**Thread Safety**

Thread safety is a concept applicable in the context of multi-threaded programs. Multiple thread can access to the same address space at the same time. So, **they can write to the exact same memory location at the same time**. It is a defining property of threads. So, **this property of thread is not good for the functionality**.

**So, Thread safety is a technique which manipulates shared data structure in a manner that guarantees the safe execution of a piece of code by the multiple threads at the same time**. A code is called thread safe if it is being called from multiple threads concurrently without the breaking of functionalities.

Thread safety **removes** the following conditions in the code:

1. **Race Condition**
2. **Deadlocks**

### **Race Condition**

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:

if (x == 5) // The "Check"

{

y = x \* 2; // The "Act"

// If another thread changed x in between "if (x == 5)" and "y = x \* 2" above,

// y will not be equal to 10.

}

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

// Obtain lock for x

if (x == 5)

{

y = x \* 2; // Now, nothing can change x until the lock is released.

// Therefore y = 10

}

// release lock for x

**Symptoms of Race Condition**

The most common symptom of a race condition is unexpected values of variable that are shared between multiple threads. In this case, sometimes one thread wins, and sometimes the other thread wins. At the other times, the result of execution may be correct. Also, if each thread executes separately, the result of variable comes expected.

**Avoid Race Condition:**

Best approach to handle race condition is to write code very carefully and proactively. Always try to use locks, mutexes, ManualResetEvents, etc to make code thread safe in the case of multi-threaded environment.

### **Deadlock**

Deadlock case happens in concurrent or multi-threaded environment. It is kind of a situation in which two or more competing threads or tasks wait for the other task to finish and they never finish.

It is a vary famous problem in multi processing systems, parallel computing and distributed systems. This case basically happens with the shared resource. This resource locks in this case. Deadlock has very vast explanation.

**Conditions of Deadlock**

A condition is called Deadlock if the following condition matches:

1. Mutual Exclusion: Only process can use the resource at the given time.
2. Hold and Wait
3. No Preemption
4. Circular Wait

These four conditions are know as the **Coffman Conditions. Edward G. Coffman, Jr.**described this in his article published in 1971.

### **Ways to Make an Object Thread-Safe**

The following are the three approaches to make an object thread-safe:

1. Synchronize the critical section.
2. Make it immutable
3. Use a thread-safe wrapper

Static constructors are guaranteed to be run only once per application domain, before any instances of a class are created or any static members are accessed.

Using a Static Constructor actually is Thread-Safe.

**Question:**Is C# Static Method Thread-Safe?

Static methods are not inherently thread-safe. CLR of C# doesn't thread different than instance method.

**Question:**Is C# Delegate Thread-Safe?

Modifying event is not thread-safe, but invoking a Delegate is thread-safe. Since a **Delegate is immutable** type so it is thread safe.

**Soru**: What does the other thread do when it encounters the lock? Does it wait? Error?

**Cevap**: Yes, the other thread will **have to wait** **until the lock is released before it can proceed**. This makes it very important that the lock is released by the holding thread when it is finished with it. **If it never releases it, then the other thread will wait indefinitely**.

-

-

-

**Thread safety** is a [computer programming](https://www.wikiwand.com/en/Computer_programming) concept applicable to [multi-threaded](https://www.wikiwand.com/en/Thread_(computing)) code.

Thread-safe code only manipulates shared data structures in a manner that ensures that **all threads behave properly** and **fulfill their design specifications** **without unintended interaction**. There are various strategies for making thread-safe data structures.[[1]](https://www.wikiwand.com/en/Thread_safety#citenote1)[[2]](https://www.wikiwand.com/en/Thread_safety#citenote2)

Levels of thread safety

Software libraries can provide certain thread-safety guarantees. For example, concurrent reads might be guaranteed to be thread-safe, but concurrent writes might not be. Whether a program using such a library is thread-safe depends on whether it uses the library in a manner consistent with those guarantees.

Different vendors use slightly different terminology for thread-safety:[3][4][5][6]

**Thread safe**: Implementation is guaranteed to be free of race conditions when accessed by multiple threads simultaneously.

**Conditionally safe**: Different threads can access different objects simultaneously, and access to shared data is protected from race conditions.

**Not thread safe**: Data structures should not be accessed simultaneously by different threads.

Thread safety guarantees usually also include design steps to prevent or limit the risk of different forms of deadlocks, as well as optimizations to maximize concurrent performance. However, deadlock-free guarantees cannot always be given, since deadlocks can be caused by callbacks and violation of architectural layering independent of the library itself.

-

-