# lessons 64 - 67

## DERS 64

**Concurrency:** belirli işlemlerin belirli bir zaman dilimi içinde beraber yapılması, ama yönteme vurgu yok. nasıl yapıosan yap, eğer yapabiliosan concurrency, bu zaman dilimi belki 1saniye belki belki 1ms, super loopdaki tasklar, kuyruk ve state machine mantığı. veya gerçekten paralel çalışmada olur tabi birden fazla core varsa veya RTOS ise.

**Parallelism:** Bu işlerin gerçekten aynı zaman noktasında yapılması demek. PC de birden fazla Core yada RTOS ile task switch gerek.  **Parallelism** , **Concurrency** nin içindedir.

**process**: daha fazla zaman ihtiyac duyar, maliyeti yüksek. İzoledir. Veriler paylaşılmaz. OS arkada yönetir. Çalışma durumunda hazır bekler. processin kendisinin içindeki örneğin main'in kendiside 1 threaddir tabi. Ama process de kabiliyet çok, donanım vs. **Thread** ise processin içinde yaratılır. daha az zaman gerekir, hafızayı paylaşırlar, threadler arası veri alışverişi mümkün. az kaynak, processden hızlı oluşur ve az zaman alır. Genel olarak, bir işlem (**process**) donanım kaynaklarına (bellek, dosya sistemine erişim, aygıtlara erişim vb.) sahipken, bir iş parçacığı (**thread**) donanım kaynaklarına doğrudan erişime sahip değildir. Bununla birlikte, bir iş parçacığı, bağlı olduğu işlem tarafından sağlanan donanım kaynaklarına erişebilir.

**thread:** donanım dan boş core u istiyoruz ve fonksiyonumuzu ( **task** ınızı) onda çalıştırıyoruz. veya OS task switch yapıor.

```
// task_x  return int yapsaydı, bu out parametreyi kullanmanın i
void task_x(int , int , int){
   // CODE
}

int main(){
   thread th1{ []{ cout << "hello "; } }; // thread copy yok, mo

   // ilgili scope içinde thread yaratıldıysa kesinlikle çağrılı
```

```
    thread th2{ task_x , 20 , 50 , 80 };   // thread yaratıp çağı

    // th2.joinable() == true oldu
    th2.join();  // task_x girilecek (işi bitecek demek değil) ve
    // @178 ,  th2.joinable() == false oldu, çağırdık çünkü
    th2.join(); // HATA, 2 KEZ ÇAĞRILMAZ

    func(); //  bu func excaption throw yaparsa th1 in threadı ça

    th1.detach();  // @179 satırına direk geçilecek veya OS taski
    // @179
}
```

**jthread**: C++ 20 de geldi RAII idiomu var bunda. C++ 20 varsa hep bunu kullan !!

```
int main(){
    thread th1{ []{ cout << "hello1 "; } };
    thread th2{ []{ cout << "hello2 "; } };

    th2.join();// Yani, th2 önce başlayacak ancak tamamlanması th
    th1.join();// taskların çağrılıp bitişi deterministik değil
}
```

```
class my_class{
     void task_x(int){

     }
}
int main(){
    my_class   obj;
    thread th1{ &my_class::task_x , obj , 5 };
```

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

## DERS 65

threadin callableı copy ctor de ve move ctorde çağırmasın istiyorsan , yani en verimli çağırma yolu →

```
class my_class{

};

void my_task( my_task & me){
// osyncstream C++20 -> farklı threadlerdeki std::cout ların çal
    osyncstream{ cout } << std::this_thread::get_id() ;  // thre
}

int main(){

    my_class obj;

    std::thread th{ my_task , std::ref(obj) };   // en verimli y

    obj.join();
}
```

**sleep_for**: duration ister. duration kadar uyutur.

**sleep_until**: time_point ister. uyutur. time pointe gelinceye kadar uyur.

```
void my_task(  ){
    std::this_thread::sleep_for(2300ms); // 2300 ms uyur direk
    std::cout << "uyandım" ;
}

int main(){
```

```
    std::thread th{ my_task };

    obj.join();
}
```

sleep_until: daha az kullanılıo ama anla:

```
auto now(){
   return std::chrono::Steady_clock::now();
}
auto awake_time(){
   using namespace std::literals;
   return now() + 2000ms;
}

int main(){

   cout << "test staring";

   const auto start = now();

   std::this_thread::Sleep_until(awake_time());  // Unutmaki mai

   std::chrono::Duration<double, std::milli>   elapsed { now()-s

   cout << "waited" << elapsed.count() ;
}
```

**std::this_thread::yield();** Demek, bana öncelik verme , git başka thredlere öncelik ver, ben çalışmasam da olur diyor!

**NOT**: std::thread aslında bir abstractıon sağlar. İlgili PC nin işletim sisteminin API leri concoruncyi kontrol edebileceğin bir arayüz sağlar bize. Ama burda çok fazla karmaşa ve OS lere bağlı farklı interface fonksiyonları ve yapıları bulunur.

std::thread bunların arasında bir abstractıon sağlar bize. Ancak bunun zararı, OS nin concoruncye dair tüm özelliklerine ulaşamayız std::thread içinde. Örneğin priority belirlenmez. Ama ilgili OS nin API sinin handle(me) objesine ulaşıp, OS API içinde bu işleri yapmak mümkündür.

```
int main(){
   std::thread th{  []{} };

   auto os_handler = th.native_handler();

   // şimdi artık ilgili os işlemleri yapılır, örnek:
   // windows_task_priorty_set(os_handler, 10);   direk OS API y
}

int my_pc_core_count = std::thread::hardware_concurrency();  //
```

thread içinde axcaption yakalamanın yolu : **std::current_excaptıon()** ile ilgili excaptıonı yakalayıp threadiçinde saklarız ve global bir bu tipden tanımlanan değişkene değer atıp başka bir yerde kontrol ederiz.

**dinamic storage class**: gördük, allocation

**static storage class**: gördük, class içi static veriler

**otomatic storage class**: gördük, normal değişken tanımlama stackte

**thread local storage class (moderc c++ ve yeni C de geldi bu ):** herşeyin ilgili thread üzerinde olduğu, sanki ayrı app ler miş gibi, global alanının , değişkenlerin vs threadlere özel olduğunu düşün. thread local değişken yaratabiliyorsun. **thread_local** keywordü,  koruma gerekmez bu değişkenler için. threade özel çünkü. paylaşılan değişken değil yani.

```
void foo(){
        int x = 9;              // otomatic storage
        static int y = 6;       // static storage
        int *z = malloc(..);    // dinamic storage
        thread_local int k = 5; // thread local storage , Bu ayı
}
```

```
thread_local  int x = 0;  // bu değişken artık her thread için ö

void task(const std::String & name){
  x++;  // her task / thread için x ayrı yaratıldı yani
  osynctream{cout} << name << x; // yani bu 3 threadde x = 1 ola
}

int main(){
   std::thread th1{  task , "task1" };
   std::thread th2{  task , "task2" };
     std::thread th3{  task , "task3" };

   th1.join();  th2.join();  th3.join();
}
```

**std::mutex:** paylaşımlı global değişkenler problem. korumak lazım. okumak için kullanıyorsanda problem yok. (yani const ise sorun yok) ama global değişkenlerim değeri değiştiriliyor ise problem (**data race-tanımsız davranış**). (atamic de değilse )

**data consition**: paylaşımlı değişkenlerin kullanılması ama tanımsız davranış yok demek. oladabilir de tabi , data race i kapsar yani.

```
int g_count = 0;

void test_1(){
   for(..10000..){
     g_count++;
   }
}
void test_2(){
   for(..10000..){
     g_count--;
   }
}
```

```
int main(){
    test_1();
    test_2();
    // g_count = 0 oldu garanti altında
}
```

```
int g_count = 0;

void test_1(){
   for(..10000..){
     g_count++;
   }
}
void test_2(){
   for(..10000..){
     g_count--;
   }
}

int main(){
    std::thread th1{test_1};
  std::thread th2{test_2};
    th1.join();th2.join();
    // g_count nin ne olduğunu Allah bilir :) kormak lazım
}
```

```
std::Atomic int g_count = 0; // araya bu değişkenin işi bitmeden

void test_1(){
   for(..10000..){
     g_count++;
   }
}
void test_2(){
```

```
   for(..10000..){
     g_count--;
   }
}

int main(){
    std::thread th1{test_1};
  std::thread th2{test_2};
    th1.join();th2.join();
    // g_count = 0 olur kesin garanti altında atamic ile
}
```

```
int g_count = 0;
std::mutex mtx;

void test_1(){
   for(..10000..){
      mtx.lock();
     g_count++;
     mtx.unlock();
   }
}
void test_2(){
   for(..10000..){
     mtx.lock();
     g_count--;
     mtx.unlock();
   }
}

int main(){
    std::thread th1{test_1};
  std::thread th2{test_2};
    th1.join();th2.join();
```

```
    // g_count = 0 garanti
}
```

## DERS 66

**std::atomic x{0};** bu x yazılırken veya okunurken bölünemez olur.

```
std::mutex                  *** en sık bu tabi, mutex atomicdir
std::timed_mutex
std::recursive_mutex
std::recursive_timed_mutex
std::shared_mutex     ** okuma hep serbest , yazmaya karşı kilit
std::shared_timed_mutex

RAII li mutexler:
lock_quard              ***** en sık
unique_lock
scoced_lock
shared_lock

----------------

std::mutex mtx;

if(mtx.try_lock()){ // true false , kontrol et çünkü:
   // aynı fonk içinde mutexi 2 kere üst üste kitleme dead lock
}

while(!mtx.try_lock()){  // mutexin boşa çıkmasını bekler

}
```

**dead lock:** bir threadın ilerleyememesi demek.(mesela: mutexi unlock etmemek ) çoklu mutex kullanımından kaçınılmalı. std::lock korur çoklu mutex için. ama lock kullanacağına düzgün sırayla ve algoritmayla mutexi kullan ki dead lock olmasın.

`std::lock` Bu fonksiyon, birden fazla kilitleme nesnesini (mutex, unique_lock, lock_guard vb.) atomik bir şekilde kilitlemek için kullanılır.

## DERS 67

**std::call_once:** ilgili fonksiyonu 50 thread çağırsada 1 kere çağrılmasını sağlar

**Not**: static yerel değişkenler thread safedir c++ da !!

scott meyers in singletonı komple thread safe ! fonk dahil!

`std::async`, C++11. Bu fonksiyon, bir iş parçacığında veya başka bir yolla asenkron olarak bir işlevi çağırmak için kullanılır. `std::async`, C++'ın iş parçacığı (thread) ve gelecek (future) kütüphanelerini kullanarak işlev çağrısını başlatır ve işlevin dönüş değerini döndüren bir `std::future` nesnesi döndürür. Bu sayede, işlevin sonucunu gelecekteki bir zamanda almak için `std::future` kullanılabilir.

book**: C++ Concurrency in Action: Practical Multithreading**