

* Farklı Threadlerin Ortak Bir Kaynağı Kullanması:

Unsynchronized data access

When two threads running in parallel read and write the same data, it is open which statement comes first.

Half-written data: When one thread reads data, which another thread modifies, the reading thread might even read the data in the middle of the write of the other thread, thus reading neither the old nor the new value.

Reordered statements: Statements and operations might be reordered so that the behavior of each single thread is correct, but in combination of all threads, expected behavior is broken.

→ CPU, cache ve compiler optimizasyonundan kaynaklı

* Mutex:
- lock / require aynı anlamı getirir!
- unlock / release aynı

* Kritik kod alanını korumak için kullanılır. (Critical Section)

```
std::mutex m1;  
std::timed_mutex m2;  
std::recursive_mutex m3;  
std::recursive_timed_mutex m4;
```

```
int main()  
{  
}  
}
```

→ Bu mutexlerin lock / unlock kontrolü C++11 idiomü ile sağlanabilir.

mutex sarmalayan RAII sınıfları

```
lock_guard  
unique_lock  
scoped_lock → C++ 17  
shared_lock
```

→ Lock / try_lock:

```
using namespace std;  
while (!m1.try_lock()) {  
    //baska is yap  
}
```

→ lock ile yok thread'i critical section'a sokar ya da bloke eder!

* Timed mutex'e özel

→ try lock for → works with duration

→ try lock until → works until timepoint

```
unsigned long long counter = 0;
std::mutex mtx;
```

```
void func()
```

```
{
    mtx.lock();
    for (unsigned long long i = 0; i < 1'000'000ull; ++i) {
        ++counter;
    }
    mtx.unlock();
}
```

→ mutexle korunmedigi sonucunda counter degeri 1 000 000 u geciyor.

```
int main()
```

```
{
    std::thread t1(func);
    std::thread t2(func);
    std::thread t3(func);
    std::thread t4(func);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    std::cout << counter << '\n';
}
```

+ Lock Guard:

```
void func()
```

```
{
    std::lock_guard<std::mutex> lk(mtx);
    for (unsigned long long i = 0; i < 1'000'000ull; ++i) {
        ++counter;
    }
}
```

→ RAII idiom ile mutex i saklamak
⇒ Nesne hayata gelince lock, ayrilay edince unlock!

→ lock guard kopyalanmaz, anlık kullanilir.

→ lock guard, interfaç'inde yalnizca destruktor bulundur.

→ lock guard, edinilmis bir mutex'i de adopt edip RAII idiom'u ile kullanmamizi saglar

```
void func()
```

```
{
    mtx.lock();
    std::lock_guard<std::mutex>(mtx, std::adopt_lock);
    for (unsigned long long i = 0; i < 1'000'000ull; ++i) {
        ++counter;
    }
}
```



```
std::mutex mtx;
```

```
void print_block(int n, char c)
```

```
{
```

```
using namespace std::literals;
```

```
std::lock_guard guard{ mtx };
```

```
for (int i = 0; i < n; ++i) {
```

```
std::this_thread::sleep_for(5ms);
```

```
std::cout << c;
```

```
}
```

```
std::cout << '\n';
```

```
}
```

```
int main()
```

```
{
```

```
std::thread th1(print_block, 50, '*');
```

```
std::thread th2(print_block, 50, '$');
```

```
std::thread th3(print_block, 50, '+');
```

```
std::thread th4(print_block, 50, '!');
```

```
th1.join();
```

```
th2.join();
```

```
th3.join();
```

```
th4.join();
```

```
}
```

CTAD ile argümanı buldu!

mutex olmadan n kere yama garantisi yok

```
#include <mutex>
#include <iostream>
#include <exception>
```

```
std::mutex mtx;
```

```
int main()
```

```
{
```

```
try {
```

```
mtx.lock();
```

```
mtx.lock();
```

```
//...
```

```
}
```

```
catch (const std::exception& ex) {
```

```
std::cout << "exception caught: " << ex.what() << '\n';
```

```
//exception caught: device or resource busy: device or resource busy
```

```
}
```

Garanti olmamasıyla birlikte:

2 kez lock acquisition exception gönderir.

→ Bu için recursive mutex kullanılıyor

```
#include <mutex>
#include <iostream>
#include <thread>

std::recursive_mutex rmtx;
int gcount = 0;
```

```
void rfunc(char c, int n)
```

```
{
```

```
if (n < 0)
    return;
```

```
rmtx.lock();
```

```
std::cout << c << ' ' << gcount++ << '\n';
```

```
rfunc(c, n - 1);
```

```
rmtx.unlock();
```

```
}
```

```
int main()
```

```
{
```

```
std::thread tx{ rfunc, 'x', 10 };
```

```
std::thread ty{ rfunc, 'y', 10 };
```

```
tx.join();
```

```
ty.join();
```

```
}
```

→ recursive mutex.

* Mülakat Sorusu:

```
int x{};

std::mutex mtx_func;
std::mutex mtx_foo;

void func()
{
    std::lock_guard guard{ mtx_func };
    for (int i = 0; i < 1000; ++i) {
        ++x;
    }
}

void foo()
{
    std::lock_guard guard{ mtx_foo };
    for (int i = 0; i < 1000; ++i) {
        ++x;
    }
}

int main()
{
    std::thread t1{ func };
    std::thread t2{ foo };

    t1.join();
    t2.join();
}
```

→ Burada x'in 2000 olma garantisi yok!!

Çünkü farklı mutexler lock guard ediyor!!

* Eğer bir işlem aynı mutex ile korunmıyorsa!

* Unique Lock:

```
//std::adopt_lock
//std::try_to_lock
//std::defer_lock

std::mutex mtx;

void func()
{
    mtx.lock();
    //std::unique_lock<std::mutex> ulock(mtx);
    //std::unique_lock<std::mutex> ulock(mtx, std::try_to_lock);
    //std::unique_lock<std::mutex> ulock(mtx, std::defer_lock);
    std::unique_lock<std::mutex> ulock(mtx, std::try_to_lock);
}
```

→ Othub'da güzel anlatmış!

Aşağıdaki kodda deadlock oluşuyor.

Her iki thread'de bloke oluyor.

foo'yu yürüten threadin devam edebilmesi için bar'ı yürüten thread'in b_mtx'i serbest bırakması gerekiyor.

bar'ı yürüten threadin devam edebilmesi için foo'yu yürüten thread'in a_mtx'i serbest bırakması gerekiyor.

Eğer mutex'ler her iki thread tarafından da aynı sırada edinilseydi bir sorun oluşmayacaktı.

*/

```
#include<mutex>
#include <iostream>
```

```
std::mutex a_mtx;
std::mutex b_mtx;
```

```
void foo()
```

```
{
    using namespace std::literals;

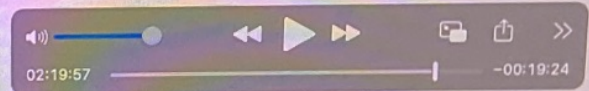
    a_mtx.lock();
    std::this_thread::sleep_for(100ms);
    b_mtx.lock();
    std::cout << "func()" << std::endl;
    a_mtx.unlock();
    b_mtx.unlock();
}
```

```
void bar()
```

```
{
    using namespace std::literals;
    b_mtx.lock();
    std::this_thread::sleep_for(100ms);
    a_mtx.lock();
    std::cout << "bar()" << std::endl;
    b_mtx.unlock();
    a_mtx.unlock();
}
```

```
int main()
```

```
{
    std::thread t1{ foo };
    std::thread t2{ bar };
    t1.join();
    t2.join();
}
```



→ threadler her iki lock'ı aynı anda alamazlar. → a_mtx lock edilemez. → b_mtx lock edilemez. → Deadlock oluşma ihtimali çok yüksek!
Bu yüzden: std::lock() global kilitlenmeyi kullanır. Deadlock olmama garantisi verir!

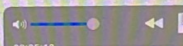
* Std::lock:

```
void foo()
```

```
{
    using namespace std::literals;
    std::lock(a_mtx, b_mtx);
    std::this_thread::sleep_for(100ms);
    std::cout << "foo()" << std::endl;
    a_mtx.unlock();
    b_mtx.unlock();
}
```

```
void bar()
```

```
{
    using namespace std::literals;
    std::lock(b_mtx, a_mtx);
    std::this_thread::sleep_for(100ms);
    std::cout << "bar()" << std::endl;
    a_mtx.unlock();
}
```



* Scoped_lock: - Cpp 17 ile gelen bir class template

```
#include <mutex>
#include <iostream>

std::mutex a_mtx;
std::mutex b_mtx;

void foo()
{
    using namespace std::literals;

    std::scoped_lock<std::mutex, std::mutex> mylock{ a_mtx, b_mtx };

    std::this_thread::sleep_for(100ms);
    std::cout << "foo()" << std::endl;
}

void bar()
{
    using namespace std::literals;

    std::scoped_lock<std::mutex, std::mutex> mylock{ a_mtx, b_mtx };

    std::this_thread::sleep_for(100ms);
    std::cout << "bar()" << std::endl;
}

int main()
{
    std::thread t1{ foo };
    std::thread t2{ bar };
    t1.join();
    t2.join();
}
```



02:26:34