Субъективный объективизм (http://www.scrutator.me/)

(https://twitter.com/ixsci) (http://www.scrutator.me/syndication.axd)

← Добро пожаловать в параллельный мир. Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

Умные указатели → (/post/2012/01/18/smart-pointers.aspx)

Добро пожаловать в параллельный мир. Часть 1: Мир многопоточный (/post/2012/04/04/parallel-world-p1.aspx)

4. апреля 2012 ixSci (http://www.scrutator.me/author/Adminaspx) разработка (/category/разработка.aspx) (4) (/post/2012/04/04/parallel-world-p1.aspx#comment)

Ну вот мы и дождались, C++, наконец, перестал игнорировать ситуацию за окном и признал, что “исполняемый мир” может состоять не только из одного потока исполнения. Действительно, это очень серьезный шаг в становлении C++, как современного языка, т.к. существующая ранее модель, описанная в стандарте, не позволяла (http://www.cs.toronto.edu/~demke/2227S.08/Papers/p261-boehm.pdf) создать кросс-платформенной библиотеки для работы. Реально, изменений в сторону появления многозадачности в C++ 2: появление новой модели памяти и добавление библиотеки с примитивами многозадачности. В данной статье(часть 1) и в следующей(часть 2 (/post/2012/06/03/parallel-world-p2.aspx)) речь пойдет об этих самых примитивах. Об изменениях в модели памяти и всём том, что это привнесло в язык речь пойдёт в последующей статье(часть 3 (/post/2012/08/28/parallel-world-p3.aspx)).

> В статье подразумевается, что читатель знаком с азами мульти-поточного программирования и ему знакомы такие понятия как mutex, deadlock и data races.

Добавляем многозадачность

Итак, встречайте: в C++ появился долгожданный объект *std::thread*! Объект этого класса представляет собой поток исполнения и довольно прост в использовании:

```
#include <iostream>
#include <thread>
#include <string>

int main()
{
    auto func = [](const std::string& first, const std::string& second)
    {
        std::cout << first << second;
    };
    std::thread thread(func, "Hello, ", "threads!");
    thread.join();
}
```

Из примера видно, что конструктор *std::thread* принимает первым аргументом функцию исполнения, т.е. функцию, код которой будет исполнен в отдельном потоке. Остальные аргументы, - есть аргументы исполняемой функции. Количество аргументов ограничено лишь реализацией variadic templates в вашем компиляторе. Важно помнить, что аргументы будут использованы в другом потоке исполнения, а следовательно **нельзя** передавать ссылки и указатели на объекты, время жизни которых не больше, чем время жизни потока. Это может обернуться некорректным поведением, в лучшем случае, и падением, в худшем. Всегда нужно помнить об этом.

Так же *thread* всегда копирует аргументы, и только потом передаёт их исполняемой функции. Поэтому, даже если ваша функция принимает ссылку в качестве аргумента, это будет не та ссылка, которую вы передали в конструкторе *thread*. Это будет ссылка на копию в объекте *thread*! Поэтому, для передачи ссылки необходимо использовать *std::ref.*

Функция начинает свое исполнения сразу по окончании работы конструктора *std::thread*. Завершение потока происходит по завершении работы исполняемой функции. В дальнейшем, я буду именовать поток исполнения, созданный посредством конструирования *std::thread*, **созданным потоком**. После того, как объект *std::thread* создан возможны три варианта развития событий:

1. Пользователь выполнил *thread.join()*. Это означает, что поток исполнения, который вызвал *join*, будет ожидать завершения исполнения **созданного потока**. Блокирует вызывающий поток.
2. Пользователь выполнил *thread.detach().* Это означает, что пользователя не интересует судьба **созданного потока** и главный поток исполнения может завершится до того как будет завершён оный**.** Не блокирует вызывающий поток.
3. Ни один из вышеупомянутых методов не был вызван. Это приведёт к вызову *std::termination* в

деструкторе объекта thread.

Вряд ли, кто-то хочет испытать на себе 3-й случай, поэтому всегда вызывайте либо *join* либо *detach,* до того, как будет вызван деструктор объекта *std::thread*. При этом использование *join* предпочтительнее. Так как *detach* используется тогда, когда это действительно нужно или оправдано, в силу того, что представляет меньше контроля за исполнением.

> Или же наоборот, предоставляется больше контроля. Т.к. можно синхронизироваться на различных примитивах, а не просто использовать join. В общем, если join вам не подходит используйте detach. Если вы не знаете, что выбрать, – выбирайте join.

В классе *thread*, так же, существует статическая функция *std::hardware_concurrency*, которая может вернуть количество потоков, которые могут выполнятся параллельно. А может и не вернуть  Вот такая вот функция. Если количество определить не удалось, тогда будет возвращен 0. Справедливости ради: у меня с использованием MSVS2011 beta она выдаёт 6, что соответствует действительности.

Идентификация потока

Бывают моменты, когда необходимо узнать, в каком потоке исполнения вы находитесь. Вариантов может быть масса: логгирование, сравнение идентификаторов различных потоков и т.п. Для решения этой задачи стандарт предлагает решение в виде уникального идентификатора *id,* который ассоциирован с каждым потоком исполнения. После завершения потока исполнения, правда, стандарт разрешает использовать *id,* почившего в бозе потока, снова. Это стоит учитывать, при завязывании какой бы то ни было логики на *id* потока. Этот *id* может быть получен из объекта *thread* с помощью метода get_id.

```
1  std::cout << "New thread id: 0x" << std::hex << thread.get_id() << "\n";
```

Следует, также, понимать, что этот *id* **может не иметь** никакого отношения к платформо-зависмым идентификаторам потоков. Поэтому не стоит полагаться на то, что они будут идентичны. Хотя, к примеру, в реализации от Microsoft в VS2011 beta это именно так:

```
1  std::cout << "Id from WinAPI: 0x" << std::hex << ::GetCurrentThreadId()
2      << "\n";
3  std::cout <<"Id from C++ API: 0x"<< std::hex << std::this_thread::get_id()
4       << "\n";
```

Вывод на моей машине:

```
Id from WinAPI: 0x1318
Id from C++ API: 0x1318
```

Но, по иронии судьбы, вы, все равно, не сможете использовать идентификатор полученный средствами C++ API в системно-зависимом API. В силу того, что никакой конвертации из thread::id в системно зависимый тип не существует.

> Можно конечно написать свою конвертацию, вытаскивая id из ostrream, но зачем?

Полезное приложение

Для упрощения работы с потоками в заголовке *thread* существует пространство имен *this_thread*, которое содержит следующие функции:

- *get_id* – возвращает идентификатор потока исполнения, в котором она вызвана
- *yield* – сигнализирует ОС, что поток желает приостановить свое выполнение и дать шанс на исполнение другим потокам. Результат зависит от многих факторов и ОС, поэтому не думаю, что его стоит обсуждать.
- *sleep_until* – поток приостанавливает выполнение до наступления момента, переданного в качестве аргумента.
- *sleep_for* – поток приостанавливает выполнение на некий, заданный промежуток времени.

> Касательно двух последних функций: гарантируется, что поток не будет пробужден раньше срока, который вы задали ему с помощью этих функций. Но нет никакой гарантии, что поток проснется именно в тот момент, который вы рассчитали. Он может проснуться гораздо позже, т.к. все это зависит от ОС и степени её нагрузки в данный момент.

```
auto days = std::chrono::system_clock::now() + std::chrono::hours(72);
std::this_thread::sleep_until(days);
std::this_thread::sleep_for(std::chrono::hours(72));
```

Обе функции кладут поток в сон на 3 дня.

> В коде выше используется код из новой библиотеки chrono, которая находится в заголовке <chrono>.

Контролируем доступ к ресурсу

Когда появляется возможность параллельного исполнения кода, очень часто возникает необходимость конкурентного доступа к разделяемому ресурсу. Будь то контейнер, переменная интегрального типа или умный указатель, – не важно. Важно то, что для получения предсказуемого результата от нашей операции над разделяемым ресурсом, мы должны упорядочить доступ к оному. В любой мульти-потоковой среде есть свои примитивы для выполнения этой задачи. C++ не стал исключением:

mutex

Я, признаюсь, не проводил никаких исследований, но мне кажется, что *mutex* является наиболее часто используемым примитивом для защиты разделяемого ресурса. Видимо эта популярность вызвана его простотой.

C++ предоставляет нам 3 типа операций над базовыми мьютексами:

- *lock* – если мьютекс не принадлежит никакому потоку, тогда поток, вызвавший *lock*, становится его обладателем. Если же некий поток уже владеет мьютексом, то текущий поток(который пытается овладеть им) блокируется до тех пор, пока мьютекс не будет освобожден и у него не появится шанса овладеть им.
- *try_lock* - если мьютекс не принадлежит никакому потоку, тогда поток, вызвавший *try_lock*, становится его обладателем и метод возвращает *true.* В противном случае возвращает *false*. *try_lock* **не** блокирует текущий поток.
- *unlock* – освобождает ранее захваченный мьютекс.

… 2 дополнительные для **временн*ы*х(timed)** мьютексов:

- *try_lock_for* – расширенная версия *try_lock*, которая позволяет задать продолжительность ожидания, прежде чем стоит прекратить попытку овладения мьютексом. Т.е. возвращает *true* в том случае, если удалось овладеть мьютексом в заданный промежуток времени. В противном случае возвращает *false*. Принимает *std::chrono::duration*, в качестве аргумента.
- *try_lock_until* – та же, что предыдущая, но принимает *std::chrono::time_point* в качестве аргумента.

… и 4 типа *mutex*:

- *std::mutex* – базовый *mutex*, которым может владеть один поток в единицу времени. При попытке повторного овладения *мьютексом*, потоком, уже владеющим им, произойдёт *deadlock(*или будет брошено исключение с кодом ошибки *resource_deadlock_would_occur)*.
- *std::recursive_mutex* – обладает теми же свойствами, что и *std::mutex*, но позволяет рекурсивное овладение мьютексом, сиречь многократный вызов метода *lock()* в потоке, который владеет мьютексом. При этом, метод *unlock()* должен быть вызван не меньшее количество раз, чем был вызван *lock().* В противном случае вы получите deadlock, т.к. этот

поток никогда не освободит *мьютекс* и остальные потоки будут находиться в вечном ожидании.

- *std::timed_mutex* – обладая свойствами *std::mutex, std::timed_mutex,* так же, обладает дополнительными методами позволяющими блокировку на время.
- std::recursive_timed_mutex – рекуррентная версия *std::timed_mutex.*

Приведу небольшой пример, с std::mutex, для передачи общей идеи.

В примере будут использованы следующие заголовки:

```
#include <thread>
#include <chrono>
#include <iostream>
#include <mutex>
#include <list>
#include <limits>
```

Класс Warehouse будет нашим разделяемым ресурсом, в который кладутся товары и из которого они изымаются.

```
class WarehouseEmpty
{
};

unsigned short c_SpecialItem =
    std::numeric_limits<unsigned short>::max();

class Warehouse
{
    std::list<unsigned short> m_Store;
public:
    void AcceptItem(unsigned short item)
    {
        m_Store.push_back(item);
    }
    unsigned short HandLastItem()
    {
        if(m_Store.empty())
            throw WarehouseEmpty();
        unsigned short item = m_Store.front();
        if(item != c_SpecialItem)
            m_Store.pop_front();
        return item;
    }
};

Warehouse g_FirstWarehouse;
```

```
Warehouse g_SecondWarehouse;
std::timed_mutex g_FirstMutex;
std::mutex g_SecondMutex;
```

Для заполнения складов(warehouses) будем использовать поставщика(supplier), который заполняет склады по очереди, если кто-то другой не пользуется складами(не обладает мьютексом), или в случайном порядке, если кто-то пользуется складом. В конце работы, поставщик помещает специальные товары, по которым потребитель поймёт, что товаров больше ждать не следует.

```
auto suplier = []()
{
    for(unsigned short i = 0, j = 0; i < 10 && j < 10;)
    {
        if(i < 10 && g_FirstMutex.try_lock())
        {
            g_FirstWarehouse.AcceptItem(i);
            i++;
            g_FirstMutex.unlock();
        }
        if(j < 10 && g_SecondMutex.try_lock())
        {
            g_SecondWarehouse.AcceptItem(j);
            j++;
            g_SecondMutex.unlock();
        }
        std::this_thread::yield();
    }
    g_FirstMutex.lock();
    g_SecondMutex.lock();
    g_FirstWarehouse.AcceptItem(c_SpecialItem);
    g_SecondWarehouse.AcceptItem(c_SpecialItem);
    g_FirstMutex.unlock();
    g_SecondMutex.unlock();
};
```

Первый потребитель приходит на первый склад и ждёт своей очереди в получении товара:

```
auto consumer = []()
{
    while(true)
    {
        g_FirstMutex.lock();
        unsigned short item = 0;
        try
        {
```

```
            item = g_FirstWarehouse.HandLastItem();
        }
        catch(Warehouse)
        {
            std::cout << "Warehouse is empty!\n";
        }
        g_FirstMutex.unlock();
        if(item == c_SpecialItem)
            break;
        std::cout << "Got new item: " << item << "!\n";
        std::this_thread::sleep_for(std::chrono::seconds(4));
    }
};
```

Второй, же, покупатель нетерпелив, он не хочет долго ждать, пока сможет воспользоваться первым складом. Поэтому он ждёт несколько секунд, после чего идёт, возмущенный, на второй склад:

```
auto impatientConsumer = []()
{
    while(true)
    {
        unsigned short item = 0;
        if(g_FirstMutex.try_lock_for(std::chrono::seconds(2)))
        {
            try
            {
                item = g_FirstWarehouse.HandLastItem();
            }
            catch(Warehouse)
            {
                std::cout << "Warehouse is empty! I'm mad!!!11\n";
            }
            g_FirstMutex.unlock();
        }
        else
        {
            std::cout << "First warehouse is always busy!!!\n";
            g_SecondMutex.lock();
            try
            {
                item = g_SecondWarehouse.HandLastItem();
            }
            catch(Warehouse)
            {
                std::cout << "2nd warehouse is empty!!!!11\n";
            }
            g_SecondMutex.unlock();
```

```
        }
        if(item == c_SpecialItem)
            break;
        std::cout << "At last I got new item: " << item << "!\n";
        std::this_thread::sleep_for(std::chrono::seconds(4));
    }
};
```

Ну и код запуска всего этого механизма:

```
std::thread supplierThread(suplier);
std::thread consumerThread(consumer);
std::thread impatientConsumerThread(impatientConsumer);

supplierThread.join();
consumerThread.join();
impatientConsumerThread.join();
```

Весь код, одним куском, лежит тут (http://pastebin.com/MieuMy2W)

Из кода выше вы могли заметить, что даже такая простая задача требует предельного внимания, т.к. вы должны всегда быть начеку. Контролировать количество *lock/unlock*, быть уверенным, что вы всюду оградили разделяемый ресурс от **гонок(data race)**и т.п. Хотя всех проблем избежать нельзя, у C++ есть, что предложить для того, чтобы сделать нашу жизнь чуточку легче. И одним из таких средств является:

std::lock_guard

lock_guard это реализация RAII (http://ru.wikipedia.org/wiki/RAII) принципа для *mutex*. При создании объекта *lock_guard* захватывается мьютекс, переданный ему в конструкторе. В деструкторе, же, происходит освобождение мьютекса. Так же, *lock_guard* содержит дополнительный конструктор, который позволяет инициализировать объект *lock_guard* с мьютексом, который уже был захвачен:

```
std::lock_guard<std::mutex> guard(mutex, std::adopt_lock);
```

В общем, довольно простой и легковесный класс, который будет очень полезен при работе с мьютексами. Применим этот класс для упрощения нашего кода:

Поставщик:

```
auto suplier = []()
{
    for(unsigned short i = 0, j = 0; i < 10 && j < 10;)
```

```
    {
        if(i < 100 && g_FirstMutex.try_lock())
        {
            g_FirstWarehouse.AcceptItem(i);
            i++;
            g_FirstMutex.unlock();
        }
        if(j < 100 && g_SecondMutex.try_lock())
        {
            g_SecondWarehouse.AcceptItem(j);
            j++;
            g_SecondMutex.unlock();
        }
        std::this_thread::yield();
    }
    std::lock_guard<std::timed_mutex> firstGuard(g_FirstMutex);
    std::lock_guard<std::mutex> secondGuard(g_SecondMutex);
    g_FirstWarehouse.AcceptItem(c_SpecialItem);
    g_SecondWarehouse.AcceptItem(c_SpecialItem);
};
```

Первый покупатель:

```
auto consumer = []()
{
    while(true)
    {
        unsigned short item = 0;
        try
        {
            std::lock_guard<std::timed_mutex> guard(g_FirstMutex);
            item = g_FirstWarehouse.HandLastItem();
        }
        catch(Warehouse)
        {
            std::cout << "Warehouse is empty!\n";
        }
        if(item == c_SpecialItem)
            break;
        std::cout << "Got new item: " << item << "!\n";
        std::this_thread::sleep_for(std::chrono::seconds(4));
    }
};
```

Второй:

```
auto impatientConsumer = []()
    {
        while(true)
        {
            unsigned short item = 0;
            if(g_FirstMutex.try_lock_for(std::chrono::seconds(2)))
            {
                try
                {
                    std::lock_guard<std::timed_mutex> guard(g_FirstMutex,
                          std::adopt_lock);
                    item = g_FirstWarehouse.HandLastItem();
                }
                catch(Warehouse)
                {
                    std::cout << "Warehouse is empty! I'm mad!!!11\n";
                }
            }
            else
            {
                std::cout << "First warehouse always busy!!!\n";
                try
                {
                    std::lock_guard<std::mutex> guard(g_SecondMutex);
                    item = g_SecondWarehouse.HandLastItem();
                }
                catch(Warehouse)
                {
                    std::cout << "2nd warehouse is empty!!!!11\n";
                }
            }
            if(item == c_SpecialItem)
                break;
            std::cout << "At last I got new item: " << item << "!\n";
            std::this_thread::sleep_for(std::chrono::seconds(4));
        }
    };
```

Хотя кода стало не намного меньше, но он стал проще и более устойчивым к непредвиденным ситуациям. Код, также, стал нагляднее, за счет того, что нет необходимости искать глазами каждый *unlock*.

Вы, наверное, заметили, что захват временного мьютекса остался без изменений. Хотя, от части, это продиктовано тем, что захват этого мьютекса происходит в условном операторе if, и, просто не целесообразно, выносить захват в отдельную строку. В большей мере это продиктовано простым

фактом, *lock_guard* просто не умеет выполнять временной захват. А так как он не умеет этого делать, то нам надо найти другое решение. Благо это решение существует в C++:

std::unique_lock

Функционал *unique_lock* по отношению. к *lock_guard*, можно сравнить с отношением функционала *std::shared_ptr* к *std::unique_ptr*, – он гораздо богаче. Итак, *unique_lock* может:

- Принимать не захваченный мьютекс в конструкторе
- Захватывать и освобождать мьютекс непосредственными вызовами *lock/unlock*
- Выполнять временной захват
- Может быть **перемещен** в другой объект *unique_lock*

Т.е. *unique_lock* может быть использован в любом контексте, в котором может быть использован “голый” мьютекс, гарантируя, при этом, что захваченный мьютекс будет освобожден в деструкторе. Так же, он включает в себя весь небогатый функционал из *lock_guard.*

Применим его на втором покупателе:

```
auto impatientConsumer = []()
{
    while(true)
    {
        unsigned short item = 0;
        std::unique_lock<std::timed_mutex> firstGuard(g_FirstMutex,
         std::defer_lock);
        if(firstGuard.try_lock_for(std::chrono::seconds(2)))
        {
            try
            {
                item = g_FirstWarehouse.HandLastItem();
            }
            catch(Warehouse)
            {
                std::cout << "Warehouse is empty! I'm mad!!!11\n";
            }
            firstGuard.unlock();
        }
        else
        {
            std::cout << "First warehouse always busy!!!\n";
            try
            {
                std::unique_lock<std::mutex> guard(g_SecondMutex);
                item = g_SecondWarehouse.HandLastItem();
            }
            catch(Warehouse)
```

```
            {
                std::cout << "2nd warehouse is empty!!!!11\n";
            }
        }
        if(item == c_SpecialItem)
            break;
        std::cout << "At last I got new item: " << item << "!\n";
        std::this_thread::sleep_for(std::chrono::seconds(4));
    }
};
```

Т.к. его использование в этом случае(применительно к временному мьютексу) избыточно и усложняет код, я не рекомендую его использовать таким образом. Я это сделал лишь ради примера.

Алгоритм выбора между *lock_guard* и *unique_lock* может быть следующим: всегда использовать *lock_guard*, пока хватает его функционала. В остальных случая использовать стоит unique_lock. При этом, как и в случае с указателями, я рекомендую использовать эти обертки вместо “голых” мьютексов везде, где это возможно и целесообразно .

Полезное приложение

В примерах выше, мы рассматривали только случаи, когда в единицу времени нам необходим доступ только к одному разделяемому ресурсу. Но бывают случаи, когда доступ необходим к нескольким разделяемым ресурсам одновременно. Как правило, у каждого такого ресурса существует свой, ассоциированный, с ним примитив синхронизации доступа(std::mutex, к примеру).

Рассмотрим пример такой функции:

```
auto call = [](std::mutex& first, std::mutex& second)
{
        first.lock();//#1
        second.lock();//#2
        //Что-то делаем
        first.unlock();
        second.unlock();
};
```

И её вызов:

```
std::mutex first;
std::mutex second;
std::thread firstThred(call, std::ref(first), std::ref(second));
std::thread secondThred(call, std::ref(second), std::ref(first));
```

Выполняя подобный код, может возникнуть ситуация, когда оба потока одновременно выполнили **#1**. Получается, что **firstThread** будет ожидать освобождения **second**, при этом захватив **first**, а **secondThread** будет ожидать освобождения **first**, при этом захватив **second**. Классический deadlock, когда два или больше потока захватывают несколько мьютексов в различном порядке. Дабы избежать подобных проблем настоятельно рекомендуется следить за порядком, в котором вы захватываете мьютексы.

В этом, выдуманном примере, это, несомненно, сделать довольно просто. Но что если ситуация куда сложнее: мьютексов больше, мест, где они используются,- много. От этого у кого хочешь голова вспухнет и начнут появляться ошибки.

К счастью, C++ может выручить нас и на этот раз. Специально, для решения таких проблем существует функция *std::lock*, которая принимает переменное число аргументов, которые должны иметь *lock*, *unlock*, *try_lock* методы. Для простоты изложения, сузим тип дозволенных аргументов до *std::mutex*. Эта функция гарантирует, что если она завершится успешно, то все переданные ей, в качестве аргументов, мьютексы будут захвачены, и **не произойдет** deadlock, в не зависимости от того, в каком порядке были переданы аргументы. Мерилом неуспешности является исключение, брошенное при попытке вызова *lock/try_lock* метода. При этом, гарантируется, что все те мьютексы, что были захвачены в функции *lock*, до момента срабатывания исключения будут освобождены.

Сделаем наш пример свободным от deadlock:

```
auto call = [](std::mutex& first, std::mutex& second)
{
        std::lock(first, second);
        //Что-то делаем
        first.unlock();
        second.unlock();
};
```

Побратимом вышеописанной функции является *std::try_lock*, поведение которой идентично *std::lock.* За тем исключением, что *try_lock* возвращает **–1** при удачном исполнении, и номер аргумента, для которого *try_lock*-метод вернул false, в противном случае.

В арсенале около-блокирующих, вспомогательных функций есть еще одна важная функция: *std::call_once*. Она выступает в тандеме со структурой *std::once_flag*. Целью данной функции, как вытекает из названия, является гарантия вызова некой процедуры один раз. Т.е. используя функцию *call_once*, с определенным объектом типа *once_flag*, вы гарантируете, что функция(переданная в аргументах *call_once*) будет вызвана один раз. В не зависимости от того, сколько потоков пытаются её вызвать. Только один из них преуспеет, а именно: тот, который “добрался” до неё первый. Остальные будут либо ждать окончания её завершения, если она еще не была вызвана, либо будут просто “пропускать” её, т.к. она уже была однажды вызвана.

Гарантия единственности вызова зиждется на объекте типа *once_flag*, который хранит всю необходимую информацию о выполнении или о не выполнении функции. Т.е. если использовать разные объекты *once_flag*, тогда функция будет вызвана столько раз, сколько разных флагов было передано:

```
1  std::once_flag flag;
2  std::call_once(flag,[&](){std::cout << "I'm called!\n";});
3  std::call_once(flag,[&](){std::cout << "One more call!\n";});
4  std::once_flag otherFlag;
5  std::call_once(otherFlag,[&](){std::cout << "Another flag\n";});
```

Вывод:

```
1  I'm called!
2  Another flag
```

Еще одним важным свойством этой функции является гарантия того, что после завершения вызова(активного) все последующие вызовы(пассивные) могут использовать всё, что было изменено в результате активного вызова. К примеру, если вы инициализируете какие-то то данные посредством *call_once*, вы можете смело их использовать сразу после вызова; в любом потоке, т.к. эти изменения будут видны во всех потоках и это гарантированно стандартом! Таким образом, нам не грозит не соответствие кэшей в двух различных ядрах, по отношению к инициализируемым нами данным.

condition_variable

Еще одним блокирующим примитивом в C++ является *std::condition_variable*. Привычным шаблоном использования подобного примитива является ожидание одного потока, наступления события в другом потоке. Можете думать об этом примитиве как о неком сигнале, появления которого необходимо ожидать.

Основными методами *condition_variable* являются:

- *wait* – ставит поток в ожидание сигнала. Ожидание не лимитировано временем. Может принимать в качестве аргумента предикат, от результата которого будет зависеть выход потока из ожидания. Т.е. если даже *wait* был завершен благодаря сигналу, происходит проверка предиката после чего поток снова становится в ожидание, если предикат ложен. На псевдокоде: (*while(!predicate) wait;*). А нужно это, в первую очередь, для того, чтобы избежать реагирования на *фальшивое(spurious)* пробуждение(см. врезку ниже).
- *wait_for* – Ожидание лимитировано согласно аргументу
- *wait_until* - Ожидание лимитировано согласно аргументу
- *notify_one* – Посылает сигнал одному из ожидающих потоков; т.е. разблокирует один поток.

Какой поток будет разбужен – не известно. Гарантировано лишь то, что один из них будет.

- *notify_all* – Посылает сигнал всем ожидающим потокам; т.е. разблокирует все потоки ожидающие на данном объекте *condition_variable*

> Фальшивое(spurious) пробуждение это когда wait завершается, без участия notify_one или notify_all. Да, к сожалению, и такое может быть и это дозволяется стандартом POSIX, подробнее читайте тут (http://en.wikipedia.org/wiki/Spurious_wakeup)

condition_variable не является самостоятельным примитивом, т.к. *wait* происходит на объекте *std::unique_lock*. При этом *unique_lock* должен быть захвачен перед тем как будет передан в функцию *wait.* Более того, вы должны гарантировать, что все *wait,* для данного объекта *condition_variable,* выполнены на одном и том же мьютексе.

Есть и другой тип *condition_variable*, – *condition_variable_any*, который, в отличие от своего собрата, может принимать любой объект, на котором можно выполнить *lock/unlock*. А это все мьютексы и обертки для них(*lock_guard, unique_lock*). Пользователь, так же, может добавить свой тип. Так как больше отличий не имеется, то и упоминать *condition_variable_any* отдельно я, в дальнейшем, не буду .

Рассмотрим пример использования и в качестве примера возьмём небольшую историю:

Один, заурядный, менеджера приходит на работу звонит в звонок и ждёт пока охранник откроет ему дверь:

```
1  std::condition_variable g_Bell;
2  std::condition_variable_any g_Door;
3
4  class Manager
5  {
6  public:
7      void ComeToWork()
8      {
9          std::cout << "Hey security, please open the door!\n";
10         g_Bell.notify_one();
11         std::mutex mutex;
12         mutex.lock();
13         g_Door.wait(mutex);
14         mutex.unlock();
15     }
16 };
```

Работа охранника, в свою очередь, ожидать звонка в звонок и открывать дверь менеджеру. Если звонок не звонит, значит можно спать. Вот такая работа. Но у него есть ещё одна ответственная обязанность, взваленная на него программистами фирмы: он должен предупредить их, если менеджер вдруг придёт неожиданно на работу:

```
class Security
{
    static bool m_SectorClear;
    static std::mutex m_SectorMutex;
public:
    static bool SectorClear()
    {
        std::lock_guard<std::mutex> lock(m_SectorMutex);
        return m_SectorClear;
    }
    void NotifyFellows()
    {
        std::lock_guard<std::mutex> lock(m_SectorMutex);
        m_SectorClear = false;
    }
    void WorkHard()
    {
        m_SectorClear = true;
        std::mutex mutex;
        std::unique_lock<std::mutex> lock(mutex);
        while(true)
        {
            if(g_Bell.wait_for(lock, std::chrono::seconds(5)) ==
        std::cv_status::timeout)
                std::this_thread::sleep_for(std::chrono::seconds(10));
            else
            {
                NotifyFellows();
                g_Door.notify_one();
                std::cout << "Hello Great Manager, your slaves are"
             "ready to serve you!\n" << std::endl;
            }
        }
    }
};
```

Программисты, же, в отсутствие менеджеры заняты очень важным делом: они играют в StarCraft. Но менеджер не понимает насколько это важное дело, и поэтому они должны переключиться на работу, когда менеджер появится в офисе:

```
class Programmer
{
public:
    void WorkHard()
    {
        std::cout << "Let's write some govnokod!\n" << std::endl;
        int i = 0;
        while(true)
        {
            i++;
            i--;
        }
    }
    void PlayStarcraft()
    {
        while(Security::SectorClear())
            ;//Играем! :)
        WorkHard();// Работаем :(
    }
};
```

Полный код можно найти здесь (http://pastebin.com/NuG8QWYL)

Надеюсь пример достаточно нагляден и не требует пояснений. Но заметьте, что нет гарантии, того, что менеджер не успеет увидеть как его подчинённые “работают” 

Завершить статью я хочу подходящей для этого функцией: *std::notify_all_at_thread_exit*.

Эта функция принимает в качестве аргументов *condition_variable* и *unique_lock*, что, собственно, ожидаемо, и, следовательно, все ограничения *condition_variable* должны быть соблюдены . Её смысл ясно виден из названия: при завершении потока, когда все деструкторы локальных(по отношению к потоку) объектов отработали, выполняется *notify_all* на переданном объекте *condition_variable*. Поток вызвавший notify_all_at_thread_exit будет обладать мьютексом до самого завершения, поэтому необходимо позаботиться о том, чтобы не произошёл deadlock где-нибудь в коде. В общем фукция не для регулярного использования, явно. Она будет полезна, когда вам необходимо гарантировать, что к определенному моменту все локальные объекты определенного потока разрушены. И вы, по какой-то причине, не можете использовать join на объекте потока.

В качестве примера я приведу код, который имитирует join для отвязанного(detached) потока:

```
#include <thread>
#include <mutex>
#include <condition_variable>

```

```
5  std::condition_variable g_Condition;
6  std::mutex g_Mutex;
7
8  int main()
9  {
10     auto call = []()
11     {
12         std::unique_lock<std::mutex> lock(g_Mutex);
13         std::this_thread::sleep_for(std::chrono::seconds(5));
14         std::notify_all_at_thread_exit(g_Condition, std::move(lock));
15     };
16     std::thread callThread(call);
17     callThread.detach();
18     std::unique_lock<std::mutex> lock(g_Mutex);
19     g_Condition.wait(lock);
20     return 0;
21 }
```

Остальные статьи цикла:

Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

Часть 3: Единый и Неделимый (/post/2012/08/28/parallel-world-p3.aspx)

Часть 4: Порядки и отношения (/post/2015/08/14/parallel-world-p4.aspx)

Часть 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)

Метки : C++ (http://www.scrutator.me/?tag=C%2b%2b), thread (http://www.scrutator.me/?tag=thread), mutex (http://www.scrutator.me/?tag=mutex), lock_guard (http://www.scrutator.me/?tag=lock_guard), call_once (http://www.scrutator.me/?tag=call_once), unique_lock (http://www.scrutator.me/?tag=unique_lock), condition_variable (http://www.scrutator.me/?tag=condition_variable)

Текущий рейтинг: 5.0 (6 голосов)

Похожие записи

Добро пожаловать в параллельный мир. Часть 1: Мир многопоточный (/post/2012/04/04/parallel-world-p1.aspx)

Ну вот мы и дождались, C++, наконец, перестал игнорировать ситуацию за окном и признал, что “исполня

Добро пожаловать в параллельный мир. Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

Продолжая тему начатую в части 1, перейдём от простейшего способа использования потоков, к следующем

Добро пожаловать в параллельный мир. Часть 3: Единый и Неделимый (/post/2012/08/28/parallel-world-p3.aspx)

Данная статья является третьей в цикле статей(часть 1 и часть 2) о многозадачности в C++11. По сути,

a...r • 6 месяцев назад

Очень хороший и доступно описанный курс. Читал с большим удовольствием. Всё читается просто на одном дыхании.

мелкие придирки - исправте пожалуйста битую ссылку тут

Остальные статьи цикла:
Часть 2: Мир асинхронный <--

Evgeniy Shcherbina Автор a...r • 6 месяцев назад

Спасибо! Починил ссылку.

Последние записи

Вся правда об указателях. Часть 3: Завершающая (/post/2016/03/30/pointers_demystified_p3.aspx)
Рейтинг: 3.5 / 4

Обзор книги Dependency Injection in .NET (/post/2016/03/19/review_dep_injection.aspx)
Нет оценок

Вся правда об указателях. Часть 2: Памятная (/post/2015/12/30/pointers_demystified_p2.aspx)
Рейтинг: 5 / 2

Обзор книги Applied Cryptography (/post/2015/12/25/review_applied_cryptography.aspx)
Рейтинг: 5 / 1

Вся правда об указателях. Часть 1: Вводная (/post/2015/11/26/pointers_demystified_p1.aspx)
Рейтинг: 5 / 3

Обзор книги C# 5.0 in a Nutshell: The Definitive Reference (/post/2015/11/24/review_csharp5_in_a_nutshell.aspx)
Нет оценок

Добро пожаловать в параллельный мир. Часть 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)
Рейтинг: 5 / 2

Список категорий

(/category/feed/книги.aspx) книги (15) (/category/книги.aspx)

(/category/feed/разработка.aspx) разработка (43) (/category/разработка.aspx)

Список записей по годам/месяцам

2011

2012

2013

2014

2015

2016

Март (/2016/03/default.aspx) (2)