NAME

shm_open, shm_unlink − create/open or unlink POSIX shared memory objects

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

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

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

DESCRIPTION

shm_open() creates and opens a new, or opens an existing, POSIX shared memory object. A POSIX shared memory object is in effect a handle which can be used by unrelated processes to **mmap**(2) the same region of shared memory. The**shm_unlink**() function performs the con verse operation, removing an object previously created by **shm_open**().

The operation of **shm_open**() is analogous to that of **open**(2). *name* specifies the shared memory object to be created or opened. For portable use, a shared memory object should be identified by a name of the form */somename*; that is, a null-terminated string of up to **NAME_MAX** (i.e., 255) characters consisting of an initial slash, followed by one or more characters, none of which are slashes.

oflag is a bit mask created by ORing together exactly one of **O_RDONLY** or **O_RDWR** and any of the other flags listed here:

O_RDONLY
: Open the object for read access. A shared memory object opened in this way can be **mmap**(2)ed only for read (**PROT_READ**) access.

O_RDWR
: Open the object for read-write access.

O_CREAT
: Create the shared memory object if it does not exist. The user and group ownership of the object are taken from the corresponding effective IDs of the calling process, and the object's permission bits are set according to the low-order 9 bits of *mode*, except that those bits set in the process file mode creation mask (see **umask**(2)) are cleared for the new object. A set of macro constants which can be used to define *mode* is listed in **open**(2). (Symbolic definitions of these constants can be obtained by including <*sys/stat.h*>.)

  A new shared memory object initially has zero length—the size of the object can be set using **ftruncate**(2). The newly allocated bytes of a shared memory object are automatically initialized to 0.

O_EXCL
: If **O_CREAT** was also specified, and a shared memory object with the given *name* already exists, return an error. The check for the existence of the object, and its creation if it does not exist, are performed atomically.

O_TRUNC
: If the shared memory object already exists, truncate it to zero bytes.

Definitions of these flag values can be obtained by including <*fcntl.h*>.

On successful completion **shm_open**() returns a new file descriptor referring to the shared memory object. This file descriptor is guaranteed to be the lowest-numbered file descriptor not previously opened within the process. The**FD_CLOEXEC** flag (see **fcntl**(2)) is set for the file descriptor .

The file descriptor is normally used in subsequent calls to **ftruncate**(2) (for a newly created object) and **mmap**(2). After a call to**mmap**(2) the file descriptor may be closed without af fecting the memory

mapping.

The operation of **shm_unlink**() is analogous to **unlink**(2): it removes a shared memory object name, and, once all processes have unmapped the object, deallocates and destroys the contents of the associated memory region. After a successful**shm_unlink**(), attempts to **shm_open**() an object with the same *name* f ail (unless **O_CREAT** was specified, in which case a new, distinct object is created).

RETURN VALUE

On success, **shm_open**() returns a file descriptor (a nonnegative integer). On success,**shm_unlink**() returns 0. On failure, both functions return −1 and set *errno* to indicate the error.

ERRORS

EACCES

Permission to **shm_unlink**() the shared memory object was denied.

EACCES

Permission was denied to **shm_open**() *name* in the specified *mode*, or **O_TRUNC** was specified and the caller does not have write permission on the object.

EEXIST

Both **O_CREAT** and **O_EXCL** were specified to **shm_open**() and the shared memory object specified by *name* already exists.

EINVAL

The *name* argument to **shm_open**() was invalid.

EMFILE

The per-process limit on the number of open file descriptors has been reached.

ENAMETOOLONG

The length of *name* exceeds **PATH_MAX**.

ENFILE

The system-wide limit on the total number of open files has been reached.

ENOENT

An attempt was made to **shm_open**() a *name* that did not exist, and **O_CREAT** was not specified.

ENOENT

An attempt was to made to **shm_unlink**() a *name* that does not exist.

VERSIONS

These functions are provided in glibc 2.2 and later.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
shm_open(), **shm_unlink**()	Thread safety	MT-Safe locale

STANDARDS

POSIX.1-2001, POSIX.1-2008.

POSIX.1-2001 says that the group ownership of a newly created shared memory object is set to either the calling process's effective group ID or "a system default group ID". POSIX.1-2008 says that the group ownership may be set to either the calling process's effective group ID or, if the object is visible in the filesystem, the group ID of the parent directory.

NOTES

POSIX leaves the behavior of the combination of **O_RDONLY** and **O_TRUNC** unspecified. On Linux, this will successfully truncate an existing shared memory object—this may not be so on other UNIX systems.

The POSIX shared memory object implementation on Linux makes use of a dedicated **tmpfs**(5) filesystem

that is normally mounted under */dev/shm*.

EXAMPLES

The programs below employ POSIX shared memory and POSIX unnamed semaphores to exchange a piece of data. The "bounce" program (which must be run first) raises the case of a string that is placed into the shared memory by the "send" program. Once the data has been modified, the "send" program then prints the contents of the modified shared memory. An example execution of the two programs is the following:

```
$ ./pshm_ucase_bounce /myshm &
[1] 270171
$ ./pshm_ucase_send /myshm hello
HELLO
```

Further detail about these programs is provided below.

Program source: pshm_ucase.h

The following header file is included by both programs below. Its primary purpose is to define a structure that will be imposed on the memory object that is shared between the two programs.

```
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

#define BUF_SIZE 1024   /* Maximum size for exchanged string */

/* Define a structure that will be imposed on the shared
   memory object */

struct shmbuf {
    sem_t  sem1;            /* POSIX unnamed semaphore */
    sem_t  sem2;            /* POSIX unnamed semaphore */
    size_t cnt;             /* Number of bytes used in 'buf' */
    char   buf[BUF_SIZE];   /* Data being transferred */
};
```

Program source: pshm_ucase_bounce.c

The "bounce" program creates a new shared memory object with the name given in its command-line argument and sizes the object to match the size of the *shmbuf* structure defined in the header file. It then maps the object into the process's address space, and initializes two POSIX semaphores inside the object to 0.

After the "send" program has posted the first of the semaphores, the "bounce" program upper cases the data that has been placed in the memory by the "send" program and then posts the second semaphore to tell the "send" program that it may now access the shared memory.

```
/* pshm_ucase_bounce.c

   Licensed under GNU General Public License v2 or later.
*/
#include <ctype.h>

#include "pshm_ucase.h"
```

```
int
main(int argc, char *argv[])
{
    int            fd;
    char           *shmpath;
    struct shmbuf  *shmp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s /shm-path\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    shmpath = argv[1];

    /* Create shared memory object and set its size to the size
       of our structure. */

    fd = shm_open(shmpath, O_CREAT | O_EXCL | O_RDWR, 0600);
    if (fd == -1)
        errExit("shm_open");

    if (ftruncate(fd, sizeof(struct shmbuf)) == -1)
        errExit("ftruncate");

    /* Map the object into the caller's address space. */

    shmp = mmap(NULL, sizeof(*shmp), PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, 0);
    if (shmp == MAP_FAILED)
        errExit("mmap");

    /* Initialize semaphores as process-shared, with value 0. */

    if (sem_init(&shmp->sem1, 1, 0) == -1)
        errExit("sem_init-sem1");
    if (sem_init(&shmp->sem2, 1, 0) == -1)
        errExit("sem_init-sem2");

    /* Wait for 'sem1' to be posted by peer before touching
       shared memory. */

    if (sem_wait(&shmp->sem1) == -1)
        errExit("sem_wait");

    /* Convert data in shared memory into upper case. */

    for (size_t j = 0; j < shmp->cnt; j++)
        shmp->buf[j] = toupper((unsigned char) shmp->buf[j]);

    /* Post 'sem2' to tell the peer that it can now
       access the modified data in shared memory. */

    if (sem_post(&shmp->sem2) == -1)
        errExit("sem_post");
```

```
        /* Unlink the shared memory object. Even if the peer process
           is still using the object, this is okay. The object will
           be removed only after all open references are closed. */

        shm_unlink(shmpath);

        exit(EXIT_SUCCESS);
    }
```

Program source: pshm_ucase_send.c

The "send" program takes two command-line arguments: the pathname of a shared memory object previously created by the "bounce" program and a string that is to be copied into that object.

The program opens the shared memory object and maps the object into its address space. It then copies the data specified in its second argument into the shared memory, and posts the first semaphore, which tells the "bounce" program that it can now access that data. After the "bounce" program posts the second semaphore, the "send" program prints the contents of the shared memory on standard output.

```
/* pshm_ucase_send.c

   Licensed under GNU General Public License v2 or later.
*/
#include <string.h>

#include "pshm_ucase.h"

int
main(int argc, char *argv[])
{
    int            fd;
    char           *shmpath, *string;
    size_t         len;
    struct shmbuf  *shmp;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s /shm-path string\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    shmpath = argv[1];
    string = argv[2];
    len = strlen(string);

    if (len > BUF_SIZE) {
        fprintf(stderr, "String is too long\n");
        exit(EXIT_FAILURE);
    }

    /* Open the existing shared memory object and map it
       into the caller's address space. */

    fd = shm_open(shmpath, O_RDWR, 0);
    if (fd == -1)
        errExit("shm_open");

    shmp = mmap(NULL, sizeof(*shmp), PROT_READ | PROT_WRITE,
```

```
                    MAP_SHARED, fd, 0);
        if (shmp == MAP_FAILED)
            errExit("mmap");

        /* Copy data into the shared memory object. */

        shmp->cnt = len;
        memcpy(&shmp->buf, string, len);

        /* Tell peer that it can now access shared memory. */

        if (sem_post(&shmp->sem1) == -1)
            errExit("sem_post");

        /* Wait until peer says that it has finished accessing
           the shared memory. */

        if (sem_wait(&shmp->sem2) == -1)
            errExit("sem_wait");

        /* Write modified data in shared memory to standard output. */

        write(STDOUT_FILENO, &shmp->buf, len);
        write(STDOUT_FILENO, "\n", 1);

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

close(2), **fchmod**(2), **fchown**(2), **fcntl**(2), **fstat**(2), **ftruncate**(2), **memfd_create**(2), **mmap**(2), **open**(2), **umask**(2), **shm_overview**(7)