* 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 anlama gelir!
- 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 RAII idiomu ile yapılabilir.

```
mutex sarmalayan RAII sınıfları

lock_guard
unique_lock
scoped_lock → cpp 17
shared_lock
```

→ lock / try_lock:

```
using namespace std;

while (!m1.try_lock()) {
    ///baska is yap
}
```

→ bool değer döndürür.

→ lock ise yan thread'i critical section'a sokar yoksa bloke eder!

* Timed mutex'e özel
- try lock for → works with duration
- try lock until → works until timepoint

```cpp
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();
}

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';
}
```

→ mutex'le korunmadığı senaryolarda counter değeri 1 000 000 olmuyor.

+ Lock Guard:

```cpp
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 sarmalıyor.
⇒ Nesne hayatı bittiğinde lock, destroy edince unlock!

→ lock guard kopyalanamaz, ancak taşınabilir.

→ lock guard, interface'inde yalnızca destructor bulunur.

→ lock guard, edinilmiş bir mutex'i de adopt edip RAII idiom'u ile kullanmamızı sağlar

```cpp
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;
    }
}
```

```cpp
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 oluşturuldu.

mutex almadan önce yazma garantisi yok

```cpp
#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 olmamakla birlikte:

2 kez lock aquisition exception gönderir.

Bunun için recursive mutex kullanılabilir.

```cpp
#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:

```cpp
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 muteksler lock_guard ediliyor!!

* Garanti olması için, aynı mutex ile korunmalı!

*Unique Lock:

```cpp
//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);
```

→ Github'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();
```

02:19:57 -00:19:24

→ Çünkü lock edilmemelidir.

→ threadler her bir fonksiyonda da lock edildiği için, Deadlock olusma ihtimali Çok yüksek!

Bu yüzden: std::lock() global fonksiyonu kullanırız. 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();
```

ya ikisini de kilitler, ya da bloke olur

02:25:12

* Scoped_lock: - C++ 17 ile gelen bir class template

```cpp
#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