Pride_St 12 марта 2019 в 12:51

Топ 20 ошибок при работе с многопоточностью на C++ и способы избежать их

Программирование*, C++*

Из песочницы

Привет, Хабр! Предлагаю вашему вниманию перевод статьи «Top 20 C++ multithreading mistakes and how to avoid them» автора Deb Haldar.

Сцена из фильма «Петля времени» (2012)

Многопоточность— одна из наиболее сложных областей в программировании, особенно в C++. За годы разработки я совершил множество ошибок. К счастью, большинство из них были выявлены на код ревью и тестировании. Тем не менее, некоторые каким-то образом проскакивали на продуктив, и нам приходилось править эксплуатируемые системы, что всегда дорого.

В этой статье я попытался категоризировать все известные мне ошибки с возможными решениями. Если вам известны еще какие-то подводные камни, либо имеете предложения по решению описанных ошибок– пожалуйста, оставляйте свои комментарии под статьей.

Ошибка №1: Не использовать join() для ожидания фоновых потоков перед завершением приложения

Если вы забыли присоединить поток (join()) или открепить его (detach()) (сделать его не joinable) до завершения программы, это приведет к аварийному завершению. (В переводе будут встречаться слова присоединить в контексте *join()* и открепить в контексте *detach()*, хотя это не совсем корректно. Фактически *join()* это точка, в которой один поток выполнения дожидается завершения другого, и никакого присоединения или объединения потоков не происходит [прим. переводчика]).

В примере ниже, мы забыли выполнить *join()* потока t1 в основном потоке:

```
 #include "stdafx.h"
#include <iostream>
#include <thread>

using namespace std;

void LaunchRocket()
{
   cout << "Launching Rocket" << endl;
}
int main()
{
   thread t1(LaunchRocket);
   //t1.join(); // как только мы забыли join- мы получаем аварийное завер
шение программы
   return 0;
}
```

Почему программа упала?! Потому что в конце функции *main()* переменная t1 вышла из области видимости и был вызван деструктор потока. В деструкторе происходит проверка является ли поток t1 *joinable*. Поток является *joinable*, если он не был откреплен. В этом

случае в его деструкторе вызывается std::terminate. Вот что, например, делает компилятор MSVC++.

```
 ~thread() _NOEXCEPT
{  // clean up
    if (joinable())
        XSTD terminate();
}
```

Есть два способа исправления проблемы в зависимости от задачи:

1. Вызвать *join()* потока t1 в основном потоке:

```
 int main()
{
  thread t1(LaunchRocket);
  t1.join(); // выполняем join потока t1, ожидаем завершение этого потока
 в основном потоке выполнения
    return 0;
}
```

2. Открепить поток t1 от основного потока, позволить ему продолжить работать как «демонизированный» поток:

```
 int main()
{
    thread t1(LaunchRocket);
    t1.detach(); // открепление  t1 от основного потока
    return 0;
}
```

Ошибка №2: Пытаться присоединить поток, который ранее был откреплен

Если в какой-то точке работы программы у вас есть открепленный (*detach*) поток, вы не можете присоединить его обратно к основному потоку. Это очень очевидная ошибка. Проблема в том, что вы можете открепить поток, а потом написать несколько сотен строк кода

и попробовать вновь присоединить его. В конце концов, кто помнит, что он писал 300 строк назад, верно?

Проблема в том, что это не вызовет ошибку компиляции, вместо этого программа аварийно завершится при запуске. Например:

```
 #include "stdafx.h"
#include <iostream>
#include <thread>

using namespace std;

void LaunchRocket()
{
    cout << "Launching Rocket" << endl;
}

int main()
{
    thread t1(LaunchRocket);
    t1.detach();
    //..... 100 строк какого-то кода
    t1.join(); // CRASH !!!
    return 0;
}
```

Решение заключается в том, что необходимо всегда делать проверку потока на *joinable()* перед тем как пытаться его присоединить к вызывающему потоку.

```
 int main()
{
  thread t1(LaunchRocket);
  t1.detach();
  //..... 100 строк какого-то кода

  if (t1.joinable())
  {
```

```
    t1.join();
  }

  return 0;
}
```

Ошибка №3: Непонимание того, что std::thread::join() блокирует вызывающий поток выполнения

В реальных приложениях вам часто может потребоваться выделить в отдельный поток «долгоиграющие» операции обработки сетевого ввода-вывода или ожидания нажатия пользователя на кнопку и т.п. Вызов *join()* для таких рабочих потоков (например поток отрисовки UI) может привести к зависанию пользовательского интерфейса. Существуют более подходящие способы реализации.

Например, в GUI приложениях рабочий поток при завершении может отправить сообщение UI потоку. UI поток имеет собственный цикл обработки событий таких как: перемещение мыши, нажатие на клавиши и т.д. Этот цикл также может принимать сообщения от рабочих потоков и реагировать на них без необходимости вызова блокирующего метода *join()*.

По этой самой причине в платформе WinRT от Microsoft практически все взаимодействия с пользователем сделаны асинхронными, а синхронные альтернативы недоступны. Эти решения были приняты для гарантии того, что разработчики будут использовать API, которое предоставляет наилучший опыт использования для конечных пользователей. Можно обратиться к руководству «Modern C++ and Windows Store Apps» для получения более подробной информации по данной теме.

Ошибка №4: Считать, что аргументы функции потока по умолчанию передаются по ссылке

Аргументы функции потока по умолчанию передаются по значению. Если вам необходимо внести изменения в передаваемые аргументы, необходимо передавать их по ссылке с помощью функции std::ref().

Под спойлером примеры из другой статьи C++11 Multithreading Tutorial via Q&A – Thread Management Basics (Deb Haldar), иллюстрирующие передачу параметров [прим. переводчика].

подробнее:

Ошибка №5: Не защищать разделяемые данные и ресурсы с помощью критической секции (например мьютексом)

В многопоточном окружении обычно более одного потока конкурируют за ресурсы и общие данные. Часто это приводит к неопределенному состоянию для ресурсов и данных, кроме случаев, когда доступ к ним защищен неким механизмом, позволяющим только одному потоку выполнения производить операции над ними в каждый момент времени.

В примере ниже *std::cout* является разделяемым ресурсом, с которым работают 6 потоков (t1-t5 + main).

```
 #include "stdafx.h"
#include <iostream>
#include <string>
#include <thread>
#include <mutex>

using namespace std;

std::mutex mu;

void CallHome(string message)
{
  cout << "Thread " << this_thread::get_id() << " says " << message << en
dl;
}

int main()
{
  thread t1(CallHome, "Hello from Jupiter");
  thread t2(CallHome, "Hello from Pluto");
  thread t3(CallHome, "Hello from Moon");

  CallHome("Hello from Main/Earth");

  thread t4(CallHome, "Hello from Uranus");
```

```
  thread t5(CallHome, "Hello from Neptune");

  t1.join();
  t2.join();
  t3.join();
  t4.join();
  t5.join();

  return 0;
}
```

Если мы выполним эту программу, то получим вывод:

```
 Thread 0x1000fb5c0 says Hello from Main/Earth
Thread Thread Thread 0x700005bd20000x700005b4f000 says says Thread Thread
Hello from Pluto0x700005c55000Hello from Jupiter says 0x700005d5b000Hello
from Moon
0x700005cd8000 says says Hello from Uranus

Hello from Neptune
```

Это происходит потому, что пять потоков одновременно обращаются к потоку вывода в произвольном порядке. Чтобы сделать вывод более определенным, необходимо защитить доступ к разделяемому ресурсу с помощью std::mutex. Просто изменим функцию *CallHome()* таким образом, чтобы она захватывала мьютекс перед использованием *std::cout* и освобождала его после.

```
 void CallHome(string message)
{
  mu.lock();
  cout << "Thread " << this_thread::get_id() << " says " << message << en
dl;
  mu.unlock();
}
```

Ошибка №6: Забыть освободить блокировку после выхода из критической секции

В предыдущем пункте вы видели как защитить критическую секцию с помощью мьютекса. Однако, вызов методов *lock()* и *unlock()* непосредственно у мьютекса не является предпочтительным вариантом потому, что вы можете забыть отдать удерживаемую блокировку. Что произойдет дальше? Все остальные потоки, которые ожидают освобождения ресурса, будут бесконечно заблокированы и программа может зависнуть.

В нашем синтетическом примере, если вы забыли разблокировать мьютекс в вызове функции *CallHome()*, в стандартный поток будет выведено первое сообщение из потока t1 и программа зависнет. Так происходит из-за того, что поток t1 получил блокировку мьютекса, а остальные потоки ждут освобождения этой блокировки.

```
 void CallHome(string message)
{
  mu.lock();
  cout << "Thread " << this_thread::get_id() << " says " << message << en
dl;
  //mu.unlock();  мы забыли освободить блокировку
}
```

Ниже приведен вывод данного кода– программа зависла, выведя единственное сообщение в терминал, и не завершается:

```
 Thread 0x700005986000 says Hello from Pluto
```

Подобные ошибки часто случаются, именно поэтому нежелательно использовать методы *lock()/unlock()* напрямую из мьютекса. Вместо этого следует использовать шаблонный класс std::lock_guard, который использует идиому RAII для управления временем жизни блокировки. Когда объект *lock_guard* создаётся, он пытается завладеть мьютексом. Когда программа выходит из области видимости *lock_guard* объекта, вызывается деструктор, который освобождает мьютекс.

Перепишем функцию *CallHome()* с применением *std::lock_guard* объекта:

```
 void CallHome(string message)
{
```

```
  std::lock_guard<std::mutex> lock(mu);  // пытаемся захватить блокировку
  cout << "Thread " << this_thread::get_id() << " says " << message << en
dl;
}// объект lock_guard уничтожится и освободит мьютекс
```

Ошибка №7: Делать размер критической секции больше, чем это необходимо

Когда один поток выполняется внутри критической секции все остальные, пытающиеся в нее войти, по сути заблокированы. Мы должны держать минимальное количество инструкций в критической секции, насколько это возможно. Для иллюстрации приведен пример плохого кода, имеющего большую критическую секцию:

```
 void CallHome(string message)
{
  std::lock_guard<std::mutex> lock(mu); // Начало критической секции, защ
ищаем доступ к std::cout

  ReadFifyThousandRecords();

  cout << "Thread " << this_thread::get_id() << " says " << message << en
dl;

}// при уничтожении объекта lock_guard блокировка на мьютекс mu освобожда
ется
```

Метод *ReadFifyThousandRecords()* не модифицирует данные. Нет никаких причин выполнять его под блокировкой. Если данный метод будет выполняться 10 секунд, считывая 50 тысяч строк из БД, все остальные потоки будут заблокированы на весь этот период без необходимости. Это может серьезно сказаться на производительности программы.

Правильным решением было бы держать в критической секции только работу с *std::cout*.

```
 void CallHome(string message)
{
  ReadFifyThousandRecords(); // Нет необходимости держать данный метод в
критической секции т.к. он не модифицирует данные
  std::lock_guard<std::mutex> lock(mu); // Начало критической секции, защ
ищаем доступ к std::cout
```

```
  cout << "Thread " << this_thread::get_id() << " says " << message << en
dl;

}//  при уничтожении объекта lock_guard блокировка на мьютекс mu освобожд
ается
```

Ошибка №8: Взятие нескольких блокировок в разном порядке

Это одна из наиболее распространенных причин взаимной блокировки (deadlock), ситуации, в которой потоки оказываются бесконечно заблокированы из-за ожидания получения доступа к ресурсам, заблокированным другими потоками. Рассмотрим пример:

поток 1	поток 2
lock A	lock B
//… какие-то операции	//… какие-то операции
lock B	lock A
//… какие-то еще операции	//… какие-то еще операции
unlock B	unlock A
unlock A	unlock B

Может возникнуть ситуация, в которой поток 1 попытается захватить блокировку B и окажется заблокированным, потому что поток 2 уже ее захватил. В тоже время, второй поток пытается захватить блокировку A, но не может этого сделать, потому что ее захватил первый поток. Поток 1 не может освободить блокировку A пока не захватит блокировку B и т.д. Другими словами, программа зависнет.

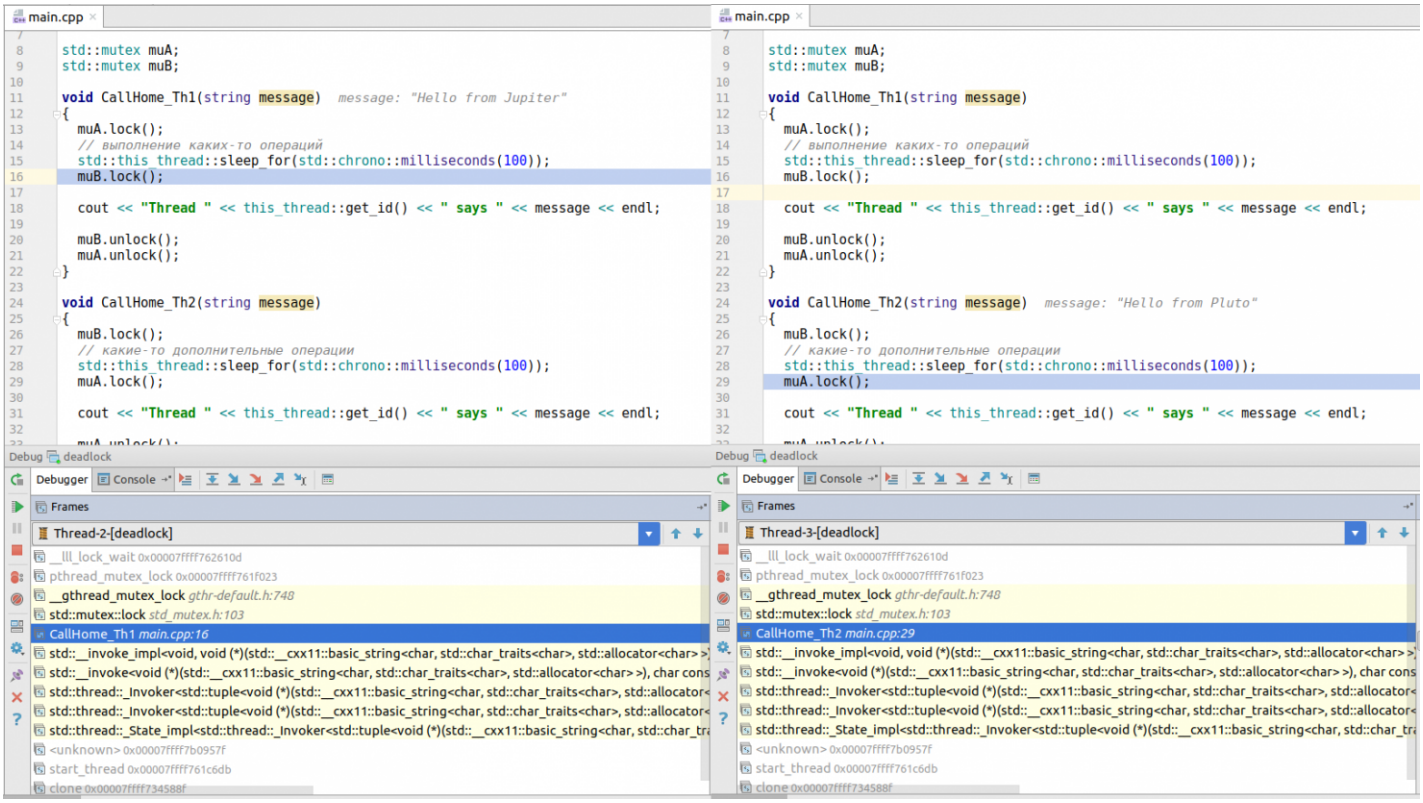
Данный пример кода поможет вам воспроизвести *deadlock*:

```
 #include "stdafx.h"
#include <iostream>
```

```
#include <string>
#include <thread>
#include <mutex>

using namespace std;

std::mutex muA;
std::mutex muB;

void CallHome_Th1(string message)
{
  muA.lock();
  // выполнение каких-то операций
  std::this_thread::sleep_for(std::chrono::milliseconds(100));
  muB.lock();

  cout << "Thread " << this_thread::get_id() << " says " << message << en
dl;

  muB.unlock();
  muA.unlock();
}

void CallHome_Th2(string message)
{
  muB.lock();
  // какие-то дополнительные операции
  std::this_thread::sleep_for(std::chrono::milliseconds(100));
  muA.lock();

  cout << "Thread " << this_thread::get_id() << " says " << message << en
dl;

  muA.unlock();
  muB.unlock();
}
```

```
int main()
{
  thread t1(CallHome_Th1, "Hello from Jupiter");
  thread t2(CallHome_Th2, "Hello from Pluto");

  t1.join();
  t2.join();

  return 0;
}
```

Если вы запустите этот код, он зависнет. Если залезть глубже в отладчик в окно потоков, вы увидите, что первый поток (вызванный из функции *CallHome_Th1()*) пытается получить блокировку мьютекса B, в то время как поток 2 (вызванный из *CallHome_Th2()*) пытается заблокировать мьютекс A. Никто из потоков не может достичь успеха, что и приводит к взаимной блокировке!



(картинка кликабельна)

Что вы можете с этим сделать? Лучшим решением было бы переструктурировать код таким образом, чтобы захват блокировок всякий раз происходил в одном и том же порядке.

В зависимости от ситуации можно воспользоваться другими стратегиями:

1. Использовать класс-обертку std::scoped_lock для совместного захвата нескольких блокировок:

```
std::scoped_lock lock{muA, muB};
```

2. Воспользоваться классом std::timed_mutex, в котором можно указать таймаут, по истечении которого блокировка будет снята, если ресурс не стал доступен.

```
std::timed_mutex m;

void DoSome(){
    std::chrono::milliseconds timeout(100);

    while(true){
        if(m.try_lock_for(timeout)){
            std::cout << std::this_thread::get_id() << ": acquire mutex s
uccessfully" << std::endl;
            m.unlock();
        } else {
            std::cout << std::this_thread::get_id() << ": can’t  acquire
mutex, do something else" << std::endl;
        }
    }
}
```

Ошибка №9: Пытаться дважды захватить блокировку std::mutex

Попытка дважды захватить блокировку приведет к неопределенному поведению. В большинстве отладочных реализаций это приведет к аварийному завершению. Например, в представленном ниже коде *LaunchRocket()* заблокирует мьютекс и после этого вызовет *StartThruster()*. Что любопытно, в приведенном коде вы не столкнетесь с этой проблемой при нормальной работе программы, проблема возникает только при выбросе исключения, который сопровождается неопределенным поведением или аварийным завершением программы.

```
#include "stdafx.h"
```

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mu;

static int counter = 0;

void StartThruster()
{
  try
  {
    // какие-то операции
  }
  catch (...)
  {
    std::lock_guard<std::mutex> lock(mu);
    std::cout << "Launching rocket" << std::endl;
  }
}

void LaunchRocket()
{
  std::lock_guard<std::mutex> lock(mu);
  counter++;
  StartThruster();
}

int main()
{
  std::thread t1(LaunchRocket);
  t1.join();
  return 0;
}
```

Для устранения этой проблемы необходимо исправить код таким образом, чтобы исключить

повторное взятие ранее полученных блокировок. В качестве костыльного решения можно использовать std::recursive_mutex, но такое решение практически всегда свидетельствует о плохой архитектуре программы.

Ошибка №10: Использовать мьютексы, когда достаточно std::atomic типов

Когда вам необходимо изменять простые типы данных, например, булево значение или целочисленный счетчик, использование std:atomic, как правило, даст лучшую производительность по сравнению с использованием мьютексов.

Например, вместо того чтобы использовать следующую конструкцию:

```
 int counter;
...
mu.lock();
counter++;
mu.unlock();
```

Лучше объявить переменную как *std::atomic*:

```
 std::atomic<int> counter;
...
counter++;
```

Для получения подробного сравнения mutex и atomic обратитесь к статье «Comparison: Lockless programming with atomics in C++ 11 vs. mutex and RW-locks»

Ошибка №11: Создавать и разрушать большое количество потоков напрямую, вместо использования пула свободных потоков

Создание и уничтожение потоков– дорогостоящие операции с точки зрения использования процессорного времени. Представим попытку создания потока в то время, как система производит ресурсоемкие вычислительные операции, например, отрисовку графики или вычисление игровой физики. Подход, часто используемый для подобных задач, заключается в создании пула предварительно выделенных потоков, которые могут обрабатывать рутинные задачи, такие как запись на диск или отправка данных по сети в течение всего жизненного цикла процесса.

Еще одно преимущество пула потоков, по сравнению с порождением и уничтожением потоков самостоятельно, заключается в том, что вам не нужно беспокоиться об thread oversubscription (ситуация, в которой количество потоков превышает количество доступных ядер и значительная часть процессорного времени тратится на переключение контекста [прим. переводчика]). Это может повлиять на производительность системы.

Кроме того, использование пула избавляет нас от мук управления жизненным циклом потоков, что в итоге выливается в более компактный код с меньшим количеством ошибок.

Две наиболее популярные библиотеки, реализующие пул потоков: Intel Thread Building Blocks(TBB) и Microsoft Parallel Patterns Library(PPL).

Ошибка №12: Не обрабатывать исключения, возникающие в фоновых потоках

Исключения, выброшенные в одном потоке, не могут быть обработаны в другом потоке. Давайте представим, что у нас есть функция которая выбрасывает исключение. Если мы выполним эту функцию в отдельном потоке, ответвлённом от основного потока выполнения, и ожидаем, что мы перехватим любое исключение, выброшенное из дополнительного потока, то это не сработает. Рассмотрим пример:

```
 #include "stdafx.h"
#include<iostream>
#include<thread>
#include<exception>
#include<stdexcept>

static std::exception_ptr teptr = nullptr;

void LaunchRocket()
{
  throw std::runtime_error("Catch me in MAIN");
}

int main()
{
  try
  {
```

```
    std::thread t1(LaunchRocket);
    t1.join();
  }
  catch (const std::exception &ex)
  {
    std::cout << "Thread exited with exception: " << ex.what() << "\n";
  }

  return 0;
}
```

При выполнении этой программы произойдет аварийное завершение, однако, catch блок в функции main() не выполнится и не обработает исключение, выброшенное в потоке t1.

Решение данной проблемы заключается в использовании возможности из C++11: std::exception_ptr применяется для обработки исключения, выброшенного в фоновом потоке. Вот шаги, которые необходимо предпринять:

- Создать глобальный экземпляр класса *std::exception_ptr*, инициализированный *nullptr*
- Внутри функции, которая выполняется в отдельном потоке, обрабатывать все исключения и устанавливать значение std::current_exception() глобальной переменной *std::exception_ptr*, объявленой на предыдущем шаге
- Внутри основного потока проверять значение глобальной переменной
- Если значение установлено, использовать функцию std::rethrow_exception(exception_ptr p) для повторного вызова пойманного ранее исключения, передав его по ссылке как параметр

Повторный вызов исключения по ссылке происходит не в том потоке, в котором оно было создано, таким образом данная возможность отлично подходит для обработки исключений в разных потоках.

В коде, представленном ниже, достигается безопасная обработка исключения, выброшенного в фоновом потоке.

```
#include "stdafx.h"
```

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

static std::exception_ptr globalExceptionPtr = nullptr;

void LaunchRocket()
{
  try
  {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    throw std::runtime_error("Catch me in MAIN");
  }
  catch (...)
  {
    //При возникновении исключения присваиваем значение указателю
    globalExceptionPtr = std::current_exception();
  }
}

int main()
{
  std::thread t1(LaunchRocket);
  t1.join();

  if (globalExceptionPtr)
  {
    try
    {
      std::rethrow_exception(globalExceptionPtr);
    }
    catch (const std::exception &ex)
    {
      std::cout << "Thread exited with exception: " << ex.what() << "\n";
    }
  }
```

```
  return 0;
}
```

Ошибка №13: Использовать потоки для симуляции асинхронной работы, вместо применения std::async

Если вам нужно, чтобы код выполнился асинхронно, т.е. без блокировки основного потока выполнения, наилучшим выбором будет использование std::async(). Это равносильно созданию потока и передаче необходимого кода на выполнение в этот поток через указатель на функцию или параметр в виде лямбда функции. Однако, в последнем случае вам необходимо следить за созданием, присоединением/отсоединением этого потока, а также за обработкой всех исключений, которые могут возникнуть в этом потоке. Если вы используете *std::async()*, вы избавляете себя от этих проблем, а также резко снижаете свои шансы попасть в *deadlock*.

Другое значительное преимущество использования *std::async* заключается в возможности получить результат выполнения асинхронной операции обратно в вызывающий поток с помощью std::future объекта. Представим, что у нас есть функция *ConjureMagic()*, которая возвращает int. Мы можем запустить асинхронную операцию, которая установит значение в будущем в *future* объект, когда выполнение задачи завершится, и мы сможем извлечь результат выполнения из этого объекта в том потоке выполнения, из которого операция была вызвана.

```
 // запуск асинхронной операции и получение обработчика для future
std::future asyncResult2 = std::async(&ConjureMagic);

//... выполнение каких-то операций пока future не будет установлено

// получение результата выполнения из future
 int v = asyncResult2.get();
```

Получение результата обратно из работающего потока в вызывающий более громоздко. Возможны два способа:

1. Передача ссылки на выходную переменную потоку, в которой он сохранит результат.

2. Хранить результат в переменной-поле объекта рабочего потока, которую можно будет считать как только поток завершит выполнение.

Kurt Guntheroth обнаружил, что с точки зрения производительности, накладные расходы на создание потока в 14 раз больше, чем использование *async*.

Итог: используйте *std::async()* по умолчанию, пока вы не найдете весомые аргументы в пользу использования непосредственно *std::thread*.

Ошибка №14: Не использовать std::launch::async если требуется асинхронность

Функция *std::async()* носит не совсем корректное название, потому что по умолчанию может не выполняться асинхронно!

Есть две политики выполнения *std::async*:

1. *std::launch::async*: переданная функция начинает выполняться незамедлительно в отдельном потоке
2. *std::launch::deferred*: переданная функция не запускается сразу же, ее запуск откладывается до того как будут произведены вызовы *get()* или *wait()* над *std::future* объектом, который будет возвращен из вызова *std::async*. В месте вызова этих методов, функция будет выполняться синхронно.

Когда мы вызываем *std::async()* с параметрами по умолчанию, происходит запуск с комбинацией этих двух параметров, что в действительности приводит к непредсказуемому поведению. Существует ряд других сложностей, связанных с использованием *std:async()* с политикой запуска по умолчанию:

- невозможность предсказать правильность доступа к локальным переменным потока
- асинхронная задача может и вовсе не запуститься из-за того, что вызовы методов *get()* и *wait()* могут не быть вызваны в течение выполнения программы
- при использовании в циклах, в которых условие выхода ожидает готовности std::future объекта, эти циклы могут никогда не завершиться, потому что std::future, возвращаемое вызовом std::async, может начаться в отложенном состоянии.

Для избежания всех этих сложностей **всегда** вызывайте *std::async* с политикой запуска *std::launch::async*.

Не делайте так:

```
 //
выполнение функции myFunction используя std::async с политикой запуска по
 умолчанию
auto myFuture = std::async(myFunction);
```

Вместо этого делайте так:

```
 //выполнение функции myFunction асинхронно
auto myFuture = std::async(std::launch::async, myFunction);
```

Более подробно этот момент рассмотрен в книге Скотта Мейерса «Эффективный и современный C++».

Ошибка №15: Вызывать метод get() у std::future объекта в блоке кода, время выполнение которого критично

Приведенный ниже код обрабатывает результат, полученный из *std::future* объекта асинхронной операции. Однако, цикл *while* будет заблокирован, пока асинхронная операция не выполнится (в данном случае на 10 секунд). Если вы хотите использовать данный цикл для вывода информации на экран, это может привести к неприятным задержкам отрисовки пользовательского интерфейса.

```
 #include "stdafx.h"
#include <future>
#include <iostream>

int main()
{
  std::future<int> myFuture = std::async(std::launch::async, []()
  {
    std::this_thread::sleep_for(std::chrono::seconds(10));
    return 8;
```

```
  });

  // Цикл обновления для выводимых данных
  while (true)
  {
    // вывод некоторой информации в терминал
    std::cout << "Rendering Data" << std::endl;
    int val = myFuture.get(); // вызов блокируется на 10 секунд
    // выполнение каких-то операций над Val
  }

  return 0;
}
```

Замечание: еще одна проблема приведенного выше кода в том, что он пытается обратиться к *std::future* объекту второй раз, хотя состояние *std::future* объекта было извлечено на первой итерации цикла и повторно не может быть получено.

Правильным решением было бы проверять валидность *std::future* объекта перед вызовом *get()* метода. Таким образом, мы не блокируем завершение асинхронного задания и не пытаемся повторно опросить уже извлеченный *std::future* объект.

Данный фрагмент кода позволяет достичь этого:

```
 #include "stdafx.h"
#include <future>
#include <iostream>

int main()
{
  std::future<int> myFuture = std::async(std::launch::async, []()
  {
    std::this_thread::sleep_for(std::chrono::seconds(10));
    return 8;
  });

  // Цикл обновления для выводимых данных
```

```
  while (true)
  {
    // вывод некоторой информации в терминал
    std::cout << "Rendering Data" << std::endl;

    if (myFuture.valid())
    {
      int val = myFuture.get(); // вызов блокируется на 10 секунд

      //  выполнение каких-то операций над Val
    }
  }

  return 0;
}
```

Ошибка №16: Непонимание того, что исключения, выброшенные внутри асинхронной операции, передадутся в вызывающий поток только при вызове std::future::get()

Представим что у нас есть следующий фрагмент кода, как вы думаете, каким будет результат вызова *std::future::get()*?

```
 #include "stdafx.h"
#include <future>
#include <iostream>

int main()
{
  std::future<int> myFuture = std::async(std::launch::async, []()
  {
    throw std::runtime_error("Catch me in MAIN");
    return 8;
  });

  if (myFuture.valid())
  {
```

```
    int result = myFuture.get();
  }

  return 0;
}
```

Если вы предположили что программа упадет– вы совершенно правы!

Исключение, выброшенное в асинхронной операции прокидывается только когда происходит вызов метода *get()* у *std::future объекта*. И если метод *get()* вызван не будет, то исключение будет проигнорировано и отброшено, когда *std::future* объект выйдет из области видимости.

Если ваша асинхронная операция может выбросить исключение, то необходимо всегда оборачивать вызов *std::future::get()* в *try/catch* блок. Пример как это может выглядеть:

```
 #include "stdafx.h"
#include <future>
#include <iostream>

int main()
{
  std::future<int> myFuture = std::async(std::launch::async, []()
  {
    throw std::runtime_error("Catch me in MAIN");
    return 8;
  });

  if (myFuture.valid())
  {
    try
    {
      int result = myFuture.get();
    }
    catch (const std::runtime_error& e)
    {
       std::cout << "Async task threw exception: " << e.what() << std::en
dl;
```

```
    }
  }
  return 0;
}
```

Ошибка №17: Использование std::async, когда требуется чёткий контроль над исполнением потока

Хотя *std::async()* достаточно в большинстве случаев, бывают ситуации, в которых вам может потребоваться тщательный контроль над выполнением вашего кода в потоке. Например, если вы хотите привязать определенный поток к конкретному ядру процессора в многопроцессорной системе (например Xbox).

Приведенный фрагмент кода устанавливает привязку потока к 5-му процессорному ядру в системе.

```
 #include "stdafx.h"
#include <windows.h>
#include <iostream>
#include <thread>

using namespace std;

void LaunchRocket()
{
  cout << "Launching Rocket" << endl;
}

int main()
{
  thread t1(LaunchRocket);

  DWORD result = ::SetThreadIdealProcessor(t1.native_handle(), 5);

  t1.join();

  return 0;
```

```
}
```

Это возможно благодаря методу native_handle() объекта *std::thread*, и передаче его в потоковую функцию Win32 API. Существует множество других возможностей, предоставляемых через потоковое Win32 API, которые не доступны в *std::thread* или *std::async()*. При работе через *std::async()* эти базовые функции платформы недоступны, что и делает этот способ непригодным для более сложных задач.

Альтернативный вариант— создать std::packaged_task и переместить его в нужный поток выполнения после установки свойств потока.

Ошибка №18: Создавать намного больше «выполняющихся» потоков, чем доступно ядер

С точки зрения архитектуры потоки можно классифицировать на две группы: «выполняющиеся» и «ожидающие».

Выполняющиеся потоки утилизируют 100% процессорного времени ядра на котором работают. Когда более одного выполняющегося потока выделено на одно ядро, эффективность утилизации процессорного времени падает. Мы не получаем выигрыша в производительности, если выполняем более одного выполняющегося потока на одном процессорном ядре– в действительности производительность падает из-за дополнительных переключений контекста.

Ожидающие потоки утилизируют только несколько тактов ядра на котором выполняются пока ожидают системных событий или сетевого ввода-вывода и т.д. При этом бОльшая часть доступного процессорного времени ядра остается не использована. Один ожидающий поток может обрабатывать данные, в то время как остальные ожидают срабатывания событий– вот почему выгодно распределять несколько ожидающих потоков на одно ядро. Планирование нескольких ожидающих потоков на одно ядро может обеспечить гораздо большую производительность программы.

Итак, как понять какое количество выполняющихся потоков поддерживает система? Используйте метод std::thread::hardware_concurrency(). Эта функция обычно возвращает количество ядер процессора, но при этом учитывает ядра, которые ведут себя как два или более логических ядер из-за гипертрединга.

Необходимо использовать полученное значение целевой платформы для планирования

максимального количества одновременно выполняющихся потоков вашей программы. Вы также можете назначить одно ядро для всех ожидающих потоков, и использовать оставшееся количество ядер для выполняющихся потоков. Например, в четырехъядерной системе используйте одно ядро для ВСЕХ ожидающих потоков, а для остальных трех ядер — три выполняющихся потока. В зависимости от эффективности вашего планировщика потоков, некоторые из ваших исполняемых потоков могут переключать контекст (из-за сбоев доступа к страницам и т.д.), оставляя ядро бездействующим в течение некоторого времени. Если вы наблюдаете эту ситуацию во время профилирования, вам следует создать чуть большее количество выполняемых потоков, чем количество ядер, и настроить эту величину для своей системы.

Ошибка №19: Использование ключевого слова volatile для синхронизации

Ключевое слово volatile перед указанием типа переменной не делает операции с этой переменной атомарными или потокобезопасными. То, что вы, вероятно, хотите, это *std::atomic*.

Посмотрите обсуждение на stackoverflow для получения подробностей.

Ошибка №20: Использование Lock Free архитектуры, кроме случаев когда это совершенно необходимо

В сложности есть что-то, что нравится каждому инженеру. Создание программ, работающих без блокировок (lock free), звучит очень соблазнительно по сравнению с обычными механизмами синхронизации, такими как мьютекс, условные переменные, асинхронность и т. д. Однако, каждый опытный разработчик C ++, с которым я говорил, придерживался мнения, что применение программирования без блокировок в качестве исходного варианта является видом преждевременной оптимизации, которая может выйти боком в самый неподходящий момент (подумайте о сбое в эксплуатируемой системе, когда у вас нет полного дампа кучи!).

В моей карьере в C ++ была только одна ситуация, для которой требовалось выполнение кода без блокировок, потому что мы работали в системе с ограниченными ресурсами, где каждая транзакция в нашем компоненте должна была занимать не более 10 микросекунд.

Перед тем как задумываться над применением подхода разработки без блокировок, пожалуйста ответьте на три вопроса:

- Пробовали ли вы спроектировать архитектуру вашей системы таким образом, чтобы она не нуждалась в механизме синхронизации? Как правило, лучшая синхронизация– отсутствие синхронизации.
- Если вам нужна синхронизация, профилировали ли вы свой код для понимания характеристик производительности? Если да, пытались ли вы оптимизировать узкие места?
- Можете ли вы горизонтально масштабироваться вместо того чтобы масштабироваться вертикально?

Резюмируя, для обычной разработки приложений, пожалуйста, рассматривайте программирование без блокировки только тогда, когда вы исчерпали все другие альтернативы. Еще один способ взглянуть на это заключается в том, что если вы все еще делаете некоторые из вышеупомянутых 19 ошибок, вам, вероятно, следует держаться подальше от программирования без блокировок.

[От. переводчика: огромное спасибо пользователю @vovo4K за помощь в подготовке данной статьи.]

Теги: перевод, многопоточное программирование, threads, c++

Хабы: Программирование, C++

17 Карма 0 Рейтинг

@Pride_St
Пользователь

Комментарии 94

ПОХОЖИЕ ПУБЛИКАЦИИ

11 августа 2021 в 11:58

Про многопоточность 1. Thread

+11 17K 78 2 +2

3 июня 2018 в 16:28

Изучаем многопоточное программирование в Go по картинкам

+46 51K 341 21 +21

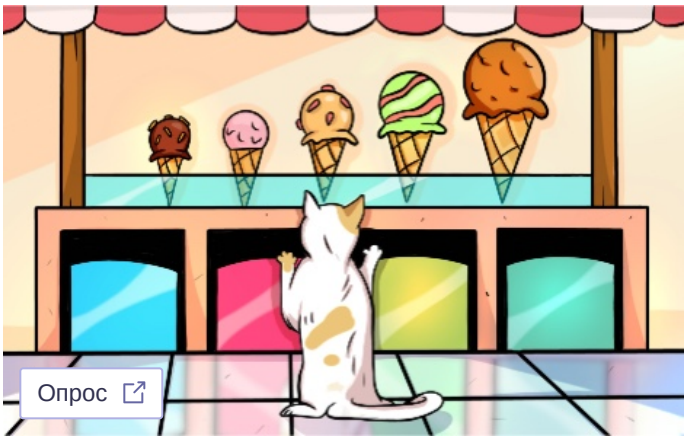
7 декабря 2017 в 03:34

Многопоточное программирование в Android с использованием RxJava 2

+14 36K 154 0

МИНУТОЧКУ ВНИМАНИЯ

Разместить



Этот маленький — но по пять: опрос по хабраблогам

Как настроить виртуальный дата-центр, чтобы ускорить разработку

вчера в 19:17

Старое железо СССР часть 6. Калининград. (Кенигсберг). Как ни странно, про практическое применение СМ ЭВМ

+18 4.1K 9 20 +20

Магия геоданных: подкаст о том, что они могут и как их заклинать

Подкаст

ЧИТАЮТ СЕЙЧАС

Ученые нашли упражнение, по

97K 175 +175

Зачем тебе, бабка, тестовое

25K 235 +235

Секретный прогноз IT-экосист у готовиться

11K 25 +25

В Москве до конца 2022 года з

972 5 +5

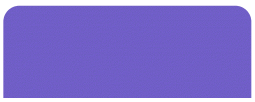
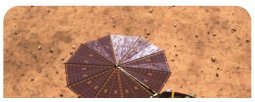
Тест самых дешёвых щелочны

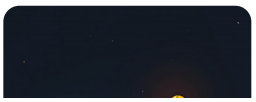
2K 4 +4

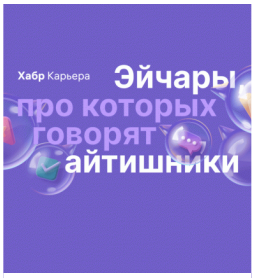
Какие они — новаторы Китая

Турбо

ИСТОРИИ

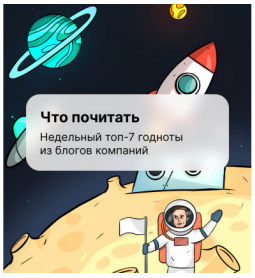





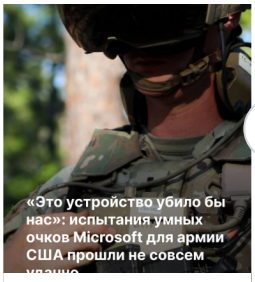
Эйчары, про которых говорят айтишники

InSight попал в песчаную бурю на Марсе

Топ-5 докладов HighLoad++ 2022 открыты для Хабра



Недельный топ годноты от компаний



Испытания умных очков Microsoft для армии США

НАСА восста ориент Capsto

РАБОТА

QT разработчик
10 вакансий

Программист C++
93 вакансии

Все вакансии

Техническая поддержка

Вернуться на старую версию