→ Dersin başında kripto piyasası hakkında konuştu :)

* once flag / call once:

```cpp
std::once_flag gflag;

void func(int id)
{
    std::call_once(gflag, [id]() {
        std::cout << id << " threadi icin cagrilddi\n";
        });

    ///...

}

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

    for (int i = 0; i < 10; ++i) {
        tvec.emplace_back(func, i);
    }


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

→ Birden fazla thread olmasına rağmen, callable yalnızca tek bir thread için çağrılır.

→ Fakat hangi thread çağırır, o deterministik değil!

```cpp
class Singleton {

public:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    static Singleton* get_instance()
    {
        std::call_once(m_init_flag, init);
        return m_instance;
    }

    static void init()
    {
        m_instance = new Singleton;
    }

private:
    inline static std::once_flag m_init_flag;
    inline static Singleton* m_instance{};
    Singleton() = default;
};
```

* Condition Variable:

- bir thread, başka bir thread'in hazırladığı veriyi kullanmasını istiyoruz. Buradaki problem, producer – consumer problem – torn read / torn write olma durumundan kaçınmamız gerek

* Concurrency in Action kitabını oku!

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

void producer()
{
    using namespace std::literals;
    std::this_thread::sleep_for(1000ms);
    std::lock_guard lk{ mtx };
    //production
    shared_variable = 999;
}

void consumer()
{
    std::unique_lock ulock{ mtx };

    while (shared_variable == 0) {
        ulock.unlock();
        ulock.lock();
    }
```

→ unique lock olmasının nedeni bir den fazla kez lock-unlock yapılıyor!

```cpp
while (shared_variable == 0) {
    ulock.unlock();
    std::this_thread::yield();
    std::this_thread::sleep_for(1000ms);
    ulock.lock();
```

Aynı şekilde thread yield edilip uyutulur.

→ Böylelikle işlemci zamanı aşırılmamış olur.

→ Fakat burada da sleep zamanında senkron kaçırabiliriz

→ En sağlıklısı bu yüzden condition variable. Task tamamlanınca bir uyandıralım!

→ Necati'nin sitesinden

condition variable

- *condition variable* bir başka *thread*'in tetikleyeceği bir *event*'i beklemenin temel mekanizmasıdır.
- Birden fazla *thread*'in kullanıldığı programlarda çoğu zaman bir *thread*'in belirli bir *event* oluşuncaya kadar başka bir *thread*'i beklemesi gerekir (giriş çıkış işleminin tamamlanması ya da bir verinin hazırlanması gibi)
- Bir *thread*'in bir *event*'in gerçekleşip gerçekleşmediğini sürekli olarak sorgulaması, işlemci zamanının boşa harcanmasına, dolayısıyla verimin düşmesine neden olabilir.
- Böyle durumlarda bekleyen *thread*'in bloke olması ve ilgili *event* gerçekleştikten sonra tekrar çalışmaya başlaması genellikle daha iyidir.
- Bir *condition variable*, belirli bir koşul *(condition)* gerçekleşinceye kadar bir *thread*'in bloke olarak beklemesini sağlayan bir senkronizasyon yapısıdır.

- Bir *condition variable* bir *event* ile ilişkilendirilir.
- Bir *event*'i beklemek isteyen bir *thread condition variable*'ın *wait* fonksiyonlarından birini çağırır. *(wait, wait_for, wait_until)*
- Bir *thread (producer)* bir *event*'in gerçekleşmiş olduğunu bir ya da birden fazla *thread*'e *(consumers)* bildirmek için *condition variable*'ın *notify_one* ya da *notify_all* fonksiyonlarından birini çağırır.
- Bildirimi alan *thread* çalışmaya devam ettiğinde istenen koşulun sağlanmış olma garantisi yoktur. Başka bir *thread* koşulun değişmesini sağlamış olabilir ya da *"spurious wakeup"* denilen durum oluşmuş olabilir. (Bekleyen bir *thread*'in aslında diğer taraftan bir bildirim almadan uyanmasına "spurious wake" denir.)

Bu nedenle uyanan *thread*'in koşulun sağlanmış olup olmadığını tekrar kontrol etmesi gerekir.

- *condition_variable* olarak *std::condition_variable* sınıfı türünden bir nesne kullanılır.
- *condition_variable* sınıfı <*condition_variable*> başlık dosyasında tanımlanmıştır. *std::condition_variable* nesneleri kopyalanamaz ve taşınamaz *(not copyable - not moveable)*.
- sınıfın *wait, wait_for* ya da *wait_until* üye fonksiyonları ile bekleyecek *thread* bloke edilir koşulun sağlanması beklenir.
- sınıfın *notify_one* ve *notify_all* üye fonksiyonları ile bekleyen *thread*'lere koşulun oluştuğu bildirilir *(signal)*.
- uyanan *thread*'in koşulu tekrar sınaması gerekir, çünkü
  - *spurious wakeup* oluşabilir.
  - sinyalin alınması ve *mutex*'in edinilmesi zaman aralığı içinde başka bir *thread* koşulu değiştirmiş olabilir.
- *wait, wait_for* ve *wait_until* fonksiyonları *mutex*'i atomik olarak edinirler ve ilgili *thread*'i bloke ederler.
- *notify_one* ve *notify_all* fonksiyonları atomiktir.

Tipik işlem akışı şöyle gerçekleştirilir:

- Tipik olarak *std::lock_guard* kullanarak bir *mutex*'i edinir.
- Kilit edinilmiş durumdayken paylaşılan değişkeni değiştirir. Yapılan değişikliğin bekleyen *thread*(ler)e doğru bir şekilde bildirilebilmesi için, paylaşılan değişken atomik olsa dahi değişikliğin kilit edinilmiş durumda yapılması gerekir.
- Bu amaçla tanımlanmış olan *std::condition_variable* nesnesinin *notify_one* ya da *notify_all* fonksiyonlarından birini çağırır. Bu fonksiyonlar çağrıldığında kilitin edinilmiş durumda olması gerekmez. Eğer bu fonksiyonlar kilit edinilmiş durumda çağrılırsa bildirim alan *thread*'ler kilidi edinemezler ve tekrar bloke olurlar.

Bekleyen bir *thread*, önce *std::unique_lock* kullanarak (aynı) *mutex*'i edinir. Daha sonra aşağıdaki iki seçenekten birini uygular:

- Birinci seçenek
  - Değişikliği zaten yapılmış ve bildirimin de gerçekleşmiş olabileceği ihtimaline karşı önce kuşulu test eder.
  - wait, wait_for, ya da wait_until fonksiyonlarından birini çağırır. Çağrılan *wait* fonksiyonu edinilmiş *mutex*'i otomatik olarak serbest bırakır ve *thread*'in çalışmasını durdurur.
  - *condition_variable* nesnesinin *notify* fonksiyonu çağrıldığında (ya da bekleme süresi dolduğunda) ya da bir *"spurious wakeup"* oluştuğunda, *thread* uyanır ve *mutex* yeniden edinilir.
- Uyanan ve kilidi edinen *thread*'in koşulun gerçekleşip gerçekleşmediğini kontrol etmesi ve eğer bir *spurious wakeup* söz konusu ise tekrar bekleme durumuna geçmesi gerekir.
- İkinci seçenek olarak bekleyen *thread*
  - bu işlemlerin hepsini sarmalayan *wait* fonksiyonlarının bir *predicate* alan *overload*'larından birini çağırır.

std::condition_variable sınıfı yalnızca *std::unique_lockstd::mutex* ile kullanılabilir. Bu şekilde kullanım zorunluluğu bazı platformlarda en yüksek verimle çalışmasını sağlar. *std::condition_variable_any* sınıfı ise *BasicLockable* niteliğini sağlayan herhangi bir nesneyle *(örneğin std::shared_lock)* çalışabilmesini sağlar.
std::condition_variable sınıfının *wait, wait_for, wait_until, notify_one ve notify_all* üye fonksiyonları birden fazla *thread* tarafından eş zamanlı çağrılabilir.

std::condition_variable sınıfının *wait* üye fonksiyonu, bloke olmadan beklemeye *(busy wait)* karşı bir optimizasyon olarak görülebilir. *wait* fonksiyonunu (ideal olmasa da) gerçekleştirimi şöyle olabilir:

```cpp
bool ready_flag{};
std::mutex mtx;
std::condition_variable cv;


void producer()
{
    std::cout << "producer is producing the data\n";
    {
        std::lock_guard lock{ mtx };
        data = 78754;
        ready_flag = true;
    }
    cv.notify_one();
}


void consumer()
{
    {
        std::unique_lock ulock{ mtx };
        cv.wait(ulock, [] {return ready_flag; });
    }
}
```

spurious wake up olmadığı sürece consumer yukarıda

eğer consumer spurious wake up durumuna düşerse, bu lambda ifadesini çağırıp, mutex'i kilitli geri yolluyor.

```cpp
template<typename Pred>
void wait(std::unique_lock<std::mutex>& lk, Predicate pred)
{
    while(!pred()) {
        lk.unlock();
        lk.lock();
    }
}
```

→ wait implementasyonu

*Thread safe Stack Example:

```cpp
#include <thread>
#include <iostream>

class IStack {
public:
    IStack() {};
    IStack(const IStack&) = delete;
    IStack& operator=(const IStack&) = delete;
    int pop()
    {
        std::unique_lock lock(m_);
        m_cv.wait(lock, [this]() {return !m_vec.empty(); });
        int val = m_vec.back();
        m_vec.pop_back();
        return val;
    }

    void push(int x)
    {
        std::scoped_lock lock(m_);
        m_vec.push_back(x);
        m_cv.notify_one();
    }
private:
    std::vector<int> m_vec;
    mutable std::mutex m_;
    mutable std::condition_variable m_cv;
};
```

```cpp
constexpr int n = 1000;
IStack s;

void producer()
{
    for (int i = 0; i < n; ++i)
        s.push(2 * i + 1);
}

void consumer()
{
    for (int i = 0; i < n; ++i)
        std::cout << s.pop() << '\n';
}

int main()
{
    std::thread th1(producer);
    std::thread th2(consumer);

    th1.join();
    th2.join();
}
```