we have considered a number of different techniques for sharing memory regions between unrelated processes:

- System V shared memory (Chapter 48);
- shared file mappings (Section 49.4.2); and
- POSIX shared memory objects (the subject of this chapter).

> Many of the points that we make in this section are also relevant for shared anonymous mappings (Section 49.7), which are used for sharing memory between processes that are related via *fork()*.

A number of points apply to all of these techniques:

- They provide fast IPC, and applications typically must use a semaphore (or other synchronization primitive) to synchronize access to the shared region.
- Once the shared memory region has been mapped into the process's virtual address space, it looks just like any other part of the process's memory space.
- The system places the shared memory regions within the process virtual address space in a similar manner. We outlined this placement while describing System V shared memory in Section 48.5. The Linux-specific `/proc/PID/maps` file lists information about all types of shared memory regions.
- Assuming that we don't attempt to map a shared memory region at a fixed address, we should ensure that all references to locations in the region are calculated as offsets (rather than pointers), since the region may be located at different virtual addresses within different processes (Section 48.6).
- The functions described in Chapter 50 that operate on regions of virtual memory can be applied to shared memory regions created using any of these techniques.

There are also a few notable differences between the techniques for shared memory:

- The fact that the contents of a shared file mapping are synchronized with the underlying mapped file means that the data stored in a shared memory region can persist across system restarts.
- System V and POSIX shared memory use different mechanisms to identify and refer to a shared memory object. System V uses its own scheme of keys and identifiers, which doesn't fit with the standard UNIX I/O model and requires separate system calls (e.g., *shmctl()*) and commands (*ipcs* and *ipcrm*). By contrast, POSIX shared memory employs names and file descriptors, and consequently shared memory objects can be examined and manipulated using a variety of existing UNIX system calls (e.g., *fstat()* and *fchmod()*).
- The size of a System V shared memory segment is fixed at the time of creation (via *shmget()*). By contrast, for a mapping backed by a file or by a POSIX shared memory object, we can use *ftruncate()* to adjust the size of the underlying object, and then re-create the mapping using *munmap()* and *mmap()* (or the Linux-specific *mremap()*).

- Historically, System V shared memory was more widely available than *mmap()* and POSIX shared memory, although most UNIX implementations now provide all of these techniques.

With the exception of the final point regarding portability, the differences listed above are advantages in favor of shared file mappings and POSIX shared memory objects. Thus, in new applications, one of these interfaces may be preferable to System V shared memory. Which one we choose depends on whether or not we require a persistent backing store. Shared file mappings provide such a store; POSIX shared memory objects allow us to avoid the overhead of using a disk file when a backing store is not required.

54.6 Summary

A POSIX shared memory object is used to share a region of memory between unrelated processes without creating an underlying disk file. To do this, we replace the call to *open()* that normally precedes *mmap()* with a call to *shm_open()*. The *shm_open()* call creates a file in a memory-based file system, and we can employ traditional file descriptor system calls to perform various operations on this virtual file. In particular, *ftruncate()* must be used to set the size of the shared memory object, since initially it has a length of zero.

We have now described three techniques for sharing memory regions between unrelated processes: System V shared memory, shared file mappings, and POSIX shared memory objects. There are several similarities between the three techniques. There are also some important differences, and, except for the issue of portability, these differences favor shared file mappings and POSIX shared memory objects.

54.7 Exercise

54-1. Rewrite the programs in Listing 48-2 (`svshm_xfr_writer.c`) and Listing 48-3 (`svshm_xfr_reader.c`) to use POSIX shared memory objects instead of System V shared memory.