Operating Systems Lab
Semaphores

By: Muhammad Ahsan

What is a Semaphore?

1. Dijkestra proposed a significant technique for managing concurrent processes for complex mutual exclusion problems. He introduced a new synchronization tool called Semaphore.
2. Posix Semaphores are Inter Process or Threads synchronization technique, just like mutex.
3. A semaphore is an integer value variable, which can be incremented and unlocked or decremented and locked by threads or processes.

POSIX types of Semaphore:

Posiz Semaphores are of 2 types:

1. Named Semaphore
2. Memory-mapped semaphore

Unamed Semaphore(memory-based semaphore):

1. No name is associated with these semaphores
2. Provides synchronization between threads and between related processes
3. Placed in a region of main memory that is shared between processes/threads
4. For threads this is done by simply making the semaphore a global variable

Kind of Semaphores:

Depending upon the value a semaphore is made to hold, it can be:

1. Binary Semaphores
2. Counting Semaphores

Uses of Semaphores:

Semaphores can be use for synchronization between Threads/Processes. It also provide a way to avoid Dead-locks.

1. **For placing Locks**
   Just like mutex(can be binary or counting)
   (a) Counting Semaphores
   Permit a limited number of threads to execute a section of the code.
   (b) Binary Semaphores
   Permit only one thread to execute a section of the code.

2. **Semaphores As Condition Variables**
   (a) Semaphores are also useful when a thread wants to halt its progress waiting for a condition to become true.
   (b) Communicate information about the state of shared data.

Semaphores vs Mutex:

1. Mutex can be locked or unlocked, like binary semaphore. Semaphores (counting) can have multiple values.
2. A locked mutex can be unlocked by the thread holding the lock. A locked semaphore can be unlocked by any thread.
3. Semaphore has state (value of semaphore) associated with it.
4. Mutex and condition variables are used together in most scenario. Looking at their functionality, it can be thought as : Semaphore = Mutex + Condition Variable
5. Posix Named Semaphore are kernel persistent. Posix Memory based semaphore, Posix Condition Variable and Posix Mutex are process persistent.

Semaphores System Calls:

```
#include<semaphore.h>

int sem_init();

int sem_wait();

int sem_trywait();

int sem_post();

int sem_destroy();
```

Create a Semaphore:

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

1. **sem:** Target semaphore
2. **pshared:** The pshared argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.
   a) 0: only threads of the creating process can use the semaphore.
   b) Non-0: other processes can use the semaphore.
3. **value:** Initial value of the semaphore.

Example:

```
# include <semaphore . h>
sem_t s ;
sem_init(&s , 0 , 1)
```

We declare a semaphore s and initialize it to the value 1 by passing 1 in as the third argument. The second argument to sem init() will be set to 0 in all of the examples we'll see; this indicates t hat the semaphore is shared between threads in the same process.

Semaphore Operations:

1. **sem_wait()** decrements (locks) the semaphore pointed to by sem.
   int sem_wait (sem_t * sem)
   a) If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns(gets lock), immediately.
   b) If the value of the semaphore is negative, the calling thread blocks; one of the blocked threads wakes up when another thread calls sem_post()
2. **sem_post()** does not wait for some particular condition to hold like sem_wait() does.
   int sem_post (sem_t * sem)
   Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.

```
int   sem_wait ( sem_t*s )      {
          decrement   the  value   of   semaphore   s   by one

         wait if value        of   semaphore   s   is    negative

}

int   sem_post ( sem_t*s )
         increment    the    value    of   semaphore   s   by one
           if there     are   one   or   more    threads     waiting ,     wake one
}
```

3. **sem_trywait()** is the version of of the sem_wait() which does not block.
   int sem_trywait (sem_t * sem)
   Decreases the semaphore by one if the semaphore does not equal to zero. If it is zero it does not block, returns zero with error code EAGAIN.
4. **sem_destroy** releases the resources that semaphore has and destroys it
   int sem_destroy (sem_t * sem)