POSIX Semaphores

Semaphores are used for process and thread synchronization. Semaphores are clubbed with message queues and shared memory under the Interprocess Communication (IPC) facilities in Unix-like systems such as Linux. There are two varieties of semaphores: the traditional System V semaphores and the newer POSIX semaphores. In this document, we will look at the POSIX semaphores.

POSIX semaphore calls are much simpler than the System V semaphore calls. However, System V semaphores are more widely available, particularly on older Unix-like systems. POSIX semaphores have been available on Linux systems post version 2.6 that use glibc.

There are two types of POSIX semaphores - named and unnamed. As the terminology suggests, named semaphores have a name, which is of the format /somename. The first character is a forward slash, followed by one or more characters, none of which is a slash. Programs using POSIX semaphores need to be linked with the pthread library.

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

sem_t *sem_open (const char *name, int oflag);
sem_t *sem_open (const char *name, int oflag, mode_t mode, unsigned int value);
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

sem_open opens an existing semaphore or creates a new semaphore and opens it for further operations. The first parameter, *name,* is the name of the semaphore. The *oflag* can have O_CREAT, in which case, the semaphore is created if it does not already exist. If both O_CREAT and O_EXCL are specified, the call gives an error, if the semaphore with the specified name already exists. If the *oflags* parameter has O_CREAT set, the second form of sem_open has to be used and two additional parameters, *mode* and *value* have to be specified. The *mode* parameter specifies the permissions for the semaphore, which are masked with the umask for the process, similar to the *mode* in the open system call for files. The last parameter, *value* is the initial value for the semaphore. If O_CREAT is specified in *oflag* and the semaphore already exists, both the *mode* and *value*parameters are ignored. sem_open returns a pointer to the semaphore on success. This pointer has to be used in the subsequent calls for the semaphore. If the call fails, sem_open returns SEM_FAILED and errno is set the appropriate error.

sem_init is the equivalent of sem_open for unnamed semaphores. sem_init initializes the semaphore pointed by *sem* with the *value.* The second argument *pshared* indicates whether the semaphore is shared between threads of a process or between processes. If *pshared* has the value 0, the semaphore is shared between threads of a process. If *pshared* has a nonzero value, it indicates that the semaphore is shared by processes. In that case the semaphore has to be placed in a shared memory segment which is attached to the concerned processes.

```
int sem_post (sem_t *sem);
```

sem_post increments the semaphore. It provides the V operation for the semaphore. It returns 0 on success and -1 on error.

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

sem_wait decrements the semaphore pointed by *sem*. If the semaphore value is non-zero, the decrement happens right away. If the semaphore value is zero, the call blocks till the time semaphore becomes greater than zero and the decrement is done. sem_wait returns zero on success and -1 on error.

sem_trywait is just like the sem_wait call, except that, if the semaphore value is zero, it does not block but returns immediately with errno set to EAGAIN.

sem_timedwait is also like the sem_wait call, except that, there is timer specified with the pointer, *abs_timeout*. If the semaphore value is greater than zero, it is decremented and the timeout value pointed by *abs_timeout* is not used. That is, in that case, the call works just like the sem_wait call. If the semaphore value is zero, the call blocks, the maximum duration of blocking being the time till the timer goes off.

```
int sem_getvalue (sem_t *sem, int *sval);
int sem_unlink (const char *name);
int sem_destroy (sem_t *sem);
```

sem_getvalue gets the value of semaphore pointed by *sem*. sem_unlink removes the semaphore associated with the *name*. sem_destroy destroys the unnamed semaphore pointed by *sem*.