**Semaphores**

Consider this: You want to meet the chairwomen of the department, but the secretary in guarding the door. If another student is already inside, you wait. As soon as the student exits, the secretary lets you in. Only one student can be visiting the chair. We could say the chair of the department is the Critical Resource/Section and the secretary is the semaphore. In systems, a binary semaphore is a kernel-maintained integer variable whose value is restricted to being 0 or 1 . When a process (Say A) arrives to access the critical section (CS) when no other process inside, the process requests the kernel for access. The kernel decrements this variable ( now it is 0) and the process goes ahead to access the CS. In the mean time, another process (say B) arrives to access the CS. Because the value is zero, the kernel makes it to wait until the value goes back to 1. This semaphore is called binary semaphore. There are semaphores that are not binary. Say, a restaurant has a capacity to serve only 20 people. If people inside exceeds 20, the new incoming people have to wait until there is a space. In this situation, you initialize the semaphore variable to 20. This is how mutual exclusion is implemented. Because kernel manages this variable, synchronization is easily implemented.

There are two types of POSIX semaphores : Named and Unnamed 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. We will not discuss unnamed semaphores as the p

POSIX semaphores 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.

**Named Semaphore:**

To work with a named semaphore, we use various functions :

* The sem\_open() function opens or creates a new 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. We generally use the sem\_unlink instead.
* The sem\_unlink() function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

On Linux, they are created as small POSIX shared memory objects with names of the form sem.name 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.

**Creating and/or Opening a Named Semaphore**

The sem\_open() function creates and opens a new named semaphore or opens an

existing semaphore. If sem\_open() is being used to open an existing semaphore, the call requires only two arguments: name and oflag.

#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

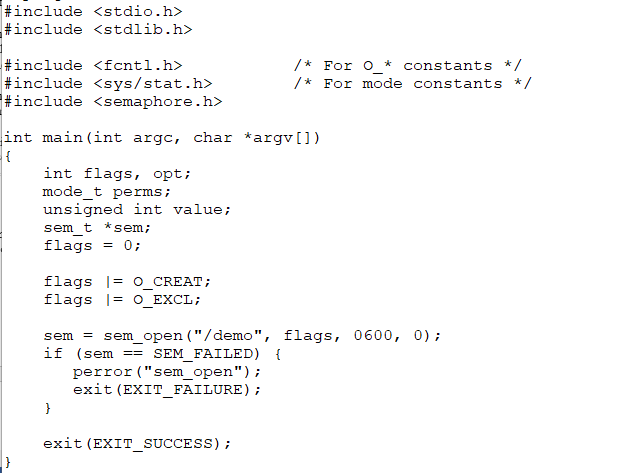
* The name argument identifies the semaphore.
* The oflag argument is a bit mask that determines whether we are opening an existing semaphore or creating and opening a new semaphore. If oflag is 0, we are accessing an existing semaphore and mode and value arguments are ignored and not needed.

If O\_CREAT is specified in flags, then two further arguments are required: mode and value.

* Mode: The mode argument is a bit mask that specifies the permissions to be placed on the new semaphore. 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. Typical value would be 666
* Value: 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.

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.

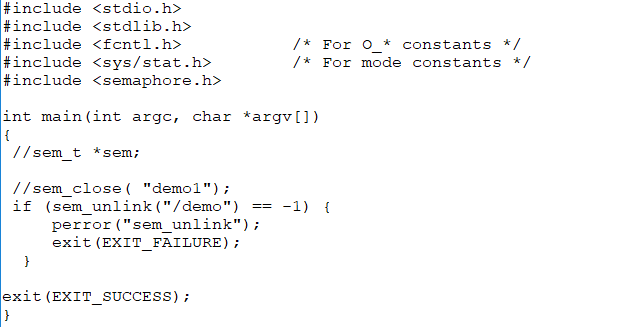
Regardless of whether we are creating a new semaphore or opening an existing semaphore, sem\_open() returns a pointer to a sem\_t value, and we use this pointer in subsequent calls to functions that operate on the semaphore. On error, sem\_open() returns the value SEM\_FAILED.



**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).

SAMPLE CODE:



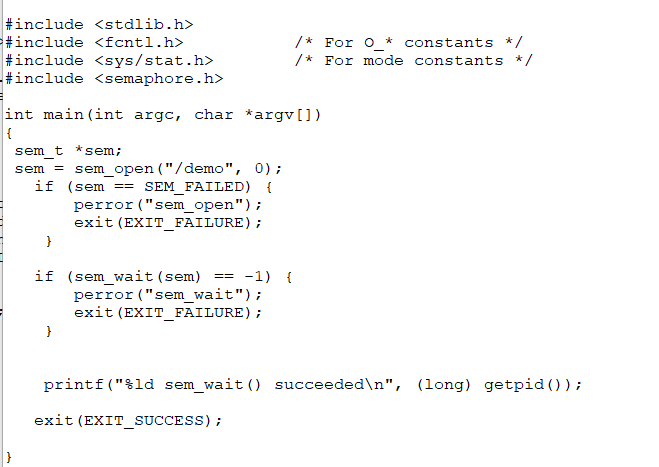
**Wait Operation on a Semaphore**

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);

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

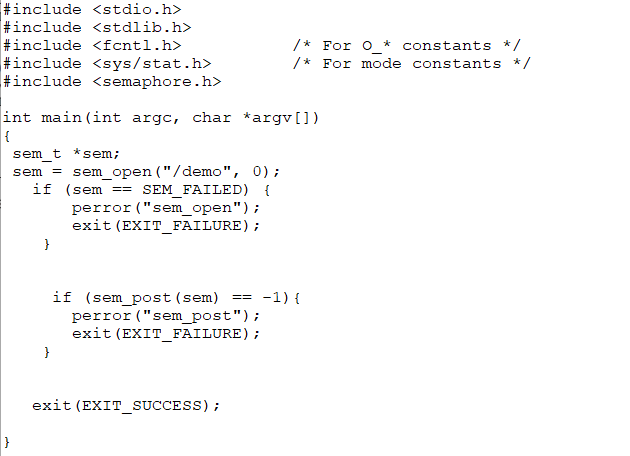


**Posting a Semaphore**

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

int **sem\_post**(sem\_t \*sem);

Posting is incrementing a POSIX semaphore. If there are processes waiting to access the CS, they are released. They all have to do a wait operation and one of them (depending on scheduling algorithms) will be given access to the CS.



**Getting Current Value of the Semaphore**

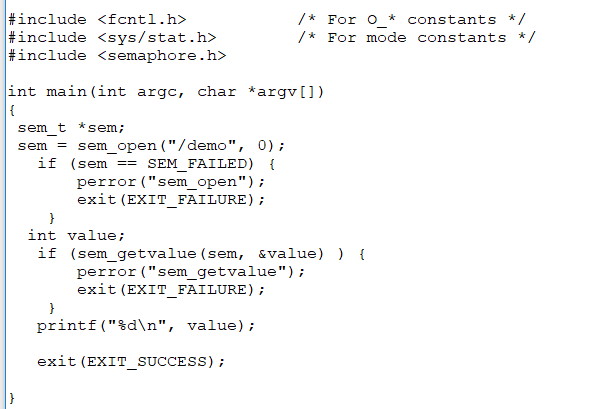
The sem\_getvalue() function returns the current value of the semaphore referred to

by sem in the int pointed to by sval.

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. The Unix standard 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. Note that by the time sem\_getvalue() returns, the value returned in sval may already be out of date.



**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. 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().

int **sem\_close**(sem\_t \*sem);

Returns 0 on success, or –1 on error

We will not discuss unnamed semaphores because they are little identical and can be easily used in applications.