2024 01 10

```cpp
#include <shared_mutex>

using namespace std::literals;

int cnt = 0;
std::shared_mutex mtx;

void writer()
{
for (int i = 0; i < 10; ++i) {
    {
        std::scoped_lock lock(mtx);
        ++cnt;
    }
    std::this_thread::sleep_for(100ms);
}
}

void reader()
{
for (int i = 0; i < 100; ++i) {
    int c;
    {
        // yazma amaçlı thread kilidi edinemez ama oku amaçlı edinebilir
        std::shared_lock lock(mtx);
        c = cnt;
    }
    std::osyncstream{ std::cout } << std::this_thread::get_id() << ' ' << c <<
'\n';
    std::this_thread::sleep_for(10ms);
}
}

int main()
{
std::vector<std::jthread> tvec;

tvec.reserve(16);
tvec.emplace_back(writer);

for (int i = 0; i < 16; ++i)
    tvec.emplace_back(reader);

}
```

```cpp
// std::unique_lock
std::mutex mtx;

int main()
{
    unique_lock<mutex> ulock;

    // diğer lock sınıflarından farklı olarak aşağıdakilere sahip:
    ulock.lock();
    ulock.unlock();
    ulock.try_lock();

    // unique_lock move only type
}
```

```cpp
using namespace std;
std::mutex mtx;

void foo()
{
    unique_lock lock{ mtx }; // kilidi ediniyoruz lock() çağrılır
}

void bar()
{
    // lock() çağrılmaz
    unique_lock lock{ mtx, adopt_lock}; // kilidi edinmiş durumda alıyoruz
}

void bar()
{
    // lock() çağrılmaz
    unique_lock lock{ mtx, defer_lock}; // kilidi kitlemiyor.
    // burada kilitlemek gerekiyor.
}

void bam()
{
    // kilidi edinmeye çalışıyor edinemezse bloke edilmiyor
    unique_lock lock{ mtx, try_to_lock};

    if (lock.owns_lock())
    {
        // kilidi edinmişse bu bölgeye girer.
        lock.unlock();
    }
}
```

```cpp
int cnt{};
std::mutex mtx;

void func()
{
    for (int i = 0; i < 1'000'000; ++i)
    {
        std::unique_lock ulock{ mtx, std::defer_lock };
        ulock.lock();
        ++cnt;
        ulock.unlock();

        ulock.lock();
        ++cnt;
        ulock.unlock();
    }
}

int main()
{
    {
        std::jthread t1{ func };
        std::jthread t2{ func };
        std::jthread t3{ func };
        std::jthread t4{ func };
        std::jthread t5{ func };
    }

    std::cout << cnt << "\n";
}
```

```cpp
// std::once_flag and std::call_once

using namespace std;

unique_ptr<string> uptr;
once_flag flag;

void initialize()
{
    osyncstream{ cout } << "initialize " << <this_thread::get_id() << "\n";
    uptr = make_unique<string>("emre bahtiyar");
}

const string& get_value()
{
    // herhangi bir thread sadece bir kez initialize fonksiyonu çağıracak
    call_once(flag, initialize);
    return *uptr;
}

void workload()
{
    const std::string& rs = get_value();
    osyncstream{ cout } << &rs << "\n";
}

 int main()
 {
     vector<thread> tvec;
     tvec.reverse(20);
     for (int i = 0; i < 16; ++i)
     {
         tvec.emplace_back(workload)
     }
 }
```

```cpp
// thread-safe singleton
class Singleton {
public:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    static Singleton* get_instance()
    {
        call_once(m_init_flag, Singleton::init);

        return m_instance;
    }

    static void init()
    {
        m_instance = new Singleton();
    }
private:
    static std::once_flag m_init_flag;
    static Singleton* m_instance;
    Singleton() = default;
};

Singleton* Singleton::m_instance{};
std::once_flag Singleton::m_init_flag;


void func()
{
    std::osyncstream{ std::cout } << Singleton::get_instance() << '\n';
}

int main()
{
    std::vector<std::thread> tvec;
    for (int i = 0; i < 100; ++i) {
        tvec.emplace_back(func);
    }

    for (auto& th : tvec)
        th.join();
}
```

```cpp
// thread-safe singleton
class Singleton {
    public:
        static Singleton& get_instance()
        {
            static Singleton s;
            return s;
        }
}

// call_once alternatif
using namespace std;
void func()
{
    osyncstream{cout} << "func cagrildi " << this_thread::get_id() << "\n";
}

void foo()
{
    std::this_thread::sleep_for(50ms);
    static auto f = [] {func(); return 0;}();
}

int main()
{
    vector<jthread> tvec;

    for(int i = 0; i< 10; ++i)
        tvec.emplace_back(foo);
}
```

std::future and std::promise

```cpp
#include <future>
int main()
{
    using namespace std;

    std::promise<int> prom;
    future<int> ft = prom.get_future();

    prom.set_value(12);

    int val = ft.get(); // val 12
}
```

```cpp
void produce(std::promise<double> prm, double val)
{
    prm.set_value(dval * dval);
}

int main()
{
    using namespace std;

    promise<double> prom;
    auto ft = prom.get_future();

    thread t{produce, move(prom), 4.543};

    auto val = ft.get();

    cout << "value = " << val << "\n";

    t.join();
}
```

```cpp
std::string foo(std::string str)
{
    auto temp = str;
    reverse(str.begin(), std.end());

    return temp + str;
}

void bar(std::promise<std::string>&& prom, std::string str)
{
    prom.set_value(foo(str));
}


using namespace std;

int main()
{
    promise<string> prom;

    future<string> ft = prom.get_future();

    thread t{bar, move(prom), "tamer" };

    cout << ft.get() << "\n";

    t.join();
}
```

```cpp
struct Div
{
    void operator()(std::promise<int>&& prom, int a, int b)
    {
        if (b == 0)
        {
            auto str = "divide by zero error " + std::to_string(a) + "\\"  +
std::to_string(b);
            prom.set_exception(std::make_exception_ptr(std::runtime_error(str));
        }
        else
        {
            prom.set_value(a / b);
        }
    }
};

using namespace std;

int main()
{
    promise<int> prom;

    auto ft = prom.get_future();

    thread th{ Div{}, move(prom), 12, 3};

    try
    {
        cout << ft.get() << "\n";
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caugth : " << ex.what() << "\n";
    }


    th.join();
}
```

Std::async

```
// std::async
int foo(int x, int y)
{
    return x * y + 5;
}
void bar(int x, int y)
{
    return x + y + 2;
}
int main()
{
    using namespace std;

    auto ft = async(foo, 10, 30); // std::future<int> döner
    auto ft1 = async(bar, 10, 30); // std::future<int> döner

    auto val = ft.get() + ft1.get();

}
```

```
// std::packaged_task
int foo(int, int);
int main()
{
    using namespace std;

    packaged_task mytask(foo, 2, 5);

    packaged_task <int(int, int)> task{ foo };

    thread th{ task, 3, 6};
}
```