47

SYSTEM V SEMAPHORES

This chapter describes System V semaphores. Unlike the IPC mechanisms described in previous chapters, System V semaphores are not used to transfer data between processes. Instead, they allow processes to synchronize their actions. One common use of a semaphore is to synchronize access to a block of shared memory, in order to prevent one process from accessing the shared memory at the same time as another process is updating it.

A semaphore is a kernel-maintained integer whose value is restricted to being greater than or equal to 0. Various operations (i.e., system calls) can be performed on a semaphore, including the following:

- setting the semaphore to an absolute value;
- adding a number to the current value of the semaphore;
- subtracting a number from the current value of the semaphore; and
- waiting for the semaphore value to be equal to 0.

The last two of these operations may cause the calling process to block. When lowering a semaphore value, the kernel blocks any attempt to decrease the value below 0. Similarly, waiting for a semaphore to equal 0 blocks the calling process if the semaphore value is not currently 0. In both cases, the calling process remains blocked until some other process alters the semaphore to a value that allows the operation to proceed, at which point the kernel wakes the blocked process. Figure 47-1 shows

the use of a semaphore to synchronize the actions of two processes that alternately move the semaphore value between 0 and 1.

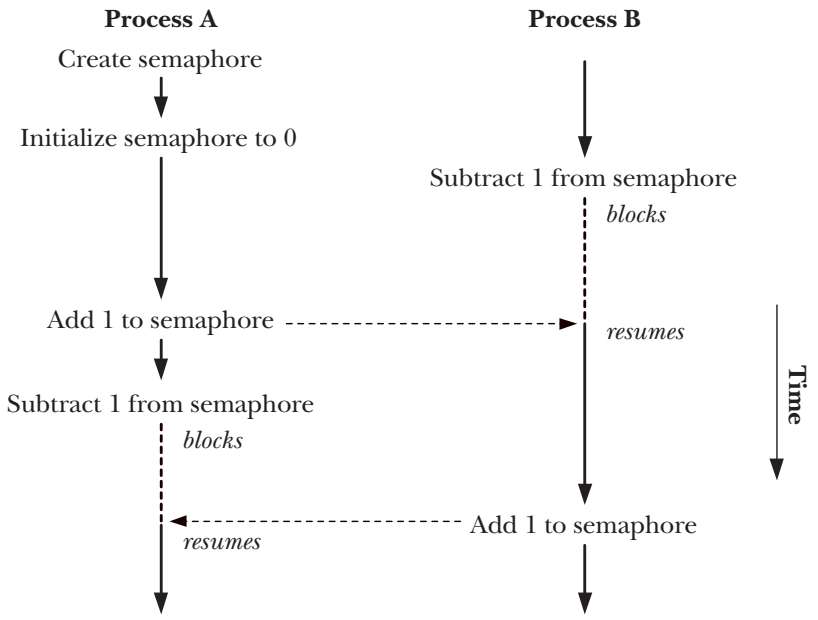

Figure 47-1: Using a semaphore to synchronize two processes

In terms of controlling the actions of a process, a semaphore has no meaning in and of itself. Its meaning is determined only by the associations given to it by the processes using the semaphore. Typically, processes agree on a convention that associates a semaphore with a shared resource, such as a region of shared memory. Other uses of semaphores are also possible, such as synchronization between parent and child processes after *fork()*. (In Section 24.5, we looked at the use of signals to accomplish the same task.)

47.1 Overview

The general steps for using a System V semaphore are the following:

- Create or open a semaphore set using *semget()*.
- Initialize the semaphores in the set using the *semctl()* `SETVAL` or `SETALL` operation. (Only one process should do this.)
- Perform operations on semaphore values using *semop()*. The processes using the semaphore typically use these operations to indicate acquisition and release of a shared resource.
- When all processes have finished using the semaphore set, remove the set using the *semctl()* `IPC_RMID` operation. (Only one process should do this.)

Most operating systems provide some type of semaphore primitive for use in application programs. However, System V semaphores are rendered unusually complex by the fact that they are allocated in groups called *semaphore sets*. The number of

semaphores in a set is specified when the set is created using the *semget()* system call. While it is common to operate on a single semaphore at a time, the *semop()* system call allows us to atomically perform a group of operations on multiple semaphores in the same set.

Because System V semaphores are created and initialized in separate steps, race conditions can result if two processes try to perform these steps at the same time. Describing this race condition and how to avoid it requires that we describe *semctl()* before describing *semop()*, which means that there is quite a lot of material to cover before we have all of the details required to fully understand semaphores.

In the meantime, we provide Listing 47-1 as a simple example of the use of the various semaphore system calls. This program operates in two modes:

- Given a single integer command-line argument, the program creates a new semaphore set containing a single semaphore, and initializes the semaphore to the value supplied in the command-line argument. The program displays the identifier of the new semaphore set.
- Given two command-line arguments, the program interprets them as (in order) the identifier of an existing semaphore set and a value to be added to the first semaphore (numbered 0) in that set. The program carries out the specified operation on that semaphore. To enable us to monitor the semaphore operation, the program prints messages before and after the operation. Each of these messages begins with the process ID, so that we can distinguish the output of multiple instances of the program.

The following shell session log demonstrates the use of the program in Listing 47-1. We begin by creating a semaphore that is initialized to 0:

```
$ ./svsem_demo 0
Semaphore ID = 98307                    ID of new semaphore set
```

We then execute a background command that tries to decrease the semaphore value by 2:

```
$ ./svsem_demo 98307 -2 &
23338: about to semop at  10:19:42
[1] 23338
```

This command blocked, because the value of the semaphore can't be decreased below 0. Now, we execute a command that adds 3 to the semaphore value:

```
$ ./svsem_demo 98307 +3
23339: about to semop at  10:19:55
23339: semop completed at 10:19:55
23338: semop completed at 10:19:55
[1]+  Done              ./svsem_demo 98307 -2
```

The semaphore increment operation succeeded immediately, and caused the semaphore decrement operation in the background command to proceed, since that operation could now be performed without leaving the semaphore's value below 0.

Listing 47-1: Creating and operating on System V semaphores

svsem/svsem_demo.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include "curr_time.h"                  /* Declaration of currTime() */
#include "semun.h"                      /* Definition of semun union */
#include "tlpi_hdr.h"

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

    if (argc < 2 || argc > 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s init-value\n"
                 "   or: %s semid operation\n", argv[0], argv[0]);

    if (argc == 2) {            /* Create and initialize semaphore */
        union semun arg;

        semid = semget(IPC_PRIVATE, 1, S_IRUSR | S_IWUSR);
        if (semid == -1)
            errExit("semid");

        arg.val = getInt(argv[1], 0, "init-value");
        if (semctl(semid, /* semnum= */ 0, SETVAL, arg) == -1)
            errExit("semctl");

        printf("Semaphore ID = %d\n", semid);

    } else {                    /* Perform an operation on first semaphore */

        struct sembuf sop;              /* Structure defining operation */

        semid = getInt(argv[1], 0, "semid");

        sop.sem_num = 0;                /* Specifies first semaphore in set */
        sop.sem_op = getInt(argv[2], 0, "operation");
                                        /* Add, subtract, or wait for 0 */
        sop.sem_flg = 0;                /* No special options for operation */

        printf("%ld: about to semop at  %s\n", (long) getpid(), currTime("%T"));
        if (semop(semid, &sop, 1) == -1)
            errExit("semop");

        printf("%ld: semop completed at %s\n", (long) getpid(), currTime("%T"));
    }

    exit(EXIT_SUCCESS);
}
```

svsem/svsem_demo.c

47.2 Creating or Opening a Semaphore Set

The *semget()* system call creates a new semaphore set or obtains the identifier of an existing set.

```
#include <sys/types.h>        /* For portability */
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Returns semaphore set identifier on success, or –1 on error

The *key* argument is a key generated using one of the methods described in Section 45.2 (i.e., usually the value `IPC_PRIVATE` or a key returned by *ftok()*).

If we are using *semget()* to create a new semaphore set, then *nsems* specifies the number of semaphores in that set, and must be greater than 0. If we are using *semget()* to obtain the identifier of an existing set, then *nsems* must be less than or equal to the size of the set (or the error `EINVAL` results). It is not possible to change the number of semaphores in an existing set.

The *semflg* argument is a bit mask specifying the permissions to be placed on a new semaphore set or checked against an existing set. These permissions are specified in the same manner as for files (Table 15-4, on page 295). In addition, zero or more of the following flags can be ORed (|) in *semflg* to control the operation of *semget()*:

`IPC_CREAT`
: If no semaphore set with the specified *key* exists, create a new set.

`IPC_EXCL`
: If `IPC_CREAT` was also specified, and a semaphore set with the specified *key* already exists, fail with the error `EEXIST`.

These flags are described in more detail in Section 45.1.

On success, *semget()* returns the identifier for the new or existing semaphore set. Subsequent system calls referring to individual semaphores must specify both the semaphore set identifier and the number of the semaphore within that set. The semaphores within a set are numbered starting at 0.

47.3 Semaphore Control Operations

The *semctl()* system call performs a variety of control operations on a semaphore set or on an individual semaphore within a set.

```
#include <sys/types.h>        /* For portability */
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ... /* union semun arg */);
```

Returns nonnegative integer on success (see text); returns –1 on error

The *semid* argument is the identifier of the semaphore set on which the operation is to be performed. For those operations performed on a single semaphore, the *semnum* argument identifies a particular semaphore within the set. For other operations, this argument is ignored, and we can specify it as 0. The *cmd* argument specifies the operation to be performed.

Certain operations require a fourth argument to *semctl()*, which we refer to by the name *arg* in the remainder of this section. This argument is a union defined as shown in Listing 47-2. We must explicitly define this union in our programs. We do this in our example programs by including the header file in Listing 47-2.

> Although placing the definition of the *semun* union in a standard header file would be sensible, SUSv3 requires the programmer to explicitly define it instead. Nevertheless, some UNIX implementations do provide this definition in <sys/sem.h>. Older versions of *glibc* (up to and including version 2.0) also provided this definition. In conformance with SUSv3, more recent versions of *glibc* do not, and the macro _SEM_SEMUN_UNDEFINED is defined with the value 1 in <sys/sem.h> to indicate this fact (i.e., an application compiled against *glibc* can test this macro to determine if the program must itself define the *semun* union).

Listing 47-2: Definition of the *semun* union

svsem/semun.h

```
#ifndef SEMUN_H
#define SEMUN_H                 /* Prevent accidental double inclusion */

#include <sys/types.h>          /* For portability */
#include <sys/sem.h>

union semun {                   /* Used in calls to semctl() */
    int                 val;
    struct semid_ds *   buf;
    unsigned short *    array;
#if defined(__linux__)
    struct seminfo *    __buf;
#endif
};

#endif
```

svsem/semun.h

SUSv2 and SUSv3 specify that the final argument to *semctl()* is optional. However, a few (mainly older) UNIX implementations (and older versions of *glibc*) prototyped *semctl()* as follows:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

This meant that the fourth argument was required even in the cases where it is not actually used (e.g., the IPC_RMID and GETVAL operations described below). For full portability, we specify a dummy final argument to *semctl()* in those calls where it is not required.

In the remainder of this section, we consider the various control operations that can be specified for *cmd*.

Generic control operations

The following operations are the same ones that can be applied to other types of System V IPC objects. In each case, the *semnum* argument is ignored. Further details about these operations, including the privileges and permissions required by the calling process, are provided in Section 45.3.

`IPC_RMID`
: Immediately remove the semaphore set and its associated *semid_ds* data structure. Any processes blocked in *semop()* calls waiting on semaphores in this set are immediately awakened, with *semop()* reporting the error `EIDRM`. The *arg* argument is not required.

`IPC_STAT`
: Place a copy of the *semid_ds* data structure associated with this semaphore set in the buffer pointed to by *arg.buf*. We describe the *semid_ds* structure in Section 47.4.

`IPC_SET`
: Update selected fields of the *semid_ds* data structure associated with this semaphore set using values in the buffer pointed to by *arg.buf*.

Retrieving and initializing semaphore values

The following operations retrieve or initialize the value(s) of an individual semaphore or of all semaphores in a set. Retrieving a semaphore value requires read permission on the semaphore, while initializing the value requires alter (write) permission.

`GETVAL`
: As its function result, *semctl()* returns the value of the *semnum*-th semaphore in the semaphore set specified by *semid*. The *arg* argument is not required.

`SETVAL`
: The value of the *semnum*-th semaphore in the set referred to by *semid* is initialized to the value specified in *arg.val*.

`GETALL`
: Retrieve the values of all of the semaphores in the set referred to by *semid*, placing them in the array pointed to by *arg.array*. The programmer must ensure that this array is of sufficient size. (The number of semaphores in a set can be obtained from the *sem_nsems* field of the *semid_ds* data structure retrieved by an `IPC_STAT` operation.) The *semnum* argument is ignored. An example of the use of the `GETALL` operation is provided in Listing 47-3.

`SETALL`
: Initialize all semaphores in the set referred to by *semid*, using the values supplied in the array pointed to by *arg.array*. The *semnum* argument is ignored. Listing 47-4 demonstrates the use of the `SETALL` operation.

If another process is waiting to perform an operation on the semaphore(s) modified by the `SETVAL` or `SETALL` operations, and the change(s) made would permit that operation to proceed, then the kernel wakes up that process.

Changing the value of a semaphore with SETVAL or SETALL clears the undo entries for that semaphore in all processes. We describe semaphore undo entries in Section 47.8.

Note that the information returned by GETVAL and GETALL may already be out of date by the time the calling process comes to use it. Any program that depends on the information returned by these operations being unchanged may be subject to time-of-check, time-of-use race conditions (Section 38.6).

Retrieving per-semaphore information

The following operations return (via the function result value) information about the *semnum*-th semaphore of the set referred to by *semid*. For all of these operations, read permission is required on the semaphore set, and the *arg* argument is not required.

GETPID
: Return the process ID of the last process to perform a *semop()* on this semaphore; this is referred to as the *sempid* value. If no process has yet performed a *semop()* on this semaphore, 0 is returned.

GETNCNT
: Return the number of processes currently waiting for the value of this semaphore to increase; this is referred to as the *semncnt* value.

GETZCNT
: Return the number of processes currently waiting for the value of this semaphore to become 0; this is referred to as the *semzcnt* value.

As with the GETVAL and GETALL operations described above, the information returned by the GETPID, GETNCNT, and GETZCNT operations may already be out of date by the time the calling process comes to use it.

Listing 47-3 demonstrates the use of these three operations.

47.4 Semaphore Associated Data Structure

Each semaphore set has an associated *semid_ds* data structure of the following form:

```
struct semid_ds {
    struct ipc_perm sem_perm;       /* Ownership and permissions */
    time_t          sem_otime;      /* Time of last semop() */
    time_t          sem_ctime;      /* Time of last change */
    unsigned long   sem_nsems;      /* Number of semaphores in set */
};
```

SUSv3 requires all of the fields that we show in the *semid_ds* structure. Some other UNIX implementations include additional nonstandard fields. On Linux 2.4 and later, the *sem_nsems* field is typed as *unsigned long*. SUSv3 specifies the type of this field as *unsigned short*, and it is so defined in Linux 2.2 and on most other UNIX implementations.

The fields of the *semid_ds* structure are implicitly updated by various semaphore system calls, and certain subfields of the *sem_perm* field can be explicitly updated using the *semctl()* `IPC_SET` operation. The details are as follows:

sem_perm
: When the semaphore set is created, the fields of this substructure are initialized as described in Section 45.3. The *uid*, *gid*, and *mode* subfields can be updated via `IPC_SET`.

sem_otime
: This field is set to 0 when the semaphore set is created, and then set to the current time on each successful *semop()*, or when the semaphore value is modified as a consequence of a `SEM_UNDO` operation (Section 47.8). This field and *sem_ctime* are typed as *time_t*, and store time in seconds since the Epoch.

sem_ctime
: This field is set to the current time when the semaphore set is created and on each successful `IPC_SET`, `SETALL`, or `SETVAL` operation. (On some UNIX implementations, the `SETALL` and `SETVAL` operations don't modify *sem_ctime*.)

sem_nsems
: When the set is created, this field is initialized to the number of semaphores in the set.

In the remainder of this section, we show two example programs that make use of the *semid_ds* data structure and some of the *semctl()* operations described in Section 47.3. We demonstrate the use of both of these programs in Section 47.6.

Monitoring a semaphore set

The program in Listing 47-3 makes use of various *semctl()* operations to display information about the existing semaphore set whose identifier is provided as its command-line argument. The program first displays the time fields from the *semid_ds* data structure. Then, for each semaphore in the set, the program displays the semaphore's current value, as well as its *sempid*, *semncnt*, and *semzcnt* values.

Listing 47-3: A semaphore monitoring program

svsem/svsem_mon.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include <time.h>
#include "semun.h"                      /* Definition of semun union */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct semid_ds ds;
    union semun arg, dummy;             /* Fourth argument for semctl() */
    int semid, j;
```

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

    semid = getInt(argv[1], 0, "semid");

    arg.buf = &ds;
    if (semctl(semid, 0, IPC_STAT, arg) == -1)
        errExit("semctl");

    printf("Semaphore changed: %s", ctime(&ds.sem_ctime));
    printf("Last semop():      %s", ctime(&ds.sem_otime));

    /* Display per-semaphore information */

    arg.array = calloc(ds.sem_nsems, sizeof(arg.array[0]));
    if (arg.array == NULL)
        errExit("calloc");
    if (semctl(semid, 0, GETALL, arg) == -1)
        errExit("semctl-GETALL");

    printf("Sem #  Value  SEMPID  SEMNCNT  SEMZCNT\n");

    for (j = 0; j < ds.sem_nsems; j++)
        printf("%3d   %5d   %5d  %5d    %5d\n", j, arg.array[j],
                semctl(semid, j, GETPID, dummy),
                semctl(semid, j, GETNCNT, dummy),
                semctl(semid, j, GETZCNT, dummy));

    exit(EXIT_SUCCESS);
}
```

svsem/svsem_mon.c

Initializing all semaphores in a set

The program in Listing 47-4 provides a command-line interface for initializing all of the semaphores in an existing set. The first command-line argument is the identifier of the semaphore set to be initialized. The remaining command-line arguments specify the values to which the semaphores are to be initialized (there must be as many of these arguments as there are semaphores in the set).

Listing 47-4: Using the SETALL operation to initialize a System V semaphore set

svsem/svsem_setall.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include "semun.h"                      /* Definition of semun union */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct semid_ds ds;
    union semun arg;                    /* Fourth argument for semctl() */
    int j, semid;
```

```
    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s semid val...\n", argv[0]);

    semid = getInt(argv[1], 0, "semid");

    /* Obtain size of semaphore set */

    arg.buf = &ds;
    if (semctl(semid, 0, IPC_STAT, arg) == -1)
        errExit("semctl");

    if (ds.sem_nsems != argc - 2)
        cmdLineErr("Set contains %ld semaphores, but %d values were supplied\n",
                (long) ds.sem_nsems, argc - 2);

    /* Set up array of values; perform semaphore initialization */

    arg.array = calloc(ds.sem_nsems, sizeof(arg.array[0]));
    if (arg.array == NULL)
        errExit("calloc");

    for (j = 2; j < argc; j++)
        arg.array[j - 2] = getInt(argv[j], 0, "val");

    if (semctl(semid, 0, SETALL, arg) == -1)
        errExit("semctl-SETALL");
    printf("Semaphore values changed (PID=%ld)\n", (long) getpid());

    exit(EXIT_SUCCESS);
}
```

svsem/svsem_setall.c

47.5 Semaphore Initialization

According to SUSv3, an implementation is not required to initialize the values of the semaphores in a set created by *semget()*. Instead, the programmer must explicitly initialize the semaphores using the *semctl()* system call. (On Linux, the semaphores returned by *semget()* are in fact initialized to 0, but we can't portably rely on this.) As stated earlier, the fact that semaphore creation and initialization must be performed by separate system calls, instead of in a single atomic step, leads to possible race conditions when initializing a semaphore. In this section, we detail the nature of the race and look at a method of avoiding it based on an idea described in [Stevens, 1999].

Suppose that we have an application consisting of multiple peer processes employing a semaphore to coordinate their actions. Since no single process is guaranteed to be the first to use the semaphore (this is what is meant by the term *peer*), each process must be prepared to create and initialize the semaphore if it doesn't already exist. For this purpose, we might consider employing the code shown in Listing 47-5.

Listing 47-5: Incorrectly initializing a System V semaphore

from **svsem/svsem_bad_init.c**

```
/* Create a set containing 1 semaphore */

semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);

if (semid != -1) {                  /* Successfully created the semaphore */
    union semun arg;

    /* XXXX */

    arg.val = 0;                    /* Initialize semaphore */
    if (semctl(semid, 0, SETVAL, arg) == -1)
        errExit("semctl");

} else {                            /* We didn't create the semaphore */
    if (errno != EEXIST) {          /* Unexpected error from semget() */
        errExit("semget");

    semid = semget(key, 1, perms);  /* Retrieve ID of existing set */
    if (semid == -1)
       errExit("semget");
}

/* Now perform some operation on the semaphore */

sops[0].sem_op = 1;                 /* Add 1... */
sops[0].sem_num = 0;                /* to semaphore 0 */
sops[0].sem_flg = 0;
if (semop(semid, sops, 1) == -1)
    errExit("semop");
```

from **svsem/svsem_bad_init.c**

The problem with the code in Listing 47-5 is that if two processes execute it at the same time, then the sequence shown in Figure 47-2 could occur, if the first process's time slice happens to expire at the point marked `XXXX` in the code. This sequence is problematic for two reasons. First, process B performs a *semop()* on an uninitialized semaphore (i.e., one whose value is arbitrary). Second, the *semctl()* call in process A overwrites the changes made by process B.

The solution to this problem relies on a historical, and now standardized, feature of the initialization of the *sem_otime* field in the *semid_ds* data structure associated with the semaphore set. When a semaphore set is first created, the *sem_otime* field is initialized to 0, and it is changed only by a subsequent *semop()* call. We can exploit this feature to eliminate the race condition described above. We do this by inserting extra code to force the second process (i.e., the one that does not create the semaphore) to wait until the first process has both initialized the semaphore *and* executed a *semop()* call that updates the *sem_otime* field, but does not modify the semaphore's value. The modified code is shown in Listing 47-6.

> Unfortunately, the solution to the initialization problem described in the main text doesn't work on all UNIX implementations. In some versions of the modern BSD derivatives, *semop()* doesn't update the *sem_otime* field.

Listing 47-6: Initializing a System V semaphore

from **svsem/svsem_good_init.c**

```
    semid = semget(key, 1, IPC_CREAT | IPC_EXCL | perms);

    if (semid != -1) {                  /* Successfully created the semaphore */
        union semun arg;
        struct sembuf sop;

        arg.val = 0;                    /* So initialize it to 0 */
        if (semctl(semid, 0, SETVAL, arg) == -1)
            errExit("semctl");

        /* Perform a "no-op" semaphore operation - changes sem_otime
           so other processes can see we've initialized the set. */

        sop.sem_num = 0;                /* Operate on semaphore 0 */
        sop.sem_op = 0;                 /* Wait for value to equal 0 */
        sop.sem_flg = 0;
        if (semop(semid, &sop, 1) == -1)
            errExit("semop");

    } else {                            /* We didn't create the semaphore set */
        const int MAX_TRIES = 10;
        int j;
        union semun arg;
        struct semid_ds ds;

        if (errno != EEXIST) {          /* Unexpected error from semget() */
            errExit("semget");

        semid = semget(key, 1, perms);  /* Retrieve ID of existing set */
        if (semid == -1)
            errExit("semget");

        /* Wait until another process has called semop() */

        arg.buf = &ds;
        for (j = 0; j < MAX_TRIES; j++) {
            if (semctl(semid, 0, IPC_STAT, arg) == -1)
                errExit("semctl");
            if (ds.sem_otime != 0)      /* semop() performed? */
                break;                  /* Yes, quit loop */
            sleep(1);                   /* If not, wait and retry */
        }

        if (ds.sem_otime == 0)          /* Loop ran to completion! */
            fatal("Existing semaphore not initialized");
    }

    /* Now perform some operation on the semaphore */
```

from **svsem/svsem_good_init.c**

We can use variations of the technique shown in Listing 47-6 to ensure that multiple semaphores in a set are correctly initialized, or that a semaphore is initialized to a nonzero value.

This rather complex solution to the race problem is not required in all applications. We don't need it if one process is guaranteed to be able to create and initialize the semaphore before any other processes attempt to use it. This would be the case, for example, if a parent creates and initializes the semaphore before creating child processes with which it shares the semaphore. In such cases, it is sufficient for the first process to follow its *semget()* call by a *semctl()* `SETVAL` or `SETALL` operation.

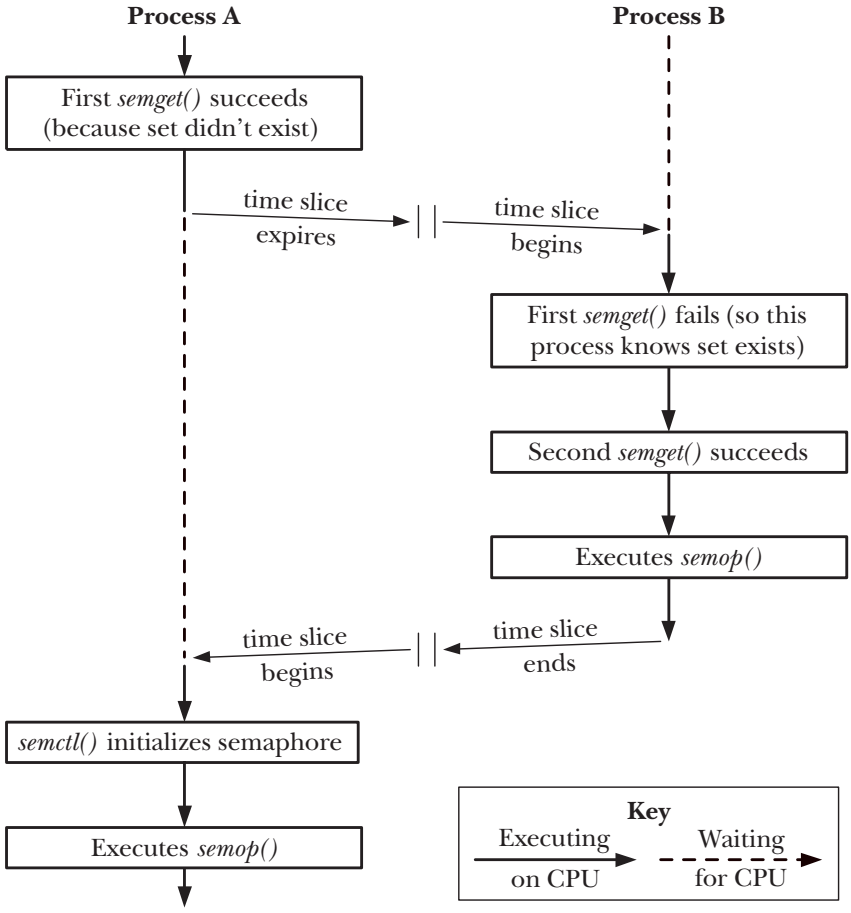

Figure 47-2: Two processes racing to initialize the same semaphore

47.6 Semaphore Operations

The *semop()* system call performs one or more operations on the semaphores in the semaphore set identified by *semid*.

```
#include <sys/types.h>        /* For portability */
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

Returns 0 on success, or –1 on error

The *sops* argument is a pointer to an array that contains the operations to be performed, and *nsops* gives the size of this array (which must contain at least one element). The operations are performed atomically and in array order. The elements of the *sops* array are structures of the following form:

```
struct sembuf {
    unsigned short sem_num;     /* Semaphore number */
    short          sem_op;      /* Operation to be performed */
    short          sem_flg;     /* Operation flags (IPC_NOWAIT and SEM_UNDO) */
};
```

The *sem_num* field identifies the semaphore within the set upon which the operation is to be performed. The *sem_op* field specifies the operation to be performed:

- If *sem_op* is greater than 0, the value of *sem_op* is added to the semaphore value. As a result, other processes waiting to decrease the semaphore value may be awakened and perform their operations. The calling process must have alter (write) permission on the semaphore.
- If *sem_op* equals 0, the value of the semaphore is checked to see whether it currently equals 0. If it does, the operation completes immediately; otherwise, *semop()* blocks until the semaphore value becomes 0. The calling process must have read permission on the semaphore.
- If *sem_op* is less than 0, decrease the value of the semaphore by the amount specified in *sem_op*. If the current value of the semaphore is greater than or equal to the absolute value of *sem_op*, the operation completes immediately. Otherwise, *semop()* blocks until the semaphore value has been increased to a level that permits the operation to be performed without resulting in a negative value. The calling process must have alter permission on the semaphore.

Semantically, increasing the value of a semaphore corresponds to making a resource available so that others can use it, while decreasing the value of a semaphore corresponds to reserving a resource for (exclusive) use by this process. When decreasing the value of a semaphore, the operation is blocked if the semaphore value is too low—that is, if some other process has already reserved the resource.

When a *semop()* call blocks, the process remains blocked until one of the following occurs:

- Another process modifies the value of the semaphore such that the requested operation can proceed.
- A signal interrupts the *semop()* call. In this case, the error `EINTR` results. (As noted in Section 21.5, *semop()* is never automatically restarted after being interrupted by a signal handler.)
- Another process deletes the semaphore referred to by *semid*. In this case, *semop()* fails with the error `EIDRM`.

We can prevent *semop()* from blocking when performing an operation on a particular semaphore by specifying the `IPC_NOWAIT` flag in the corresponding *sem_flg* field. In this case, if *semop()* would have blocked, it instead fails with the error `EAGAIN`.

While it is usual to operate on a single semaphore at a time, it is possible to make a *semop()* call that performs operations on multiple semaphores in a set. The key point to note is that this group of operations is performed atomically; that is, *semop()* either performs all of the operations immediately, if possible, or blocks until it would be possible to perform all of the operations simultaneously.

> Few systems document the fact that *semop()* performs operations in array order, although all systems known to the author do so, and a few applications depend on this behavior. SUSv4 adds text that explicitly requires this behavior.

Listing 47-7 demonstrates the use of *semop()* to perform operations on three semaphores in a set. The operations on semaphores 0 and 2 may not be able to proceed immediately, depending on the current values of the semaphores. If the operation on semaphore 0 can't be performed immediately, then none of the requested operations is performed, and *semop()* blocks. On the other hand, if the operation on semaphore 0 could be performed immediately, but the operation on semaphore 2 could not, then—because the `IPC_NOWAIT` flag was specified—none of the requested operations is performed, and *semop()* returns immediately with the error `EAGAIN`.

The *semtimedop()* system call performs the same task as *semop()*, except that the *timeout* argument specifies an upper limit on the time for which the call will block.

```
#define _GNU_SOURCE
#include <sys/types.h>        /* For portability */
#include <sys/sem.h>

int semtimedop(int semid, struct sembuf *sops, unsigned int nsops,
               struct timespec *timeout);
```

Returns 0 on success, or −1 on error

The *timeout* argument is a pointer to a *timespec* structure (Section 23.4.2), which allows a time interval to be expressed as a number of seconds and nanoseconds. If the specified time interval expires before it is possible to complete the semaphore operation, *semtimedop()* fails with the error `EAGAIN`. If *timeout* is specified as `NULL`, *semtimedop()* is exactly the same as *semop()*.

The *semtimedop()* system call is provided as a more efficient method of setting a timeout on a semaphore operation than using *setitimer()* plus *semop()*. The small performance benefit that this confers is significant for certain applications (notably, some database systems) that need to frequently perform such operations. However, *semtimedop()* is not specified in SUSv3 and is present on only a few other UNIX implementations.

> The *semtimedop()* system call appeared as a new feature in Linux 2.6 and was subsequently back-ported into Linux 2.4, starting with kernel 2.4.22.

Listing 47-7: Using *semop()* to perform operations on multiple System V semaphores

```
struct sembuf sops[3];

sops[0].sem_num = 0;                /* Subtract 1 from semaphore 0 */
sops[0].sem_op = -1;
sops[0].sem_flg = 0;

sops[1].sem_num = 1;                /* Add 2 to semaphore 1 */
sops[1].sem_op = 2;
sops[1].sem_flg = 0;

sops[2].sem_num = 2;                /* Wait for semaphore 2 to equal 0 */
sops[2].sem_op = 0;
sops[2].sem_flg = IPC_NOWAIT;       /* But don't block if operation
                                       can't be performed immediately */
if (semop(semid, sops, 3) == -1) {
    if (errno == EAGAIN)            /* Semaphore 2 would have blocked */
        printf("Operation would have blocked\n");
    else
        errExit("semop");           /* Some other error */
}
```

Example program

The program in Listing 47-8 provides a command-line interface to the *semop()* system call. The first argument to this program is the identifier of the semaphore set upon which operations are to be performed.

Each of the remaining command-line arguments specifies a group of semaphore operations to be performed in a single *semop()* call. The operations within a single command-line argument are delimited by commas. Each operation has one of the following forms:

- *semnum*+*value*: add *value* to semaphore *semnum*.
- *semnum*-*value*: subtract *value* from semaphore *semnum*.
- *semnum*=0: test semaphore *semnum* to see if it equals 0.

At the end of each operation, we can optionally include an *n*, a *u*, or both. The letter *n* means include `IPC_NOWAIT` in the *sem_flg* value for this operation. The letter *u* means include `SEM_UNDO` in *sem_flg*. (We describe the `SEM_UNDO` flag in Section 47.8.)

The following command line specifies two *semop()* calls on the semaphore set whose identifier is 0:

```
$ ./svsem_op 0 0=0 0-1,1-2n
```

The first command-line argument specifies a *semop()* call that waits until semaphore zero equals 0. The second argument specifies a *semop()* call that subtracts 1 from semaphore 0, and subtracts 2 from semaphore 1. For the operation on semaphore 0, *sem_flg* is 0; for the operation on semaphore 1, *sem_flg* is `IPC_NOWAIT`.

Listing 47-8: Performing System V semaphore operations with *semop()*

svsem/svsem_op.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include <ctype.h>
#include "curr_time.h"          /* Declaration of currTime() */
#include "tlpi_hdr.h"

#define MAX_SEMOPS 1000         /* Maximum operations that we permit for
                                   a single semop() */

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s semid op[,op...] ...\n\n", progName);
    fprintf(stderr, "'op' is either: <sem#>{+|-}<value>[n][u]\n");
    fprintf(stderr, "            or: <sem#>=0[n]\n");
    fprintf(stderr, "       \"n\" means include IPC_NOWAIT in 'op'\n");
    fprintf(stderr, "       \"u\" means include SEM_UNDO in 'op'\n\n");
    fprintf(stderr, "The operations in each argument are "
                    "performed in a single semop() call\n\n");
    fprintf(stderr, "e.g.: %s 12345 0+1,1-2un\n", progName);
    fprintf(stderr, "      %s 12345 0=0n 1+1,2-1u 1=0\n", progName);
    exit(EXIT_FAILURE);
}

/* Parse comma-delimited operations in 'arg', returning them in the
   array 'sops'. Return number of operations as function result. */

static int
parseOps(char *arg, struct sembuf sops[])
{
    char *comma, *sign, *remaining, *flags;
    int numOps;                 /* Number of operations in 'arg' */

    for (numOps = 0, remaining = arg; ; numOps++) {
        if (numOps >= MAX_SEMOPS)
            cmdLineErr("Too many operations (maximum=%d): \"%s\"\n",
                       MAX_SEMOPS, arg);

        if (*remaining == '\0')
            fatal("Trailing comma or empty argument: \"%s\"", arg);
        if (!isdigit((unsigned char) *remaining))
            cmdLineErr("Expected initial digit: \"%s\"\n", arg);

        sops[numOps].sem_num = strtol(remaining, &sign, 10);

        if (*sign == '\0' || strchr("+-=", *sign) == NULL)
            cmdLineErr("Expected '+', '-', or '=' in \"%s\"\n", arg);
        if (!isdigit((unsigned char) *(sign + 1)))
            cmdLineErr("Expected digit after '%c' in \"%s\"\n", *sign, arg);

        sops[numOps].sem_op = strtol(sign + 1, &flags, 10);
```

```
        if (*sign == '-')                       /* Reverse sign of operation */
            sops[numOps].sem_op = - sops[numOps].sem_op;
        else if (*sign == '=')                  /* Should be '=0' */
            if (sops[numOps].sem_op != 0)
                cmdLineErr("Expected \"=0\" in \"%s\"\n", arg);

        sops[numOps].sem_flg = 0;
        for (;; flags++) {
            if (*flags == 'n')
                sops[numOps].sem_flg |= IPC_NOWAIT;
            else if (*flags == 'u')
                sops[numOps].sem_flg |= SEM_UNDO;
            else
                break;
        }

        if (*flags != ',' && *flags != '\0')
            cmdLineErr("Bad trailing character (%c) in \"%s\"\n", *flags, arg);

        comma = strchr(remaining, ',');
        if (comma == NULL)
            break;                              /* No comma --> no more ops */
        else
            remaining = comma + 1;
    }

    return numOps + 1;
}

int
main(int argc, char *argv[])
{
    struct sembuf sops[MAX_SEMOPS];
    int ind, nsops;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageError(argv[0]);

    for (ind = 2; argv[ind] != NULL; ind++) {
        nsops = parseOps(argv[ind], sops);

        printf("%5ld, %s: about to semop()  [%s]\n", (long) getpid(),
                currTime("%T"), argv[ind]);

        if (semop(getInt(argv[1], 0, "semid"), sops, nsops) == -1)
            errExit("semop (PID=%ld)", (long) getpid());

        printf("%5ld, %s: semop() completed [%s]\n", (long) getpid(),
                currTime("%T"), argv[ind]);
    }

    exit(EXIT_SUCCESS);
}
```

svsem/svsem_op.c

Using the program in Listing 47-8, along with various others shown in this chapter, we can study the operation of System V semaphores, as demonstrated in the following shell session. We begin by using a program that creates a semaphore set containing two semaphores, which we initialize to 1 and 0:

```
$ ./svsem_create -p 2
32769                                          ID of semaphore set
$ ./svsem_setall 32769 1 0
Semaphore values changed (PID=3658)
```

> We don't show the code of the `svsem/svsem_create.c` program in this chapter, but it is provided in the source code distribution for this book. This program performs the same function for semaphores as the program in Listing 46-1 (on page 938) performs for message queues; that is, it creates a semaphore set. The only notable difference is that `svsem_create.c` takes an additional argument specifying the size of the semaphore set to be created.

Next, we start three background instances of the program in Listing 47-8 to perform *semop()* operations on the semaphore set. The program prints messages before and after each semaphore operation. These messages include the time, so that we can see when each operation starts and when it completes, and the process ID, so that we can track the operation of multiple instances of the program. The first command makes a request to decrease both semaphores by 1:

```
$ ./svsem_op 32769 0-1,1-1 &                   Operation 1
 3659, 16:02:05: about to semop()  [0-1,1-1]
[1] 3659
```

In the above output, we see that the program printed a message saying that the *semop()* operation is about to be performed, but did not print any further messages, because the *semop()* call blocks. The call blocks because semaphore 1 has the value 0.

Next, we execute a command that makes a request to decrease semaphore 1 by 1:

```
$ ./svsem_op 32769 1-1 &                       Operation 2
 3660, 16:02:22: about to semop()  [1-1]
[2] 3660
```

This command also blocks. Next, we execute a command that waits for the value of semaphore 0 to equal 0:

```
$ ./svsem_op 32769 0=0 &                       Operation 3
 3661, 16:02:27: about to semop()  [0=0]
[3] 3661
```

Again, this command blocks, in this case because the value of semaphore 0 is currently 1.

Now, we use the program in Listing 47-3 to inspect the semaphore set:

```
$ ./svsem_mon 32769
Semaphore changed: Sun Jul 25 16:01:53 2010
Last semop():      Thu Jan  1 01:00:00 1970
Sem #  Value  SEMPID  SEMNCNT  SEMZCNT
  0       1       0       1        1
  1       0       0       2        0
```

When a semaphore set is created, the *sem_otime* field of the associated *semid_ds* data structure is initialized to 0. A calendar time value of 0 corresponds to the Epoch (Section 10.1), and *ctime()* displays this as 1 AM, 1 January 1970, since the local timezone is Central Europe, one hour ahead of UTC.

Examining the output further, we can see that, for semaphore 0, the *semncnt* value is 1 because operation 1 is waiting to decrease the semaphore value, and *semzcnt* is 1 because operation 3 is waiting for this semaphore to equal 0. For semaphore 1, the *semncnt* value of 2 reflects the fact that operation 1 and operation 2 are waiting to decrease the semaphore value.

Next, we try a nonblocking operation on the semaphore set. This operation waits for semaphore 0 to equal 0. Since this operation can't be immediately performed, *semop()* fails with the error `EAGAIN`:

```
$ ./svsem_op 32769 0=0n                                    Operation 4
 3673, 16:03:13: about to semop()  [0=0n]
ERROR [EAGAIN/EWOULDBLOCK Resource temporarily unavailable] semop (PID=3673)
```

Now we add 1 to semaphore 1. This causes two of the earlier blocked operations (1 and 3) to unblock:

```
$ ./svsem_op 32769 1+1                                     Operation 5
 3674, 16:03:29: about to semop()  [1+1]
 3659, 16:03:29: semop() completed [0-1,1-1]               Operation 1 completes
 3661, 16:03:29: semop() completed [0=0]                   Operation 3 completes
 3674, 16:03:29: semop() completed [1+1]                   Operation 5 completes
[1]   Done              ./svsem_op 32769 0-1,1-1
[3]+  Done              ./svsem_op 32769 0=0
```

When we use our monitoring program to inspect the state of the semaphore set, we see that the *sem_otime* field of the associated *semid_ds* data structure has been updated, and the *sempid* values of both semaphores have been updated. We also see that the *semncnt* value for semaphore 1 is 1, since operation 2 is still blocked, waiting to decrease the value of this semaphore:

```
$ ./svsem_mon 32769
Semaphore changed: Sun Jul 25 16:01:53 2010
Last semop():      Sun Jul 25 16:03:29 2010
Sem #  Value  SEMPID  SEMNCNT  SEMZCNT
  0       0    3661      0        0
  1       0    3659      1        0
```

From the above output, we see that the *sem_otime* value has been updated. We also see that semaphore 0 was last operated on by process ID 3661 (operation 3) and semaphore 1 was last operated on by process ID 3659 (operation 1).

Finally, we remove the semaphore set. This causes the still blocked operation 2 to fail with the error `EIDRM`:

```
$ ./svsem_rm 32769
ERROR [EIDRM Identifier removed] semop (PID=3660)
```

> We don't show the source code for the `svsem/svsem_rm.c` program in this chapter, but it is provided in the source code distribution for this book. This program removes the semaphore set identified by its command-line argument.

47.7 Handling of Multiple Blocked Semaphore Operations

If multiple processes are blocked trying to decrease the value of a semaphore by the same amount, then it is indeterminate which process will be permitted to perform the operation first when it becomes possible (i.e., which process is able to perform the operation will depend on vagaries of the kernel process scheduling algorithm).

On the other hand, if processes are blocked trying to decrease a semaphore value by different amounts, then the requests are served in the order in which they become possible. Suppose that a semaphore currently has the value 0, and process A requests to decrease the semaphore's value by 2, and then process B requests to decrease the value by 1. If a third process then adds 1 to the semaphore, process B would be the first to unblock and perform its operation, even though process A was the first to request an operation against the semaphore. In poorly designed applications, such scenarios can lead to *starvation*, whereby a process remains blocked forever because the state of the semaphore is never such that the requested operation proceeds. Continuing our example, we can envisage scenarios where multiple processes adjust the semaphore in such a way that its value is never more than 1, with the result that process A remains blocked forever.

Starvation can also occur if a process is blocked trying to perform operations on multiple semaphores. Consider the following scenario, performed on a pair of semaphores, both of which initially have the value 0:

1. Process A makes a request to subtract 1 from semaphores 0 and 1 (*blocks*).
2. Process B makes a request to subtract 1 from semaphore 0 (*blocks*).
3. Process C adds 1 to semaphore 0.

At this point, process B unblocks and completes its request, even though it placed its request later than process A. Again, it is possible to devise scenarios in which process A is starved while other processes adjust and block on the values of the individual semaphores.

47.8 Semaphore Undo Values

Suppose that, having adjusted the value of a semaphore (e.g., decreased the semaphore value so that it is now 0), a process then terminates, either deliberately or accidentally. By default, the semaphore's value is left unchanged. This may constitute a problem for other processes using the semaphore, since they may be blocked waiting on that semaphore—that is, waiting for the now-terminated process to undo the change it made.

To avoid such problems, we can employ the `SEM_UNDO` flag when changing the value of a semaphore via *semop()*. When this flag is specified, the kernel records the effect of the semaphore operation, and then undoes the operation if the process terminates. The undo happens regardless of whether the process terminates normally or abnormally.

The kernel doesn't need to keep a record of all operations performed using SEM_UNDO. It suffices to record the *sum* of all of the semaphore adjustments performed using SEM_UNDO in a per-semaphore, per-process integer total called the *semadj* (semaphore adjustment) value. When the process terminates, all that is necessary is to subtract this total from the semaphore's current value.

> Since Linux 2.6, processes (threads) created using *clone()* share *semadj* values if the CLONE_SYSVSEM flag is employed. Such sharing is required for a conforming implementation of POSIX threads. The NPTL threading implementation employs CLONE_SYSVSEM for the implementation of *pthread_create()*.

When a semaphore value is set using the *semctl()* SETVAL or SETALL operation, the corresponding *semadj* values are cleared (i.e., set to 0) in all processes using the semaphore. This makes sense, since absolutely setting the value of a semaphore destroys the value associated with the historical record maintained in the *semadj* total.

A child created via *fork()* doesn't inherit its parent's *semadj* values; it doesn't make sense for a child to undo its parent's semaphore operations. On the other hand, *semadj* values are preserved across an *exec()*. This permits us to adjust a semaphore value using SEM_UNDO, and then *exec()* a program that performs no operation on the semaphore, but does automatically adjust the semaphore on process termination. (This can be used as a technique that allows another process to discover when this process terminates.)

Example of the effect of SEM_UNDO

The following shell session log shows the effect of performing operations on two semaphores: one operation with the SEM_UNDO flag and one without. We begin by creating a set containing two semaphores:

```
$ ./svsem_create -p 2
131073
```

Next, we execute a command that adds 1 to both semaphores and then terminates. The operation on semaphore 0 specifies the SEM_UNDO flag:

```
$ ./svsem_op 131073 0+1u 1+1
 2248, 06:41:56: about to semop()
 2248, 06:41:56: semop() completed
```

Now, we use the program in Listing 47-3 to check the state of the semaphores:

```
$ ./svsem_mon 131073
Semaphore changed: Sun Jul 25 06:41:34 2010
Last semop():      Sun Jul 25 06:41:56 2010
Sem #  Value  SEMPID  SEMNCNT  SEMZCNT
  0       0    2248      0        0
  1       1    2248      0        0
```

Looking at the semaphore values in the last two lines of the above output, we can see that the operation on semaphore 0 was undone, but the operation on semaphore 1 was not undone.

Limitations of SEM_UNDO

We conclude by noting that the SEM_UNDO flag is less useful than it first appears, for two reasons. One is that because modifying a semaphore typically corresponds to acquiring or releasing some shared resource, the use of SEM_UNDO on its own may be insufficient to allow a multiprocess application to recover in the event that a process unexpectedly terminates. Unless process termination also automatically returns the shared resource state to a consistent state (unlikely in many scenarios), undoing a semaphore operation is probably insufficient to allow the application to recover.

The second factor limiting the utility of SEM_UNDO is that, in some cases, it is not possible to perform semaphore adjustments when a process terminates. Consider the following scenario, applied to a semaphore whose initial value is 0:

1. Process A increases the value of a semaphore by 2, specifying the SEM_UNDO flag for the operation.
2. Process B decreases the value of the semaphore by 1, so that it has the value 1.
3. Process A terminates.

At this point, it is impossible to completely undo the effect of process A's operation in step 1, since the value of the semaphore is too low. There are three possible ways to resolve this situation:

- Force the process to block until the semaphore adjustment is possible.
- Decrease the semaphore value as far as possible (i.e., to 0) and exit.
- Exit without performing any semaphore adjustment.

The first solution is infeasible since it might force a terminating process to block forever. Linux adopts the second solution. Some other UNIX implementations adopt the third solution. SUSv3 is silent on what an implementation should do in this situation.

> An undo operation that attempts to raise a semaphore's value above its permitted maximum value of 32,767 (the SEMVMX limit, described Section 47.10) also causes anomalous behavior. In this case, the kernel always performs the adjustment, thus (illegitimately) raising the semaphore's value above SEMVMX.

47.9 Implementing a Binary Semaphores Protocol

The API for System V semaphores is complex, both because semaphore values can be adjusted by arbitrary amounts, and because semaphores are allocated and operated upon in sets. Both of these features provide more functionality than is typically needed within applications, and so it is useful to implement some simpler protocols (APIs) on top of System V semaphores.

One commonly used protocol is binary semaphores. A binary semaphore has two values: *available* (free) and *reserved* (in use). Two operations are defined for binary semaphores:

- *Reserve*: Attempt to reserve this semaphore for exclusive use. If the semaphore is already reserved by another process, then block until the semaphore is released.

- *Release*: Free a currently reserved semaphore, so that it can be reserved by another process.

> In academic computer science, these two operations often go by the names *P* and *V*, the first letters of the Dutch terms for these operations. This nomenclature was coined by the late Dutch computer scientist Edsger Dijkstra, who produced much of the early theoretical work on semaphores. The terms *down* (decrement the semaphore) and *up* (increment the semaphore) are also used. POSIX terms the two operations *wait* and *post*.

A third operation is also sometimes defined:

- *Reserve conditionally*: Make a nonblocking attempt to reserve this semaphore for exclusive use. If the semaphore is already reserved, then immediately return a status indicating that the semaphore is unavailable.

In implementing binary semaphores, we must choose how to represent the *available* and *reserved* states, and how to implement the above operations. A moment's reflection leads us to realize that the best way to represent the states is to use the value 1 for *free* and the value 0 for *reserved*, with the *reserve* and *release* operations decrementing and incrementing the semaphore value by one.

Listing 47-9 and Listing 47-10 provide an implementation of binary semaphores using System V semaphores. As well as providing the prototypes of the functions in the implementation, the header file in Listing 47-9 declares two global Boolean variables exposed by the implementation. The *bsUseSemUndo* variable controls whether the implementation uses the `SEM_UNDO` flag in *semop()* calls. The *bsRetryOnEintr* variable controls whether the implementation restarts *semop()* calls that are interrupted by signals.

Listing 47-9: Header file for `binary_sems.c`

svsem/binary_sems.h

```
#ifndef BINARY_SEMS_H           /* Prevent accidental double inclusion */
#define BINARY_SEMS_H

#include "tlpi_hdr.h"

/* Variables controlling operation of functions below */

extern Boolean bsUseSemUndo;            /* Use SEM_UNDO during semop()? */
extern Boolean bsRetryOnEintr;          /* Retry if semop() interrupted by
                                           signal handler? */

int initSemAvailable(int semId, int semNum);

int initSemInUse(int semId, int semNum);

int reserveSem(int semId, int semNum);

int releaseSem(int semId, int semNum);

#endif
```

svsem/binary_sems.h

Listing 47-10 shows the implementation of the binary semaphore functions. Each function in this implementation takes two arguments, which identify a semaphore set and the number of a semaphore within that set. (These functions don't deal with the creation and deletion of semaphore sets; nor do they handle the race condition described in Section 47.5.) We employ these functions in the example programs shown in Section 48.4.

Listing 47-10: Implementing binary semaphores using System V semaphores

svsem/binary_sems.c

```
#include <sys/types.h>
#include <sys/sem.h>
#include "semun.h"                      /* Definition of semun union */
#include "binary_sems.h"

Boolean bsUseSemUndo = FALSE;
Boolean bsRetryOnEintr = TRUE;

int                     /* Initialize semaphore to 1 (i.e., "available") */
initSemAvailable(int semId, int semNum)
{
    union semun arg;

    arg.val = 1;
    return semctl(semId, semNum, SETVAL, arg);
}

int                     /* Initialize semaphore to 0 (i.e., "in use") */
initSemInUse(int semId, int semNum)
{
    union semun arg;

    arg.val = 0;
    return semctl(semId, semNum, SETVAL, arg);
}

/* Reserve semaphore (blocking), return 0 on success, or -1 with 'errno'
   set to EINTR if operation was interrupted by a signal handler */

int                     /* Reserve semaphore - decrement it by 1 */
reserveSem(int semId, int semNum)
{
    struct sembuf sops;

    sops.sem_num = semNum;
    sops.sem_op = -1;
    sops.sem_flg = bsUseSemUndo ? SEM_UNDO : 0;

    while (semop(semId, &sops, 1) == -1)
        if (errno != EINTR || !bsRetryOnEintr)
            return -1;

    return 0;
}
```

```
int                     /* Release semaphore - increment it by 1 */
releaseSem(int semId, int semNum)
{
    struct sembuf sops;

    sops.sem_num = semNum;
    sops.sem_op = 1;
    sops.sem_flg = bsUseSemUndo ? SEM_UNDO : 0;

    return semop(semId, &sops, 1);
}
```

svsem/binary_sems.c

47.10 Semaphore Limits

Most UNIX implementations impose various limits on the operation of System V semaphores. The following is a list of the Linux semaphore limits. The system call affected by the limit and the error that results if the limit is reached are noted in parentheses.

SEMAEM
: This is the maximum value that can be recorded in a *semadj* total. SEMAEM is defined to have the same value as SEMVMX (described below). (*semop()*, ERANGE)

SEMMNI
: This is a system-wide limit on the number of semaphore identifiers (in other words, semaphore sets) that can be created. (*semget()*, ENOSPC)

SEMMSL
: This is the maximum number of semaphores that can be allocated in a semaphore set. (*semget()*, EINVAL)

SEMMNS
: This is a system-wide limit on the number of semaphores in all semaphore sets. The number of semaphores on the system is also limited by SEMMNI and SEMMSL; in fact, the default value for SEMMNS is the product of the defaults for these two limits. (*semget()*, ENOSPC)

SEMOPM
: This is the maximum number of operations per *semop()* call. (*semop()*, E2BIG)

SEMVMX
: This is the maximum value for a semaphore. (*semop()*, ERANGE)

The limits above appear on most UNIX implementations. Some UNIX implementations (but not Linux) impose the following additional limits relating to semaphore undo operations (Section 47.8):

SEMMNU
: This is a system-wide limit on the total number of semaphore undo structures. Undo structures are allocated to store *semadj* values.

SEMUME
: This is the maximum number of undo entries per semaphore undo structure.

At system startup, the semaphore limits are set to default values. These defaults may vary across kernel versions. (Some distributors' kernels set different defaults from those provided by vanilla kernels.) Some of these limits can be modified by changing the values stored in the Linux-specific /proc/sys/kernel/sem file. This file contains four space-delimited numbers defining, in order, the limits SEMMSL, SEMMNS, SEMOPM, and SEMMNI. (The SEMVMX and SEMAEM limits can't be changed; both are fixed at 32,767.) As an example, here are the default limits that we see for Linux 2.6.31 on one x86-32 system:

```
$ cd /proc/sys/kernel
$ cat sem
250     32000   32      128
```

> The formats employed in the Linux /proc file system are inconsistent for the three System V IPC mechanisms. For message queues and shared memory, each configurable limit is controlled by a separate file. For semaphores, one file holds all configurable limits. This is a historical accident that occurred during the development of these APIs and is difficult to rectify for compatibility reasons.

Table 47-1 shows the maximum value to which each limit can be raised on the x86-32 architecture. Note the following supplementary information to this table:

- It is possible to raise SEMMSL to values larger than 65,536, and create semaphore sets up to that larger size. However, it isn't possible to use *semop()* to adjust semaphores in the set beyond the 65,536th element.

  > Because of certain limitations in the current implementation, the practical recommended upper limit on the size of a semaphore set is around 8000.

- The practical ceiling for the SEMMNS limit is governed by the amount of RAM available on the system.
- The ceiling value for the SEMOPM limit is determined by memory allocation primitives used within the kernel. The recommended maximum is 1000. In practical usage, it is rarely useful to perform more than a few operations in a single *semop()* call.

Table 47-1: System V semaphore limits

Limit	Ceiling value (x86-32)
SEMMNI	32768 (IPCMNI)
SEMMSL	65536
SEMMNS	2147483647 (INT_MAX)
SEMOPM	See text

The Linux-specific *semctl()* IPC_INFO operation retrieves a structure of type *seminfo*, which contains the values of the various semaphore limits:

```
union semun arg;
struct seminfo buf;

arg.__buf = &buf;
semctl(0, 0, IPC_INFO, arg);
```

A related Linux-specific operation, SEM_INFO, retrieves a *seminfo* structure that contains information about actual resources used for semaphore objects. An example of the use of SEM_INFO is provided in the file svsem/svsem_info.c in the source code distribution for this book.

Details about IPC_INFO, SEM_INFO, and the *seminfo* structure can be found in the *semctl(2)* manual page.

47.11 Disadvantages of System V Semaphores

System V semaphores have many of the same disadvantages as message queues (Section 46.9), including the following:

- Semaphores are referred to by identifiers, rather than the file descriptors used by most other UNIX I/O and IPC mechanisms. This makes it difficult to perform operations such as simultaneously waiting both on a semaphore and on input from a file descriptor. (It is possible to resolve this difficulty by creating a child process or thread that operates on the semaphore and writes messages to a pipe monitored, along with other file descriptors, using one of the methods described in Chapter 63.)
- The use of keys, rather than filenames, to identify semaphores results in additional programming complexity.
- The use of separate system calls for creating and initializing semaphores means that, in some cases, we must do extra programming work to avoid race conditions when initializing a semaphore.
- The kernel doesn't maintain a count of the number of processes referring to a semaphore set. This complicates the decision about when it is appropriate to delete a semaphore set and makes it difficult to ensure that an unused set is deleted.
- The programming interface provided by System V is overly complex. In the common case, a program operates on a single semaphore. The ability to simultaneously operate on multiple semaphores in a set is unnecessary.
- There are various limits on the operation of semaphores. These limits are configurable, but if an application operates outside the range of the default limits, this nevertheless requires extra work when installing the application.

However, unlike the situation with message queues, there are fewer alternatives to System V semaphores, and consequently there are more situations in which we may choose to employ them. One alternative to the use of semaphores is record locking, which we describe in Chapter 55. Also, from kernel 2.6 onward, Linux supports the use of POSIX semaphores for process synchronization. We describe POSIX semaphores in Chapter 53.

47.12 Summary

System V semaphores allow processes to synchronize their actions. This is useful when a process must gain exclusive access to some shared resource, such as a region of shared memory.

Semaphores are created and operated upon in sets containing one or more semaphores. Each semaphore within a set is an integer whose value is always greater than or equal to 0. The *semop()* system call allows the caller to add an integer to a semaphore, subtract an integer from a semaphore, or wait for a semaphore to equal 0. The last two of these operations may cause the caller to block.

A semaphore implementation is not required to initialize the members of a new semaphore set, so an application must initialize the set after creating it. When any of a number of peer processes may try to create and initialize the semaphore, special care must be taken to avoid the race condition that results from the fact that these two steps are performed via separate system calls.

Where multiple processes are trying to decrease a semaphore by the same amount, it is indeterminate which process will actually be permitted to perform the operation first. However, where different processes are trying to decrease a semaphore by different amounts, the operations complete in the order in which they become possible, and we may need to take care to avoid scenarios where a process is starved because the semaphore value never reaches a level that would allow the process's operation to proceed.

The `SEM_UNDO` flag allows a process's semaphore operations to be automatically undone on process termination. This can be useful to avoid scenarios where a process accidentally terminates, leaving a semaphore in a state that causes other processes to block indefinitely waiting for the semaphore's value to be changed by the terminated process.

System V semaphores are allocated and operated upon in sets, and can be increased and decreased by arbitrary amounts. This provides more functionality than is needed by most applications. A common requirement is for individual binary semaphores, which take on only the values 0 and 1. We showed how to implement binary semaphores on top of System V semaphores.

Further information

[Bovet & Cesati, 2005] and [Maxwell, 1999] provide some background on the implementation of semaphores on Linux. [Dijkstra, 1968] is a classic early paper on semaphore theory.

47.13 Exercises

47-1. Experiment with the program in Listing 47-8 (`svsem_op.c`) to confirm your understanding of the *semop()* system call.

47-2. Modify the program in Listing 24-6 (`fork_sig_sync.c`, on page 528) to use semaphores instead of signals to synchronize the parent and child processes.

47-3. Experiment with the program in Listing 47-8 (`svsem_op.c`) and the other semaphore programs provided in this chapter to see what happens to the *sempid* value if an exiting process performs a `SEM_UNDO` adjustment to a semaphore.

47-4. Add a *reserveSemNB()* function to the code in Listing 47-10 (`binary_sems.c`) to implement the *reserve conditionally* operation, using the `IPC_NOWAIT` flag.

47-5. For the VMS operating system, Digital provided a synchronization method similar to a binary semaphore, called an *event flag*. An event flag has two possible values, *clear* and *set*, and the following four operations can be performed: *setEventFlag*, to set the flag; *clearEventFlag*, to clear the flag; *waitForEventFlag*, to block until the flag is set; and *getFlagState*, to obtain the current state of the flag. Devise an implementation of event flags using System V semaphores. This implementation will require two arguments for each of the functions above: a semaphore identifier and a semaphore number. (Consideration of the *waitForEventFlag* operation will lead you to realize that the values chosen for *clear* and *set* are not the obvious choices.)

47-6. Implement a binary semaphores protocol using named pipes. Provide functions to reserve, release, and conditionally reserve the semaphore.

47-7. Write a program, analogous to the program in Listing 46-6 (`svmsg_ls.c`, on page 953), that uses the *semctl()* `SEM_INFO` and `SEM_STAT` operations to obtain and display a list of all semaphore sets on the system.