Mutex vs Semaphore

The concept of and the differences between a mutex and a semaphore will befuddle most developers. In this lesson, we discuss the differences between the two most fundamental concurrency constructs offered by almost all language frameworks. Difference between a mutex and a semaphore makes a pet interview question for senior engineering positions!

Mutex vs Semaphore

Having laid the foundation of concurrent programming concepts and their associated issues, we'll now discuss the all-important mechanisms of locking and signaling in multi-threaded applications and the differences between these constructs.

Mutex

Mutex as the name hints implies ***mutual exclusion***. A mutex is used to guard shared data such as a linked-list, an array, or any primitive type. A mutex allows only a single thread to access a resource or critical section.

Once a thread acquires a mutex, all other threads attempting to acquire the same mutex are blocked until the first thread releases the mutex. Once released, most implementations arbitrarily chose one of the waiting threads to acquire the mutex and make progress.

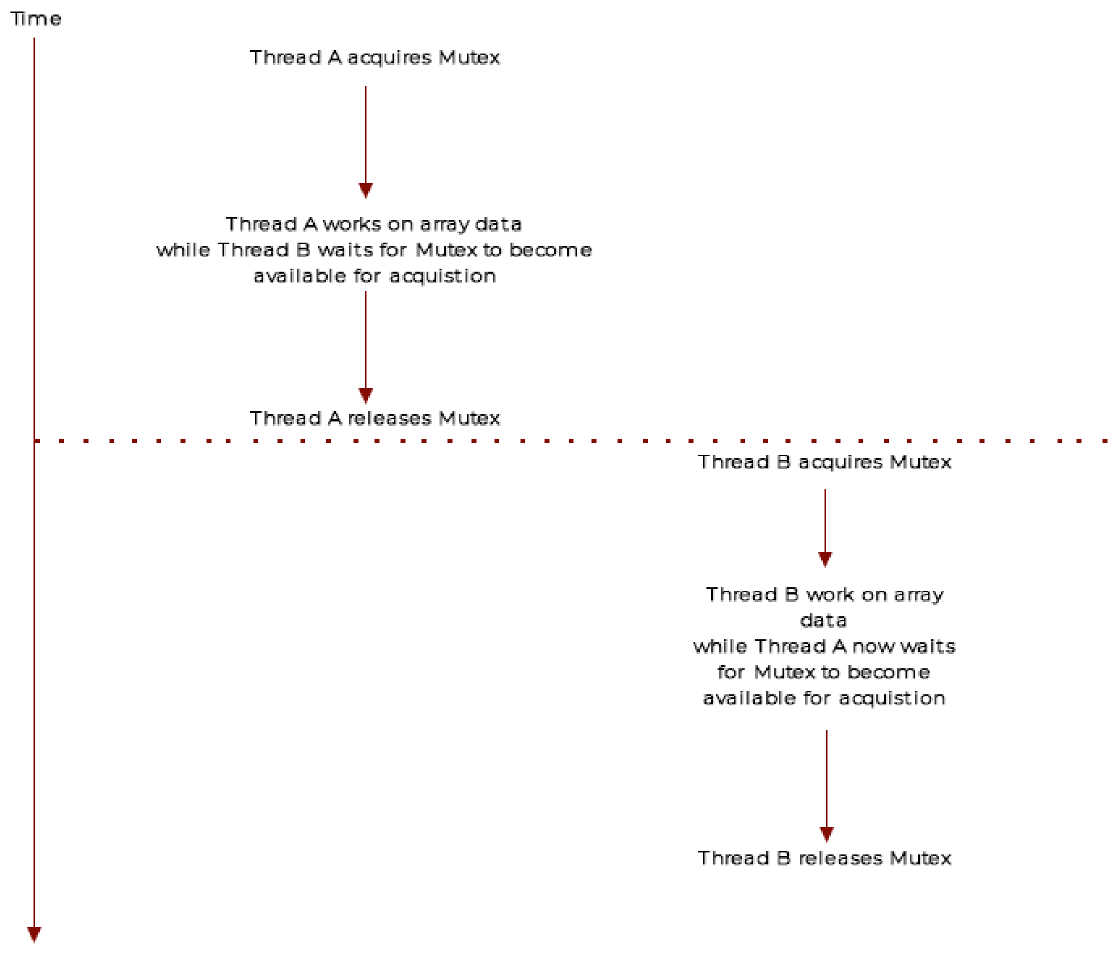
Semaphore

Semaphore, on the other hand, is used for limiting access to a collection of resources. Think of semaphore as having a limited number of permits to give out. If a semaphore has given out all the permits it has, then any new thread that comes along requesting a permit will be blocked till an earlier thread with a permit returns it to the semaphore. A typical example would be a pool of database connections that can be handed out to requesting threads. Say there are ten available connections but 50 requesting threads. In such a scenario, a semaphore can only give out ten permits or connections at any given point.

A semaphore with a single permit is called a **binary semaphore** and is often thought of as equivalent to a mutex, which isn't completely correct as we'll shortly explain. Semaphores can also be used for signaling among threads. This is an important distinction as it allows threads to work towards completing a task cooperatively. A mutex, on the other hand, is strictly limited to serializing access to shared state among competing threads.

Mutex Example

The following illustration shows how two threads acquire and release a mutex one after the other to gain access to shared data. Mutex guarantees that the shared state isn't corrupted when competing threads work on it.


Time
Thread A acquires Mutex
Thread A works on array data while Thread B waits for Mutex to become available for acquistion
Thread A releases Mutex
Thread B acquires Mutex
Thread B work on array data while Thread A now waits for Mutex to become available for acquistion
Thread B releases Mutex

When can a Semaphore masquerade as a Mutex?

A semaphore can potentially act as a mutex if the permits it can give out is set to 1. However, the most important difference between the two is that in case of a mutex ***the same thread must call acquire and subsequent release on the mutex*** whereas in case of a binary semaphore, ***different threads can call acquire and release on the semaphore***. The pthreads library documentation states this in the `pthread_mutex_unlock()` method's description.

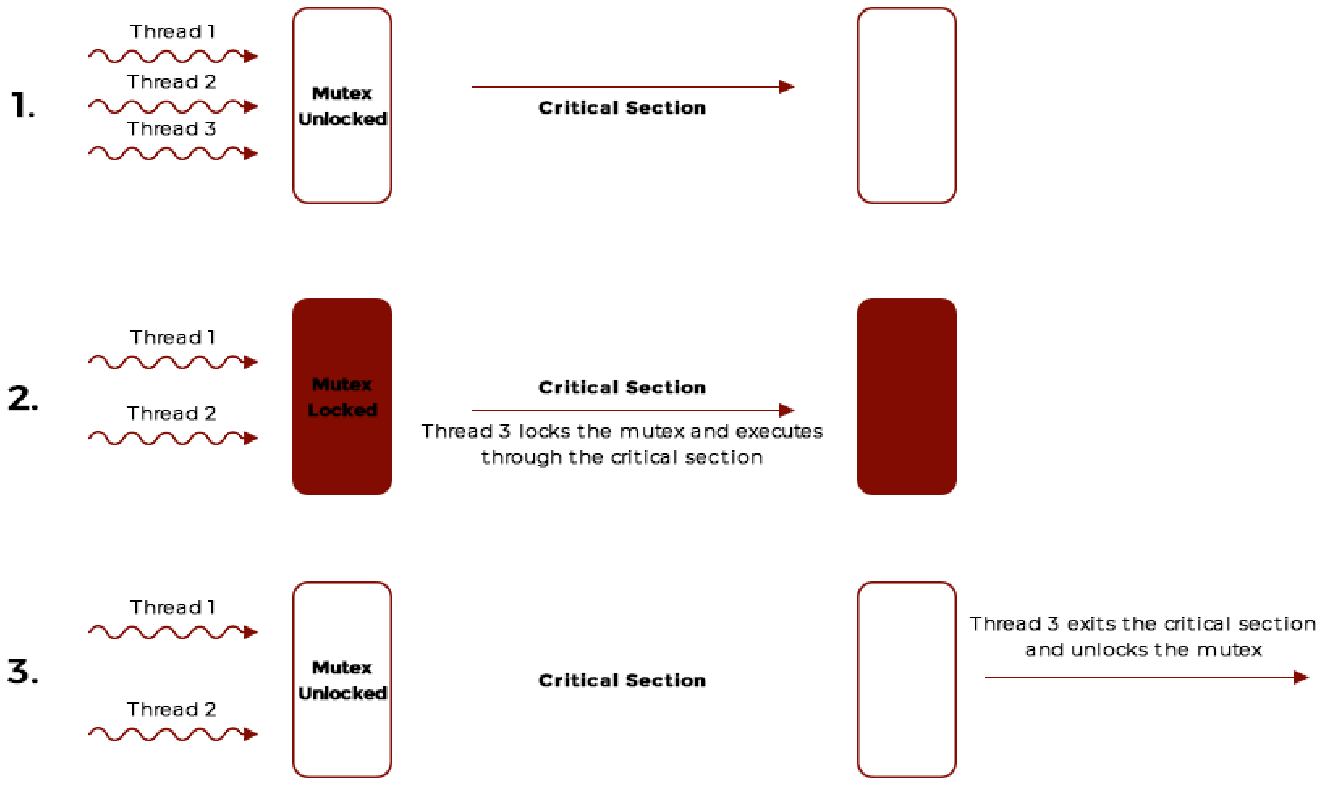
```
If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlock
```

This leads us to the concept of **ownership**. **A mutex is owned by the thread acquiring it till the point the owning-thread releases it, whereas, for a semaphore, there's no notion of ownership.**

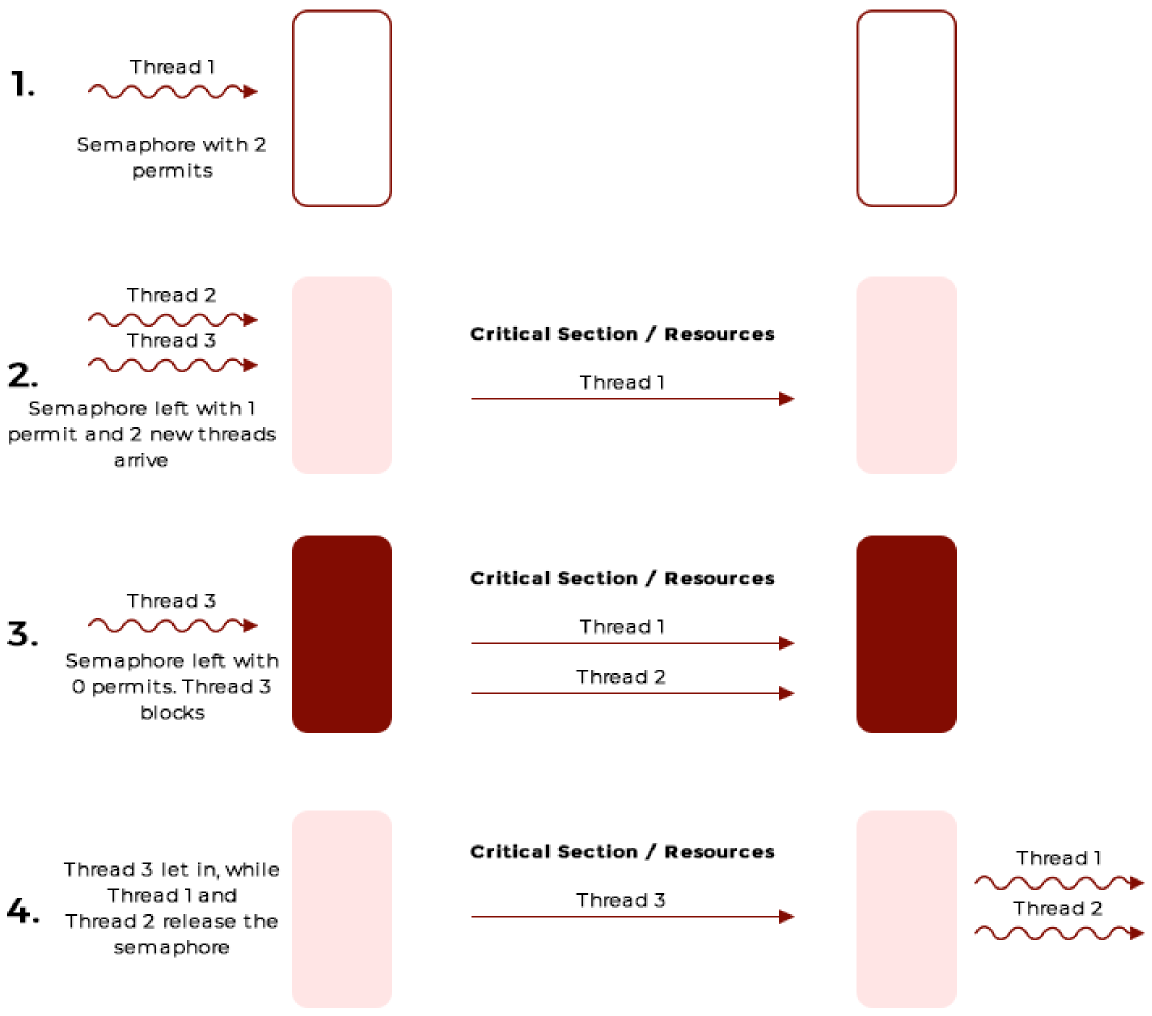
Semaphore for Signaling

Another distinction between a semaphore and a mutex is that semaphores can be used for signaling amongst threads. For example, in the case of the classical **producer/consumer** problem the producer thread can signal the consumer thread by incrementing the semaphore count to indicate to the consumer thread to consume the freshly produced item. A mutex, in contrast, only guards access to shared data among competing threads by forcing threads to serialize their access to critical sections and shared data-structures.

Below is a pictorial representation of how a mutex works.



Below is a depiction of how a semaphore works. The semaphore initially has two permits and allows at most two threads to enter the critical section or access protected resources.


1.
Thread 1
Semaphore with 2 permits
2.
Thread 2
Thread 3
Semaphore left with 1 permit and 2 new threads arrive
Critical Section / Resources
Thread 1
3.
Thread 3
Semaphore left with 0 permits. Thread 3 blocks
Critical Section / Resources
Thread 1
Thread 2
4.
Thread 3 let in, while Thread 1 and Thread 2 release the semaphore
Critical Section / Resources
Thread 3
Thread 1
Thread 2