53

POSIX SEMAPHORES

This chapter describes POSIX semaphores, which allow processes and threads to synchronize access to shared resources. In Chapter 47, we described System V semaphores, and we'll assume that the reader is familiar with the general semaphore concepts and rationale for using semaphores that were presented at the start of that chapter. During the course of this chapter, we'll make comparisons between POSIX semaphores and System V semaphores to clarify the ways in which these two semaphore APIs are the same and the ways in which they differ.

53.1 Overview

SUSv3 specifies two types of POSIX semaphores:

- *Named semaphores*: This type of semaphore has a name. By calling *sem_open()* with the same name, unrelated processes can access the same semaphore.
- *Unnamed semaphores*: This type of semaphore doesn't have a name; instead, it resides at an agreed-upon location in memory. Unnamed semaphores can be shared between processes or between a group of threads. When shared between processes, the semaphore must reside in a region of (System V, POSIX, or *mmap()*) shared memory. When shared between threads, the semaphore may reside in an area of memory shared by the threads (e.g., on the heap or in a global variable).

POSIX semaphores operate in a manner similar to System V semaphores; that is, a POSIX semaphore is an integer whose value is not permitted to fall below 0. If a process attempts to decrease the value of a semaphore below 0, then, depending on the function used, the call either blocks or fails with an error indicating that the operation was not currently possible.

Some systems don't provide a full implementation of POSIX semaphores. A typical restriction is that only unnamed thread-shared semaphores are supported. That was the situation on Linux 2.4; with Linux 2.6 and a *glibc* that provides NPTL, a full implementation of POSIX semaphores is available.

> On Linux 2.6 with NPTL, semaphore operations (increment and decrement) are implemented using the *futex(2)* system call.

53.2 Named Semaphores

To work with a named semaphore, we employ the following functions:

- The *sem_open()* function opens or creates a semaphore, initializes the semaphore if it is created by the call, and returns a handle for use in later calls.
- The *sem_post(sem)* and *sem_wait(sem)* functions respectively increment and decrement a semaphore's value.
- The *sem_getvalue()* function retrieves a semaphore's current value.
- The *sem_close()* function removes the calling process's association with a semaphore that it previously opened.
- The *sem_unlink()* function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

SUSv3 doesn't specify how named semaphores are to be implemented. Some UNIX implementations create them as files in a special location in the standard file system. On Linux, they are created as small POSIX shared memory objects with names of the form `sem.`*name*, in a dedicated *tmpfs* file system (Section 14.10) mounted under the directory `/dev/shm`. This file system has kernel persistence—the semaphore 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.

Named semaphores are supported on Linux since kernel 2.6.

53.2.1 Opening a Named Semaphore

The *sem_open()* function creates and opens a new named semaphore or opens an existing semaphore.

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

sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */ );
```

Returns pointer to semaphore on success, or `SEM_FAILED` on error

The *name* argument identifies the semaphore. It is specified according to the rules given in Section 51.1.

The *oflag* argument is a bit mask that determines whether we are opening an existing semaphore or creating and opening a new semaphore. If *oflag* is 0, we are accessing an existing semaphore. If `O_CREAT` is specified in *oflag*, then a new semaphore is created if one with the given *name* doesn't already exist. If *oflag* specifies both `O_CREAT` and `O_EXCL`, and a semaphore with the given *name* already exists, then *sem_open()* fails.

If *sem_open()* is being used to open an existing semaphore, the call requires only two arguments. However, if `O_CREAT` is specified in *flags*, then two further arguments are required: *mode* and *value*. (If the semaphore specified by *name* already exists, then these two arguments are ignored.) These arguments are as follows:

- The *mode* argument is a bit mask that specifies the permissions to be placed on the new semaphore. The bit values are the same as for files (Table 15-4, on page 295), and, as with *open()*, the value in *mode* is masked against the process umask (Section 15.4.6). SUSv3 doesn't specify any access mode flags (`O_RDONLY`, `O_WRONLY`, and `O_RDWR`) for *oflag*. Many implementations, including Linux, assume an access mode of `O_RDWR` when opening a semaphore, since most applications using semaphores must employ both *sem_post()* and *sem_wait()*, which involve reading and modifying a semaphore's value. This means that we should ensure that both read and write permissions are granted to each category of user—owner, group, and other—that needs to access the semaphore.
- The *value* argument is an unsigned integer that specifies the initial value to be assigned to the new semaphore. The creation and initialization of the semaphore are performed atomically. This avoids the complexities required for the initialization of System V semaphores (Section 47.5).

Regardless of whether we are creating a new semaphore or opening an existing semaphore, *sem_open()* returns a pointer to a *sem_t* value, and we employ this pointer in subsequent calls to functions that operate on the semaphore. On error, *sem_open()* returns the value `SEM_FAILED`. (On most implementations, `SEM_FAILED` is defined as either *((sem_t *) 0)* or *((sem_t *) –1)*; Linux defines it as the former.)

SUSv3 states that the results are undefined if we attempt to perform operations (*sem_post()*, *sem_wait()*, and so on) on a *copy* of the *sem_t* variable pointed to by the return value of *sem_open()*. In other words, the following use of *sem2* is not permissible:

```
sem_t *sp, sem2
sp = sem_open(...);
sem2 = *sp;
sem_wait(&sem2);
```

When a child is created via *fork()*, it inherits references to all of the named semaphores that are open in its parent. After the *fork()*, the parent and child can use these semaphores to synchronize their actions.

Example program

The program in Listing 53-1 provides a simple command-line interface to the *sem_open()* function. The command format for this program is shown in the *usageError()* function.

The following shell session log demonstrates the use of this program. We first use the *umask* command to deny all permissions to users in the class other. We then exclusively create a semaphore and examine the contents of the Linux-specific virtual directory that contains named semaphores.

```
$ umask 007
$ ./psem_create -cx /demo 666             666 means read+write for all users
$ ls -l /dev/shm/sem.*
-rw-rw----  1 mtk users 16 Jul  6 12:09 /dev/shm/sem.demo
```

The output of the *ls* command shows that the process umask overrode the specified permissions of read plus write for the user class other.

If we try once more to exclusively create a semaphore with the same name, the operation fails, because the name already exists.

```
$ ./psem_create -cx /demo 666
ERROR [EEXIST File exists] sem_open       Failed because of O_EXCL
```

Listing 53-1: Using *sem_open()* to open or create a POSIX named semaphore

psem/psem_create.c

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

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

int
main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    unsigned int value;
    sem_t *sem;

    flags = 0;
    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 >= argc)
        usageError(argv[0]);

    /* Default permissions are rw-------; default semaphore initialization
       value is 0 */

    perms = (argc <= optind + 1) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind + 1], GN_BASE_8, "octal-perms");
    value = (argc <= optind + 2) ? 0 : getInt(argv[optind + 2], 0, "value");

    sem = sem_open(argv[optind], flags, perms, value);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    exit(EXIT_SUCCESS);
}
```

psem/psem_create.c

53.2.2 Closing a Semaphore

When a process opens a named semaphore, the system records the association between the process and the semaphore. The *sem_close()* function terminates this association (i.e., closes the semaphore), releases any resources that the system has associated with the semaphore for this process, and decreases the count of processes referencing the semaphore.

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Returns 0 on success, or –1 on error

Open named semaphores are also automatically closed on process termination or if the process performs an *exec()*.

Closing a semaphore does not delete it. For that purpose, we need to use *sem_unlink()*.

53.2.3 Removing a Named Semaphore

The *sem_unlink()* function removes the semaphore identified by *name* and marks the semaphore to be destroyed once all processes cease using it (this may mean immediately, if all processes that had the semaphore open have already closed it).

```
#include <semaphore.h>

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

Returns 0 on success, or –1 on error

Listing 53-2 demonstrates the use of *sem_unlink()*.

Listing 53-2: Using *sem_unlink()* to unlink a POSIX named semaphore

psem/psem_unlink.c

```
#include <semaphore.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sem-name\n", argv[0]);

    if (sem_unlink(argv[1]) == -1)
        errExit("sem_unlink");
    exit(EXIT_SUCCESS);
}
```

psem/psem_unlink.c

53.3 Semaphore Operations

As with a System V semaphore, a POSIX semaphore is an integer that the system never allows to go below 0. However, POSIX semaphore operations differ from their System V counterparts in the following respects:

- The functions for changing a semaphore's value—*sem_post()* and *sem_wait()*—operate on just one semaphore at a time. By contrast, the System V *semop()* system call can operate on multiple semaphores in a set.
- The *sem_post()* and *sem_wait()* functions increment and decrement a semaphore's value by exactly one. By contrast, *semop()* can add and subtract arbitrary values.
- There is no equivalent of the wait-for-zero operation provided by System V semaphores (a *semop()* call where the *sops.sem_op* field is specified as 0).

From this list, it may seem that POSIX semaphores are less powerful than System V semaphores. However, this is not the case—anything that we can do with System V semaphores can also be done with POSIX semaphores. In a few cases, a bit more programming effort may be required, but, for typical scenarios, using POSIX semaphores actually requires less programming effort. (The System V semaphore API is rather more complicated than is required for most applications.)

53.3.1 Waiting on a Semaphore

The *sem_wait()* function decrements (decreases by 1) the value of the semaphore referred to by *sem*.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

Returns 0 on success, or –1 on error

If the semaphore currently has a value greater than 0, *sem_wait()* returns immediately. If the value of the semaphore is currently 0, *sem_wait()* blocks until the semaphore value rises above 0; at that time, the semaphore is then decremented and *sem_wait()* returns.

If a blocked *sem_wait()* call is interrupted by a signal handler, then it fails with the error EINTR, regardless of whether the SA_RESTART flag was used when establishing the signal handler with *sigaction()*. (On some other UNIX implementations, SA_RESTART does cause *sem_wait()* to automatically restart.)

The program in Listing 53-3 provides a command-line interface to the *sem_wait()* function. We demonstrate the use of this program shortly.

Listing 53-3: Using *sem_wait()* to decrement a POSIX semaphore

psem/psem_wait.c

```
#include <semaphore.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    sem_t *sem;

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

    sem = sem_open(argv[1], 0);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    if (sem_wait(sem) == -1)
        errExit("sem_wait");

    printf("%ld sem_wait() succeeded\n", (long) getpid());
    exit(EXIT_SUCCESS);
}
```

psem/psem_wait.c

The *sem_trywait()* function is a nonblocking version of *sem_wait()*.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

Returns 0 on success, or –1 on error

If the decrement operation can't be performed immediately, *sem_trywait()* fails with the error EAGAIN.

The *sem_timedwait()* function is another variation on *sem_wait()*. It allows the caller to specify a limit on the time for which the call will block.

```
#define _XOPEN_SOURCE 600
#include <semaphore.h>

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Returns 0 on success, or –1 on error

If a *sem_timedwait()* call times out without being able to decrement the semaphore, then the call fails with the error ETIMEDOUT.

The *abs_timeout* argument is a *timespec* structure (Section 23.4.2) that specifies the timeout as an absolute value in seconds and nanoseconds since the Epoch. If we want to perform a relative timeout, then we must fetch the current value of the CLOCK_REALTIME clock using *clock_gettime()* and add the required amount to that value to produce a *timespec* structure suitable for use with *sem_timedwait()*.

The *sem_timedwait()* function was originally specified in POSIX.1d (1999) and is not available on all UNIX implementations.

53.3.2 Posting a Semaphore

The *sem_post()* function increments (increases by 1) the value of the semaphore referred to by *sem*.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns 0 on success, or –1 on error

If the value of the semaphore was 0 before the *sem_post()* call, and some other process (or thread) is blocked waiting to decrement the semaphore, then that process is awoken, and its *sem_wait()* call proceeds to decrement the semaphore. If multiple processes (or threads) are blocked in *sem_wait()*, then, if the processes are being scheduled under the default round-robin time-sharing policy, it is indeterminate which one will be awoken and allowed to decrement the semaphore. (Like their System V counterparts, POSIX semaphores are only a synchronization mechanism, not a queuing mechanism.)

> SUSv3 specifies that if processes or threads are being executed under a realtime scheduling policy, then the process or thread that will be awoken is the one with the highest priority that has been waiting the longest.

As with System V semaphores, incrementing a POSIX semaphore corresponds to releasing some shared resource for use by another process or thread.

The program in Listing 53-4 provides a command-line interface to the *sem_post()* function. We demonstrate the use of this program shortly.

Listing 53-4: Using *sem_post()* to increment a POSIX semaphore

psem/psem_post.c

```
#include <semaphore.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    sem_t *sem;

    if (argc != 2)
        usageErr("%s sem-name\n", argv[0]);

    sem = sem_open(argv[1], 0);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    if (sem_post(sem) == -1)
        errExit("sem_post");
    exit(EXIT_SUCCESS);
}
```

psem/psem_post.c

53.3.3 Retrieving the Current Value of a Semaphore

The *sem_getvalue()* function returns the current value of the semaphore referred to by *sem* in the *int* pointed to by *sval*.

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

Returns 0 on success, or –1 on error

If one or more processes (or threads) are currently blocked waiting to decrement the semaphore's value, then the value returned in *sval* depends on the implementation. SUSv3 permits two possibilities: 0 or a negative number whose absolute value is the number of waiters blocked in *sem_wait()*. Linux and several other implementations adopt the former behavior; a few other implementations adopt the latter behavior.

> Although returning a negative *sval* if there are blocked waiters can be useful, especially for debugging purposes, SUSv3 doesn't require this behavior because the techniques that some systems use to efficiently implement POSIX semaphores don't (in fact, can't) record counts of blocked waiters.

Note that by the time *sem_getvalue()* returns, the value returned in *sval* may already be out of date. A program that depends on the information returned by *sem_getvalue()* being unchanged by the time of a subsequent operation will be subject to time-of-check, time-of-use race conditions (Section 38.6).

The program in Listing 53-5 uses *sem_getvalue()* to retrieve the value of the semaphore named in its command-line argument, and then displays that value on standard output.

Listing 53-5: Using *sem_getvalue()* to retrieve the value of a POSIX semaphore

psem/psem_getvalue.c

```
#include <semaphore.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int value;
    sem_t *sem;

    if (argc != 2)
        usageErr("%s sem-name\n", argv[0]);

    sem = sem_open(argv[1], 0);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    if (sem_getvalue(sem, &value) == -1)
        errExit("sem_getvalue");

    printf("%d\n", value);
    exit(EXIT_SUCCESS);
}
```

psem/psem_getvalue.c

Example

The following shell session log demonstrates the use of the programs we have shown so far in this chapter. We begin by creating a semaphore whose initial value is zero, and then start a program in the background that attempts to decrement the semaphore:

```
$ ./psem_create -c /demo 600 0
$ ./psem_wait /demo &
[1] 31208
```

The background command blocks, because the semaphore value is currently 0 and therefore can't be decreased.

We then retrieve the semaphore value:

```
$ ./psem_getvalue /demo
0
```

We see the value 0 above. On some other implementations, we might see the value −1, indicating that one process is waiting on the semaphore.

We then execute a command that increments the semaphore. This causes the blocked *sem_wait()* in the background program to complete:

```
$ ./psem_post /demo
$ 31208 sem_wait() succeeded
```

(The last line of output above shows the shell prompt mixed with the output of the background job.)

We press *Enter* to see the next shell prompt, which also causes the shell to report on the terminated background job, and then perform further operations on the semaphore:

```
Press Enter
[1]-  Done          ./psem_wait /demo
$ ./psem_post /demo                        Increment semaphore
$ ./psem_getvalue /demo                    Retrieve semaphore value
1
$ ./psem_unlink /demo                      We're done with this semaphore
```

53.4 Unnamed Semaphores

Unnamed semaphores (also known as *memory-based semaphores*) are variables of type *sem_t* that are stored in memory allocated by the application. The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they share.

Operations on unnamed semaphores use the same functions (*sem_wait()*, *sem_post()*, *sem_getvalue()*, and so on) that are used to operate on named semaphores. In addition, two further functions are required:

- The *sem_init()* function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process.
- The *sem_destroy(sem)* function destroys a semaphore.

These functions should not be used with named semaphores.

Unnamed versus named semaphores

Using an unnamed semaphore allows us to avoid the work of creating a name for a semaphore. This can be useful in the following cases:

- A semaphore that is shared between threads doesn't need a name. Making an unnamed semaphore a shared (global or heap) variable automatically makes it accessible to all threads.
- A semaphore that is being shared between related processes doesn't need a name. If a parent process allocates an unnamed semaphore in a region of shared memory (e.g., a shared anonymous mapping), then a child automatically inherits the mapping and thus the semaphore as part of the operation of *fork()*.

- If we are building a dynamic data structure (e.g., a binary tree), each of whose items requires an associated semaphore, then the simplest approach is to allocate an unnamed semaphore within each item. Opening a named semaphore for each item would require us to design a convention for generating a (unique) semaphore name for each item and to manage those names (e.g., unlinking them when they are no longer required).

53.4.1 Initializing an Unnamed Semaphore

The *sem_init()* function initializes the unnamed semaphore pointed to by *sem* to the value specified by *value*.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Returns 0 on success, or –1 on error

The *pshared* argument indicates whether the semaphore is to be shared between threads or between processes.

- If *pshared* is 0, then the semaphore is to be shared between the threads of the calling process. In this case, *sem* is typically specified as the address of either a global variable or a variable allocated on the heap. A thread-shared semaphore has process persistence; it is destroyed when the process terminates.
- If *pshared* is nonzero, then the semaphore is to be shared between processes. In this case, *sem* must be the address of a location in a region of shared memory (a POSIX shared memory object, a shared mapping created using *mmap()*, or a System V shared memory segment). The semaphore persists as long as the shared memory in which it resides. (The shared memory regions created by most of these techniques have kernel persistence. The exception is shared anonymous mappings, which persist only as long as at least one process maintains the mapping.) Since a child produced via *fork()* inherits its parent's memory mappings, process-shared semaphores are inherited by the child of a *fork()*, and the parent and child can use these semaphores to synchronize their actions.

The *pshared* argument is necessary for the following reasons:

- Some implementations don't support process-shared semaphores. On these systems, specifying a nonzero value for *pshared* causes *sem_init()* to return an error. Linux did not support unnamed process-shared semaphores until kernel 2.6 and the advent of the NPTL threading implementation. (The older LinuxThreads implementation of *sem_init()* fails with the error `ENOSYS` if a nonzero value is specified for *pshared*.)
- On implementations that support both process-shared and thread-shared semaphores, specifying which kind of sharing is required may be necessary because the system must take special actions to support the requested sharing. Providing this information may also permit the system to perform optimizations depending on the type of sharing.

The NPTL *sem_init()* implementation ignores *pshared*, since no special action is required for either type of sharing. Nevertheless, portable and future-proof applications should specify an appropriate value for *pshared*.

> The SUSv3 specification for *sem_init()* defines a failure return of –1, but makes no statement about the return value on success. Nevertheless, the manual pages on most modern UNIX implementations document a 0 return on success. (One notable exception is Solaris, where the description of the return value is similar to the SUSv3 specification. However, inspecting the OpenSolaris source code shows that, on that implementation, *sem_init()* does return 0 on success.) SUSv4 rectifies the situation, specifying that *sem_init()* shall return 0 on success.

There are no permission settings associated with an unnamed semaphore (i.e., *sem_init()* has no analog of the *mode* argument of *sem_open()*). Access to an unnamed semaphore is governed by the permissions that are granted to the process for the underlying shared memory region.

SUSv3 specifies that initializing an already initialized unnamed semaphore results in undefined behavior. In other words, we must design our applications so that just one process or thread calls *sem_init()* to initialize a semaphore.

As with named semaphores, SUSv3 says that the results are undefined if we attempt to perform operations on a *copy* of the *sem_t* variable whose address is passed as the *sem* argument of *sem_init()*. Operations should always be performed only on the "original" semaphore.

Example program

In Section 30.1.2, we presented a program (Listing 30-2) that used mutexes to protect a critical section in which two threads accessed the same global variable. The program in Listing 53-6 solves the same problem using an unnamed thread-shared semaphore.

Listing 53-6: Using a POSIX unnamed semaphore to protect access to a global variable

psem/thread_incr_psem.c

```
#include <semaphore.h>
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static sem_t sem;

static void *                   /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;

    for (j = 0; j < loops; j++) {
        if (sem_wait(&sem) == -1)
            errExit("sem_wait");
        loc = glob;
        loc++;
        glob = loc;

        if (sem_post(&sem) == -1)
            errExit("sem_post");
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    /* Initialize a thread-shared mutex with the value 1 */

    if (sem_init(&sem, 0, 1) == -1)
        errExit("sem_init");

    /* Create two threads that increment 'glob' */

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    /* Wait for threads to terminate */

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```
psem/thread_incr_psem.c

53.4.2 Destroying an Unnamed Semaphore

The *sem_destroy()* function destroys the semaphore *sem*, which must be an unnamed semaphore that was previously initialized using *sem_init()*. It is safe to destroy a semaphore only if no processes or threads are waiting on it.

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
                                        Returns 0 on success, or –1 on error
```

After an unnamed semaphore segment has been destroyed with *sem_destroy()*, it can be reinitialized with *sem_init()*.

An unnamed semaphore should be destroyed before its underlying memory is deallocated. For example, if the semaphore is an automatically allocated variable, it should be destroyed before its host function returns. If the semaphore resides in a POSIX shared memory region, then it should be destroyed after all processes have ceased using the semaphore and before the shared memory object is unlinked with *shm_unlink()*.

On some implementations, omitting calls to *sem_destroy()* doesn't cause problems. However, on other implementations, failing to call *sem_destroy()* can result in resource leaks. Portable applications should call *sem_destroy()* to avoid such problems.

53.5 Comparisons with Other Synchronization Techniques

In this section, we compare POSIX semaphores with two other synchronization techniques: System V semaphores and mutexes.

POSIX semaphores versus System V semaphores

POSIX semaphores and System V semaphores can both be used to synchronize the actions of processes. Section 51.2 listed various advantages of POSIX IPC over System V IPC: the POSIX IPC interface is simpler and more consistent with the traditional UNIX file model, and POSIX IPC objects are reference counted, which simplifies the task of determining when to delete an IPC object. These general advantages also apply to the specific case of POSIX (named) semaphores versus System V semaphores.

POSIX semaphores have the following further advantages over System V semaphores:

- The POSIX semaphore interface is much simpler than the System V semaphore interface. This simplicity is achieved without loss of functional power.
- POSIX named semaphores eliminate the initialization problem associated with System V semaphores (Section 47.5).
- It is easier to associate a POSIX unnamed semaphore with a dynamically allocated memory object: the semaphore can simply be embedded inside the object.
- In scenarios where there is a high degree of contention for a semaphore (i.e., operations on the semaphore are frequently blocked because another process has set the semaphore to a value that prevents the operation proceeding immediately), then the performance of POSIX semaphores and System V semaphores is similar. However, in cases where there is low contention for a semaphore (i.e., the semaphore's value is such that operations can normally

proceed without blocking), then POSIX semaphores perform considerably better than System V semaphores. (On the systems tested by the author, the difference in performance is more than an order of magnitude; see Exercise 53-4.) POSIX semaphores perform so much better in this case because the way in which they are implemented only requires a system call when contention occurs, whereas System V semaphore operations always require a system call, regardless of contention.

However, POSIX semaphores also have the following disadvantages compared to System V semaphores:

- POSIX semaphores are somewhat less portable. (On Linux, named semaphores have been supported only since kernel 2.6.)
- POSIX semaphores don't provide an equivalent of the System V semaphore undo feature. (However, as we noted in Section 47.8, this feature may not be useful in some circumstances.)

POSIX semaphores versus Pthreads mutexes

POSIX semaphores and Pthreads mutexes can both be used to synchronize the actions of threads within the same process, and their performance is similar. However, mutexes are usually preferable, because the ownership property of mutexes enforces good structuring of code (only the thread that locks a mutex can unlock it). By contrast, one thread can increment a semaphore that was decremented by another thread. This flexibility can lead to poorly structured synchronization designs. (For this reason, semaphores are sometimes referred to as the "gotos" of concurrent programming.)

There is one circumstance in which mutexes can't be used in a multithreaded application and semaphores may therefore be preferable. Because it is async-signal-safe (see Table 21-1, on page 426), the *sem_post()* function can be used from within a signal handler to synchronize with another thread. This is not possible with mutexes, because the Pthreads functions for operating on mutexes are not async-signal-safe. However, because it is usually preferable to deal with asynchronous signals by accepting them using *sigwaitinfo()* (or similar), rather than using signal handlers (see Section 33.2.4), this advantage of semaphores over mutexes is seldom required.

53.6 Semaphore Limits

SUSv3 defines two limits applying to semaphores:

`SEM_NSEMS_MAX`

This is the maximum number of POSIX semaphores that a process may have. SUSv3 requires that this limit be at least 256. On Linux, the number of POSIX semaphores is effectively limited only by available memory.

`SEM_VALUE_MAX`

This is the maximum value that a POSIX semaphore may reach. Semaphores may assume any value from 0 up to this limit. SUSv3 requires this limit to be at least 32,767; the Linux implementation allows values up to `INT_MAX` (2,147,483,647 on Linux/x86-32).

53.7 Summary

POSIX semaphores allow processes or threads to synchronize their actions. POSIX semaphores come in two types: named and unnamed. A named semaphore is identified by a name, and can be shared by any processes that have permission to open the semaphore. An unnamed semaphore has no name, but processes or threads can share the same semaphore by placing it in a region of memory that they share (e.g., in a POSIX shared memory object for process sharing, or in a global variable for thread sharing).

The POSIX semaphore interface is simpler than the System V semaphore interface. Semaphores are allocated and operated on individually, and the wait and post operations adjust a semaphore's value by one.

POSIX semaphores have a number of advantages over System V semaphores, but they are somewhat less portable. For synchronization within multithreaded applications, mutexes are generally preferred over semaphores.

Further information

[Stevens, 1999] provides an alternative presentation of POSIX semaphores and shows user-space implementations using various other IPC mechanisms (FIFOs, memory-mapped files, and System V semaphores). [Butenhof, 1996] describes the use of POSIX semaphores in multithreaded applications.

53.8 Exercises

53-1. Rewrite the programs in Listing 48-2 and Listing 48-3 (Section 48.4) as a threaded application, with the two threads passing data to each other via a global buffer, and using POSIX semaphores for synchronization.

53-2. Modify the program in Listing 53-3 (`psem_wait.c`) to use *sem_timedwait()* instead of *sem_wait()*. The program should take an additional command-line argument that specifies a (relative) number of seconds to be used as the timeout for the *sem_timedwait()* call.

53-3. Devise an implementation of POSIX semaphores using System V semaphores.

53-4. In Section 53.5, we noted that POSIX semaphores perform much better than System V semaphores in the case where the semaphore is uncontended. Write two programs (one for each semaphore type) to verify this. Each program should simply increment and decrement a semaphore a specified number of times. Compare the times required for the two programs.