Semaphores

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed.

Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter.

In the computer version, a semaphore appears to be a simple integer. A thread waits for permission to proceed and then signals that it has proceeded by performing a `P` operation on the semaphore.

The semantics of the operation are such that the thread must wait until the semaphore's value is positive, then change the semaphore's value by subtracting one from it. When it is finished, the thread performs a `V` operation, which changes the semaphore's value by adding one to it. It is crucial that these operations take place atomically—they cannot be subdivided into pieces between which other actions on the semaphore can take place. In the `P` operation, the semaphore's value must be positive just before it is decremented (resulting in a value that is guaranteed to be nonnegative and one less than what it was before it was decremented).

In both `P` and `V` operations, the arithmetic must take place without interference. If two `V` operations are performed simultaneously on the same semaphore, the net effect should be that the semaphore's new value is two greater than it was.

The mnemonic significance of `P` and `V` is unclear to most of the world, as Dijkstra is Dutch. However, in the interest of true scholarship: `P` stands for prolagen, a made-up word derived from proberen te verlagen, which means **try to decrease**. `V` stands for verhogen, which means **increase**. This is discussed in one of Dijkstra's technical notes, EWD 74.

`sem_wait`(3RT) and `sem_post`(3RT) correspond to Dijkstra's `P` and `V` operations. `sem_trywait`(3RT) is a conditional form of the `P` operation: if the calling thread cannot decrement the value of the semaphore without waiting, the call returns immediately with a nonzero value.

There are two basic sorts of semaphores: binary semaphores, which never take on values other than zero or one, and counting semaphores, which can take on arbitrary nonnegative values. A binary semaphore is logically just like a mutex.

However, although it is not enforced, mutexes should be unlocked only by the thread holding the lock. There is no notion of “the thread holding the semaphore,” so any thread can perform a `V` (or

Counting semaphores are about as powerful as conditional variables (used in conjunction with mutexes). In many cases, the code might be simpler when it is implemented with counting semaphores rather than with condition variables (as shown in the next few examples).

However, when a mutex is used with condition variables, there is an implied bracketing—it is clear which part of the program is being protected. This is not necessarily the case for a semaphore, which might be called the **go to** of concurrent programming—it is powerful but too easy to use in an unstructured, indeterminate way.

Counting Semaphores

Conceptually, a semaphore is a nonnegative integer count. Semaphores are typically used to coordinate access to resources, with the semaphore count initialized to the number of free resources. Threads then atomically increment the count when resources are added and atomically decrement the count when resources are removed.

When the semaphore count becomes zero, indicating that no more resources are present, threads trying to decrement the semaphore block wait until the count becomes greater than zero.

Table 4–7 Routines for Semaphores

Operation	Destination Discussion
Initialize a semaphore	sem_init(3RT)
Increment a semaphore	sem_post(3RT)
Block on a semaphore count	sem_wait(3RT)
Decrement a semaphore count	sem_trywait(3RT)
Destroy the semaphore state	sem_destroy(3RT)

Because semaphores need not be acquired and released by the same thread, they can be used for asynchronous event notification (such as in signal handlers). And, because semaphores contain state, they can be used asynchronously without acquiring a mutex lock as is required by condition variables. However, semaphores are not as efficient as mutex locks.

By default, there is no defined order of unblocking if multiple threads are waiting for a semaphore.

Semaphores must be initialized before use, but they do not have attributes.

Initialize a Semaphore

sem_init(3RT)

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

```
#include <semaphore.h>

sem_t sem;
int pshared;
int ret;
int value;

/* initialize a private semaphore */
pshared = 0;
value = 1;
ret = sem_init(&sem, pshared, value);
```

Use sema_init(3THR) to initialize the semaphore variable pointed to by *sem* to *value* amount. If the value of pshared is zero, then the semaphore cannot be shared between processes. If the value of pshared is nonzero, then the semaphore can be shared between processes. (For Solaris threads, see sema_init(3THR).)

Multiple threads must not initialize the same semaphore.

A semaphore must not be reinitialized while other threads might be using it.

Return Values

`sem_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

The value argument exceeds `SEM_VALUE_MAX`.

`ENOSPC`

A resource required to initialize the semaphore has been exhausted. The limit on semaphores `SEM_NSEMS_MAX` has been reached.

`EPERM`

The process lacks the appropriate privileges to initialize the semaphore.

Initializing Semaphores With Intraprocess Scope

When *pshared* is 0, the semaphore can be used by all the threads in this process only.

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be used within this process only */
ret = sem_init(&sem, 0, count);
```

Initializing Semaphores With Interprocess Scope

When *pshared* is nonzero, the semaphore can be shared by other processes.

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be shared among processes */
ret = sem_init(&sem, 1, count);
```

Named Semaphores

The functions `sem_open(3RT)`, `sem_getvalue(3RT)`, `sem_close(3RT)`, and `sem_unlink(3RT)` are available to `open`, `retrieve`, `close`, and `remove` named semaphores. Using **`sem_open()`**, you can create a semaphore that has a name defined in the file system name space.

Named semaphores are like process shared semaphores, except that they are referenced with a pathname rather than a *pshared* value.

For more information about named semaphores, see `sem_open(3RT)`, `sem_getvalue(3RT)`, `sem_close(3RT)`, and `sem_unlink(3RT)`.

Increment a Semaphore

sem_post(3RT)

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

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_post(&sem); /* semaphore is posted */
```

Use sema_post(3THR) to atomically increment the semaphore pointed to by *sem*. When any threads are blocked on the semaphore, one of them is unblocked. (For Solaris threads, see sema_post(3THR).)

Return Values

`sem_post()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

sem points to an illegal address.

Block on a Semaphore Count

sem_wait(3RT)

```
Prototype:
int     sem_wait(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_wait(&sem); /* wait for semaphore */
```

Use sema_wait(3THR) to block the calling thread until the count in the semaphore pointed to by *sem* becomes greater than zero, then atomically decrement it.

Return Values

`sem_wait()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

sem points to an illegal address.

`EINTR`

A signal interrupted this function.

Decrement a Semaphore Count

sem_trywait(3RT)

```
Prototype:
int     sem_trywait(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_trywait(&sem); /* try to wait for semaphore*/
```

Use sem_trywait(3RT) to try to atomically decrement the count in the semaphore pointed to by *sem* when the count is greater than zero. This function is a nonblocking version of **`sem_wait()`**; that is it returns immediately if unsuccessful.

Return Values

`sem_trywait()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

sem points to an illegal address.

`EINTR`

A signal interrupted this function.

`EAGAIN`

The semaphore was already locked, so it cannot be immediately locked by the **`sem_trywait()`** operation.

Destroy the Semaphore State

sem_destroy(3RT)

```
Prototype:
int     sem_destroy(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_destroy(&sem); /* the semaphore is destroyed */
```

Use sem_destroy(3RT) to destroy any state associated with the semaphore pointed to by *sem*. The space for storing the semaphore is not freed. (For Solaris threads, see sem_destroy(3THR).)

Return Values

`sem_destroy()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

sem points to an illegal address.

The Producer/Consumer Problem, Using Semaphores

The data structure in Example 4–14 is similar to that used for the condition variables example (see Example 4–11). Two semaphores represent the number of full and empty buffers and ensure that producers wait until there are empty buffers and that consumers wait until there are full buffers.

Example 4–14 The Producer/Consumer Problem With Semaphores

```
typedef struct {
    char buf[BSIZE];
    sem_t occupied;
    sem_t empty;
    int nextin;
    int nextout;
    sem_t pmut;
    sem_t cmut;
} buffer_t;

buffer_t buffer;

sem_init(&buffer.occupied, 0, 0);
sem_init(&buffer.empty,0, BSIZE);
sem_init(&buffer.pmut, 0, 1);
sem_init(&buffer.cmut, 0, 1);
buffer.nextin = buffer.nextout = 0;
```

Another pair of (binary) semaphores plays the same role as mutexes, controlling access to the buffer when there are multiple producers and multiple empty buffer slots, and when there are multiple consumers and multiple full buffer slots. Mutexes would work better here, but would not provide as good an example of semaphore use.

Example 4–15 The Producer/Consumer Problem—the Producer

```
void producer(buffer_t *b, char item) {
    sem_wait(&b->empty);
    sem_wait(&b->pmut);

    b->buf[b->nextin] = item;
    b->nextin++;
    b->nextin %= BSIZE;

    sem_post(&b->pmut);
    sem_post(&b->occupied);
}
```

Example 4–16 The Producer/Consumer Problem—the Consumer

```
char consumer(buffer_t *b) {
    char item;

    sem_wait(&b->occupied);

    sem_wait(&b->cmut);

    item = b->buf[b->nextout];
    b->nextout++;
    b->nextout %= BSIZE;

    sem_post(&b->cmut);

    sem_post(&b->empty);

    return(item);
}
```
