Semaphores

Prof. Naga Kandasamy
ECE Department, Drexel University

This material has been derived from: Michael Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, No Starch Press, San Francisco, 2010.

Semaphores allow processes and threads to synchronize their accesses to shared memory. Two types of semaphores are supported in Linux:

- *Named Semaphores*: This type of semaphore has a name. By invoking *sem_open()* with the same name, unrelated processes can access the same semaphore.
- *Unnamed semaphores*: This type of semaphore does not have a name; instead it resides in 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 shared memory. When shared between threads, the semaphore may reside in an area of memory shared by threads; for example, on the heap or in a global variable.

Semaphore is a programming construct that allows threads to synchronize their accesses to shared memory. The semaphore is an integer whose value is not permitted to fall below zero. If a thread attempts to decrease the value of a semaphore below zero, then depending on the function used, the call either blocks or fails with an error indicating that the operation was not currently possible. Two basic operations can be performed on a semaphore:

- The *sem_wait()* operation decrements or locks the semaphore. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement, that is, until the semaphore value rises above zero.
- The *sem_post()* operation increments or unlocks the semaphore. If the semaphore's value consequently becomes greater than zero, then another thread blocked on that semaphore will be woken up and proceed to lock the semaphore.

Named Semaphores

The following functions are used when working with named semaphores.

- 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.

```
#include <fcntl.h> /* Defines the O_* constants */
#include <sys/stat.h> /* Defines the 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; *oflag* is a bit-mask that determines if we are opening an existing semaphore, or creating and opening a new semaphore; the *mode* argument is a bit mask that specifies the permissions to be placed on the new semaphore; and *value* specifies an initial value to be assigned to the new semaphore.

The relevant snippet from *psem_create.c* is presented here.

```
int flags |= O_CREAT |= O_EXCL;

/* Default permissions are rw------- and the
semaphore initialization value is 0 */
mode_t perms = S_IRUSR | S_IWUSR;

unsigned int value = 0;

/* Assume semaphore name is provided via argv[1] */
sem_t *sem = sem_open(argv[1], flags, perms, value);
```

- The *sem_close()* terminates the association that the calling process has with the semaphore and releases any resources that the system has associated with the semaphore for this process. Open named semaphores are also automatically closed upon process termination. Note that closing a semaphore does not delete it. For that purpose, we must use *sem_unlink()*.
- The *sem_unlink()* function removes the specified semaphore and marks the semaphore to be destroyed once all processes are done using it. See *psem_unlink.c* for more details.
- The *sem_wait()* function decrements the value of a semaphore. If the semaphore value is greater than zero, *sem_wait()* returns immediately; if the value is currently zero, *sem_wait()* blocks until the semaphore value rises above zero; at that time, the semaphore is then decremented and *sem_wait()* returns. See *psem_wait.c* for more details.
- The *sem_post()* function increases the value of a semaphore by one. If the semaphore's value was zero before the *sem_post()* call and some other process is blocked waiting to decrement the semaphore, then that process is woken up and its *sem_wait()* call proceeds to decrement the semaphore. If multiple processes are blocked in *sem_wait()* and if the processes are being scheduled using the default round-robin policy, it is indeterminate which one will be woken up and allowed to decrement the semaphore. See *psem_post.c* for more details.
- The *sem_getvalue()* function returns the current value of the semaphore. If one or more processes are currently blocked waiting to decrement the semaphore's value, then the value returned is zero.

The following shell session log demonstrates the use of the above-described programs. We begin by creating a semaphore called *my_sem* whose initial value is zero, and then start a program in the background that attempts to decrement the semaphore:

```
[nk78@xunil-03 sem]$ ./psem_create -c /my_sem
[nk78@xunil-03 sem]$ ls -al /dev/shm/ | grep nk78
-rw-------    1 nk78  domain users    32 Jul 26 15:26 sem.my_sem
[nk78@xunil-03 sem]$ ./psem_wait /my_sem &
[1] 28151
```

The program blocks in the background since the value of the semaphore is currently zero and cannot be decreased. We retrieve the value of the semaphore to check:

```
[nk78@xunil-03 sem]$ ./psem_getvalue /my_sem
Semaphore value = 0
```

We then execute the program that increments the semaphore, which causes the blocked *sem_wait()* in the background process to complete:

```
[nk78@xunil-03 sem]$ ./psem_post /my_sem
28160 sem_post () succeeded
28151 sem_wait () succeeded
[1]+  Done                    ./psem_wait /my_sem
[nk78@xunil-03 sem]$ ./psem_unlink /my_sem
[nk78@xunil-03 sem]$
```

We finally delete the semaphore from the file system:

```
[nk78@xunil-03 sem]$ ./psem_unlink /my_sem
```

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 shared memory. Unnamed semaphores are 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 in the program.
- A semaphore that is being shared between related processes doesn't need a name. If the parent process allocates an unnamed semaphore in a region of the shared memory map, than a child automatically inherits the mapping and thus the semaphore as part of the *fork()* operation.

In addition to the functions associated with named semaphores, two additional functions are needed for unnamed semaphores:

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

Note that the above functions must not be used with named semaphores.