In a multithreaded application, two or more threads may need to access a shared resource at the same time, resulting in unexpected behavior. Examples of such shared resources are data-structures, input-output devices, files, and network connections.

We call this scenario a *race condition*. And, the part of the program which accesses the shared resource is known as the *critical section*. **So, to avoid a race condition, we need to synchronize access to the critical section.**

## Mutex (or mutual exclusion)

A mutex (or mutual exclusion) is the simplest type of *synchronizer* – it **ensures that only one thread can execute the critical section of a computer program at a time**.

To access a critical section, a thread acquires the mutex, then accesses the critical section, and finally releases the mutex. In the meantime, **all other threads block till the mutex releases.** As soon as a thread exits the critical section, another thread can enter the critical section.

**There are various ways, we can implement a mutex in Java.**

1. **Using synchronized Keyword**
2. **Using ReentrantLock**
3. **Using Semaphore**

### Using synchronized keyword

Using synchronized keyword, which is the simplest way to implement a mutex in Java.

Every object in Java has an intrinsic lock associated with it. **The *synchronized* method and the *synchronized* block use this intrinsic lock** to restrict the access of the critical section to only one thread at a time.

Therefore, when a thread invokes a *synchronized* method or enters a *synchronized* block, **it automatically acquires the lock.** The lock releases when the method or block completes or an exception is thrown from them.

**What is difference between Mutex and Semaphore?**

**Answer: -**

| Mutex | Semaphore |
|---|---|
| Only one thread to access a resource at once. Example, when a client is accessing a file, no one else should have access the same file at the same time. | Restrict the number of threads that can access a resource. Example, limit max 10 connections to access a file simultaneously. |
| Consider a situation of using lockers in the bank. Usually, the rule is that only one person is allowed to enter the locker room. | Consider an ATM cubicle with 4 ATMs, Semaphore can make sure only 4 people can access simultaneously. |

***1. Can a thread acquire more than one lock (Mutex)?***

Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

***2. Can a mutex be locked more than once?***

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a *recursive mutex* can be locked more than once (POSIX compliant systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

***3. What happens if a non-recursive mutex is locked more than once.***

Deadlock. If a thread that had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in a deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of the mutex and return if it is already locked by a same thread to prevent deadlocks.

***4. Are binary semaphore and mutex same?***

No. We suggest treating them separately, as it is explained in signaling vs locking mechanisms. But a binary semaphore may experience the same critical issues (e.g., priority inversion) associated with a mutex. We will cover these in a later article.

A programmer can prefer mutex rather than creating a semaphore with count 1.

### *5. What is a mutex and critical section?*

Some operating systems use the same word *critical section* in the API. Usually, a mutex is a costly operation due to protection protocols associated with it. At last, the objective of mutex is atomic access. There are other ways to achieve atomic access like disabling interrupts which can be much faster but ruins responsiveness. The alternate API makes use of disabling interrupts.

### *6. What are events?*

The semantics of mutex, semaphore, event, critical section, etc… are same. All are synchronization primitives. Based on their cost in using them they are different. We should consult the OS documentation for exact details.

### *7. Can we acquire mutex/semaphore in an Interrupt Service Routine?*

An ISR will run asynchronously in the context of current running thread. It is **not recommended** to query (blocking call) the availability of synchronization primitives in an ISR. The ISR are meant be short, the call to mutex/semaphore may block the current running thread. However, an ISR can signal a semaphore or unlock a mutex.

### *8. What we mean by "thread blocking on mutex/semaphore" when they are not available?*

Every synchronization primitive has a waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of processors to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list gets the resource (more precisely, it depends on the scheduling policies).

### *9. Is it necessary that a thread must block always when resource is not available?*

Not necessary. If the design is sure '*what has to be done when resource is not available '*, the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.

For example, POSIX pthread_mutex_trylock() API. When the mutex is not available the function returns immediately whereas the API pthread_mutex_lock() blocks the thread till the resource is available.