

# Deadlocks, Liveness & Reentrant Locks

We discuss important concurrency concepts such as deadlock, liveness, live-lock, starvation, and reentrant locks in depth. Also included are executable code examples for illustrating these concepts.

## Deadlock and Liveness

Logical follies committed in multithreaded code, while trying to avoid race conditions and guarding critical sections, can lead to a host of subtle and hard to find bugs and side-effects. Some of these incorrect usage patterns have their specific names and are discussed below.

### Deadlock

Deadlocks occur when two or more threads aren't able to make any progress because the resource required by the first thread is held by the second and the resource required by the second thread is held by the first.

### Liveness

The ability of a program or an application to execute in a timely manner is called liveness. If a program experiences a deadlock, then it's not exhibiting liveness.

### Live-Lock

A live-lock occurs when two threads continuously react in response to the actions by the other thread without making any real progress. The best analogy is to think of two persons trying to cross each other in a hallway. John moves to the left to let Arun pass, and Arun moves to his right to let John pass. Both block each other now. John sees he's blocking Arun again and moves to his right, and Arun moves to his left, seeing he's blocking John. They never cross each other and keep blocking each other. This scenario is an example of a live-lock. A process seems to be running and

not deadlocked but in reality, isn't making any progress.

### Starvation

Other than a deadlock, an application thread can also experience starvation when it never gets CPU time or access to shared resources. Other **greedy** threads continuously hog shared system resources, not letting the starving thread make any progress.

### Deadlock Example

Consider the pseudocode below:

```
void increment() {  
  
    acquire MUTEX_A  
    acquire MUTEX_B  
    // do work here  
    release MUTEX_B  
    release MUTEX_A  
  
}  
  
void decrement() {  
  
    acquire MUTEX_B  
    acquire MUTEX_A  
    // do work here  
    release MUTEX_A  
    release MUTEX_B  
  
}
```

The above code can potentially result in a deadlock. Note that deadlock may not always happen, but for certain execution sequences, deadlock can occur. Consider the execution sequence below that ends up in a deadlock:

```
T1 enters function increment  
  
T1 acquires MUTEX_A  
  
T1 gets context switched by the operating system  
  
T2 enters function decrement
```

```
T2 acquires MUTEX_B
```

```
both threads are blocked now
```

Thread **T2** can't make progress as it requires **MUTEX\_A** which is being held by **T1**. Now when **T1** wakes up, it can't make progress as it requires **MUTEX\_B** and that is being held up by **T2**. This is a classic text-book example of a deadlock.

If you run the code snippet below, you'll see that the statements for acquiring mutexes: **lockA** and **lockB** print out but there's no progress after that and the Ruby interpreter barfs about the program entering into a deadlock. In this scenario, the deadlock occurs because the mutexes are being acquired in a nested fashion.

```
mutexA = Mutex.new
mutexB = Mutex.new

t1 = Thread.new do
  puts "Acquiring Mutex A"
  mutexA.lock()
  sleep(1)
  mutexB.lock()
  puts "Thread t1 exiting"
end

t2 = Thread.new do
  puts "Acquiring Mutex B"
  mutexB.lock()
  sleep(1)
  mutexA.lock()
  puts "Thread t2 exiting"
end

t1.join()
t2.join()
```



## Reentrant Lock

Reentrant locks allow for re-locking or re-entering of a synchronization lock. If a synchronization primitive doesn't allow reacquisition of itself by

a thread that has already acquired it, then such a thread would block as soon as it attempts to reacquire the primitive a second time.

This concept is best explained with an example. Consider the snippet below that creates a **Mutex** object and attempts to acquire it twice. Since the lock isn't reentrant the Ruby interpreter barfs warning about recursively locking the mutex.

```
mutex = Mutex.new

mutex.lock()
mutex.lock()
puts "Hello"
mutex.unlock()
mutex.unlock()
```



On the contrary a **Monitor** object can be recursively entered as shown below:

```
monitor = Monitor.new
monitor.enter()
monitor.enter()
puts "I am reentrant"
monitor.exit()
monitor.exit()
```



We'll discuss both the synchronization constructs in detail in lessons ahead.