目录

[**std::this\_thread::yield** 3](#_Toc503528560)

[**std::this\_thread::get\_id** 4](#_Toc503528561)

[**std::this\_thread::sleep\_for** 5](#_Toc503528562)

[**std::mutex** 6](#_Toc503528563)

[**std::lock\_guard** 8](#_Toc503528564)

[**std::lock** 9](#_Toc503528565)

[**Mutex和Critical\_Section** 12](#_Toc503528566)

线程支持库

C++包含对线程，互斥，条件变量和期货的内置支持。

线程：

线程使程序能够在多核处理器上执行。

头文件： <thread>

thread(C++11) 管理一个单独的线程

管理线程的函数

sleep\_until（C++11）

|  |  |  |  |
| --- | --- | --- | --- |
| 成员 | 标准 | 类型 | 说明 |
| thread | C++11 | 类 | 管理一个独立线程 |
| [yield](#_std::this_thread::yield) | C++11 | 函数 | 向线程的重新调度执行的实施提供线索，以允许别的线程执行 |
| [get\_id](#_std::this_thread::get_id) | C++11 | 函数 | 返回当前进程的id |
| sleep\_for | C++11 | 函数 | 在一个特定的时间段内停止当前进程的执行 |
| sleep\_until | C++11 | 函数 | 在特定的时间点之前停止当前进程的执行 |
| Cache size access，头文件:<new> | | | |
| hardware\_destructive\_interference\_size | C++17 |  |  |
| hardware\_constructive\_interference\_size | C++17 |  |  |
| 互斥  互斥算法阻止多个线程同时访问共享资源。这会避免数据竞态并提供线程间的同步。 | | | |
| 头文件：<mutex> | | | |
| mutex | C++11 |  |  |
| timed\_mutex | C++11 |  |  |
| recursive\_mutex | C++11 |  |  |
| recursive\_timed\_mutex | C++11 |  |  |
| 头文件：<shared\_mutex> | | | |
| shared\_mutex | C++17 |  |  |
| shared\_timed\_mutex | C++14 |  |  |
| 通用互斥管理，头文件：<mutex> | | | |
| lock\_guard | C++11 |  |  |
| scoped\_lock | C++17 |  |  |
| unique\_lock | C++11 |  |  |
| shared\_lock | C++14 |  |  |
| defer\_lock\_t | C++11 |  |  |
| try\_to\_lock\_t | C++11 |  |  |
| adopt\_lock\_t | C++11 |  |  |
| defer\_lock | C++11 |  |  |
| try\_to\_lock | C++11 |  |  |
| adopt\_lock | C++11 |  |  |
| 通用加锁算法，头文件：<mutex> | | | |
| try\_lock | C++11 |  |  |
| lock | C++11 |  |  |
| Call once | | | |
| once\_flag | C++11 |  |  |
| call\_once | C++11 |  |  |
| 条件变量，头文件<condition\_variable>  条件变量是允许多个线程通信同步原语。它允许其它线程等待通知（或者可能等待一段时间）。条件变量总是与一个互斥锁相关联。 | | | |
| condition\_variable | C++11 |  |  |
| condition\_variable\_any | C++11 |  |  |
| notify\_all\_at\_thread\_exit | C++11 |  |  |
| cv\_status | C++11 |  |  |

# **std::this\_thread::yield**

1．头文件：<thread>

void yield() noexcept;（C++11后）

向线程的重新调度执行的实施提供线索，以允许别的线程执行。

2.参数（无）

3.返回值（无）

4.注意

这个函数的真实行为依赖于具体的实现，尤其是OS调度器的机理和系统的状态。例如，对于先进先出实时调度（SCHED\_FIFO-Linux）将会挂起当前的线程，把它放到相同优先级线程队列中，等待执行。（如果没有相同优先级的其它线程，yield函数将不会有影响）

5.例程，代码如下

#include <iostream>

#include <chrono>

#include <thread>

// "busy sleep" while suggesting that other threads run

// for a small amount of time

void little\_sleep(std::chrono::microseconds us)

{

auto start = std::chrono::high\_resolution\_clock::now();

auto end = start + us;

do {

std::this\_thread::yield();

} while (std::chrono::high\_resolution\_clock::now() < end);

}

int main()

{

auto start = std::chrono::high\_resolution\_clock::now();

little\_sleep(std::chrono::microseconds(100));

auto elapsed = std::chrono::high\_resolution\_clock::now() - start;

std::cout << "waited for "

<< std::chrono::duration\_cast<std::chrono::microseconds>(elapsed).count()

<< " microseconds\n";

}

# **std::this\_thread::get\_id**

1. 函数原型及定义

std::thread::id get\_id() noexcept;（C++11后）

声明位于头文件<thread>。

2. 参数（无）

3. 返回值（无）

当前线程的ID。

4. 例程：

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

std::mutex g\_display\_mutex;

void foo()

{

std::thread::id this\_id = std::this\_thread::get\_id();

g\_display\_mutex.lock();

std::cout << "thread [" << this\_id << "] sleeping...\n";

g\_display\_mutex.unlock();

std::this\_thread::sleep\_for(std::chrono::seconds(1));

}

int main()

{

std::thread t1(foo);

std::thread t2(foo);

t1.join();

t2.join();

}

# **std::this\_thread::sleep\_for**

1. 函数原型及定义

template< class Rep, class Period >

void sleep\_for( const std::chrono::duration<Rep, Period>& sleep\_duration )

在时间间隔内阻塞当前进程的执行。由于调度资源竞争延迟，该函数可能阻塞进程的时间大于sleep\_duration时间段。建议使用稳定的时钟来测量持续时间，如果使用系统时钟代替，则等待时间对时钟的调整非常敏感。

2. 参数

sleep\_duration - 睡眠的时间周期

3. 返回值

无

4. 异常

时钟，时间点或执行周期内抛出的任何异常。（由标准库提供的时钟，时间点，周期内不会抛出）

5. 例程

#include <iostream>

#include <chrono>

#include <thread>

int main()

{

// using namespace std::chrono\_literals;

std::cout << "Hello waiter" << std::endl; // flush is intentional

auto start = std::chrono::high\_resolution\_clock::now();

std::this\_thread::sleep\_for(std::chrono::seconds(2));

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double, std::milli> elapsed = end-start;

std::cout << "Waited " << elapsed.count() << " ms\n";

}

# **std::mutex**

头文件<mutex>

class mutex; (C++11后)

mutex类是一个同步原语，可以保护共享数据不被多个线程同时访问。

mutex提供独有的，非递归所有权语义：

* 调用线程拥有一个mutex的时间从调用lock或者try\_lock开始，直到调用unlock释放为止。
* 当线程拥有一个互斥量mutex时，所有其它线程获取该mutex的所有权时，则会发生阻塞（调用lock）或者返回false（调用try\_lock）。
* 调用线程在调用lock或try\_lock之前，不能拥有互斥量mutex。

如果互斥量mutex任然被任一个线程拥有，mutex就被销毁或者拥有mutex的线程被销毁，程序的行为是不确定的。互斥量mutex类满足Mutex和StandardLayoutType的所有要求。

std :: mutex既不可复制也不可移动。

成员类型

|  |  |
| --- | --- |
| 成员类型 | 定义 |
| native\_handle\_type(optional) |  |
| constructor | 构建互斥量mutex（public） |
| destructor | 销毁mutex（public） |
| operator=[deleted] | 不可复制分配（public） |
| lock | 给mutex加锁，如果mutex不可用，则阻塞 |
| try\_lock | 给mutex加锁，如果mutex不可用，则返回false |
| unlock | 解锁mutex |
| native\_handle | 返回底层实现定义的线程句柄 |

注意：

通常不会直接访问std::mutex。std::unique\_lock，std::lock\_guard或std::scoped\_lock（C ++ 17后）以更加异常安全的方式管理锁定。

举例：

这个例子展示了如何使用一个互斥体来保护在两个线程之间共享的std :: map。代码如下：

#include <iostream>

#include <map>

#include <string>

#include <chrono>

#include <thread>

#include <mutex>

std::map<std::string, std::string> g\_pages;

std::mutex g\_pages\_mutex;

void save\_page(const std::string &url)

{

// simulate a long page fetch

std::this\_thread::sleep\_for(std::chrono::seconds(2));

std::string result = "fake content";

std::lock\_guard<std::mutex> guard(g\_pages\_mutex);

g\_pages[url] = result;

}

int main()

{

std::thread t1(save\_page, "http://foo");

std::thread t2(save\_page, "http://bar");

t1.join();

t2.join();

// safe to access g\_pages without lock now, as the threads are joined

for (const auto &pair : g\_pages) {

std::cout << pair.first << " => " << pair.second << '\n';

}

}

# **std::lock\_guard**

1. 原型

头文件<mutex>

template< class Mutex >

class lock\_guard;

类lock\_guard是一个互斥锁的封装，它提供了一个方便的RAII风格的机制，用于在作用域块持续时间内拥有一个互斥锁。

当一个lock\_guard对象被创建时，它会尝试获取给它的互斥锁的所有权。当离开lock\_guard对象被创建的作用域时，lock\_guard被销毁并释放互斥锁。

lock\_guard类不可复制。

2.模板参数：

Mutex 互斥锁的类型。该类型必须满足BasicLockable要求。

3.定义

class lock\_guard{

private:

mutex\_type Mutex;

public:

lock\_guard();

~lock\_guard();

operator=[deleted]

};

4.例程

#include <thread>

#include <mutex>

#include <iostream>

int g\_i = 0;

std::mutex g\_i\_mutex; // protects g\_i

void safe\_increment()

{

std::lock\_guard<std::mutex> lock(g\_i\_mutex);

++g\_i;

std::cout << std::this\_thread::get\_id() << ": " << g\_i << '\n';

// g\_i\_mutex is automatically released when lock

// goes out of scope

}

int main()

{

std::cout << "main: " << g\_i << '\n';

std::thread t1(safe\_increment);

std::thread t2(safe\_increment);

t1.join();

t2.join();

std::cout << "main: " << g\_i << '\n';

}

# **std::lock**

1.原型

头文件<mutex>

template< class Lockable1, class Lockable2, class... LockableN >

void lock( Lockable1& lock1, Lockable2& lock2, LockableN&... lockn );

（C++11后）

锁定给定的可以加锁的对象，lock1，lock2，…lockn，使用了死锁避免算法避免发生死锁。这些对象被lock，try\_lock，unlock调用而锁定后，此时如果调用lock或者unlock导致异常，在异常抛出之前，对于任何锁定的对象，unlock会被调用。

2. 参数

lock1, lock2,… ,lockn：加锁对象

3. 返回值（无）

4．注意

Boost提供了这个函数的一个版本，它接受由一对迭代器定义的一系列lockable对象。

5. 例程

以下示例使用std :: lock来锁定互斥锁对，而不发生死锁。

#include <mutex>

#include <thread>

#include <iostream>

#include <vector>

#include <functional>

#include <chrono>

#include <string>

struct Employee

{

Employee(std::string id) : id(id) {}

std::string id;

std::vector<std::string> lunch\_partners;

std::mutex m;

std::string output() const

{

std::string ret = "Employee " + id + " has lunch partners: ";

for( const auto& partner : lunch\_partners )

ret += partner + " ";

return ret;

}

};

void send\_mail(Employee &, Employee &)

{

// simulate a time-consuming messaging operation

std::this\_thread::sleep\_for(std::chrono::seconds(1));

}

void assign\_lunch\_partner(Employee &e1, Employee &e2)

{

static std::mutex io\_mutex;

{

std::lock\_guard<std::mutex> lk(io\_mutex);

std::cout << e1.id << " and " << e2.id << " are waiting for locks" << std::endl;

}

// use std::lock to acquire two locks without worrying about

// other calls to assign\_lunch\_partner deadlocking us

{

std::lock(e1.m, e2.m);

std::lock\_guard<std::mutex> lk1(e1.m, std::adopt\_lock);

std::lock\_guard<std::mutex> lk2(e2.m, std::adopt\_lock);

// Equivalent code (if unique\_locks are needed, e.g. for condition variables)

// std::unique\_lock<std::mutex> lk1(e1.m, std::defer\_lock);

// std::unique\_lock<std::mutex> lk2(e2.m, std::defer\_lock);

// std::lock(lk1, lk2);

{

std::lock\_guard<std::mutex> lk(io\_mutex);

std::cout << e1.id << " and " << e2.id << " got locks" << std::endl;

}

e1.lunch\_partners.push\_back(e2.id);

e2.lunch\_partners.push\_back(e1.id);

}

send\_mail(e1, e2);

send\_mail(e2, e1);

}

int main()

{

Employee alice("alice"), bob("bob"), christina("christina"), dave("dave");

// assign in parallel threads because mailing users about lunch assignments

// takes a long time

std::vector<std::thread> threads;

threads.emplace\_back(assign\_lunch\_partner, std::ref(alice), std::ref(bob));

threads.emplace\_back(assign\_lunch\_partner, std::ref(christina), std::ref(bob));

threads.emplace\_back(assign\_lunch\_partner, std::ref(christina), std::ref(alice));

threads.emplace\_back(assign\_lunch\_partner, std::ref(dave), std::ref(bob));

for (auto &thread : threads) thread.join();

std::cout << alice.output() << '\n' << bob.output() << '\n'

<< christina.output() << '\n' << dave.output() << '\n';

}

std::condition\_variable

1. 类定义

头文件：<condition\_variable>

class condition\_variable; （C++11）

condition\_variable类是一个同步原语，常被用来阻塞一个线程或者多个线程，直到另一个线程修改共享变量（条件），并通知condition\_variable。

打算修改变量的线程必须拥有以下条件：

（1）获取一个互斥锁std::mutex（通常使用std::lock\_guard）

# **Mutex和Critical\_Section**

Mutex和Critical Section都是主要用于限制多线程（Multithread）对全局或共享的变量、对象或内存空间的访问。下面是其主要的异同点（不同的地方用红色表示）。

|  |  |  |
| --- | --- | --- |
|  | Mutex | Critical Section |
| 性能和速度 | 慢。  Mutex是内核对象，相关函数的执行 （WaitForSingleObject，ReleaseMutex）需要用户模式（User Mode）到内核模式（Kernel Mode）的转换，在x86处理器上这种转化一般要花费600个左右的CPU指令周期。 | 快。  Critical Section本身不是内核对象，相关函数（EnterCriticalSection，LeaveCriticalSection）的调用一般都在用户模式内执行，在x86处理器上一般只需要发费9个左右的CPU指令周期。只有当想要获得的锁正好被别的线程拥有时才会退化成和Mutex一样，即转换到内核模式，花费600个左右的 CPU指令周期。 |
| 能否跨越进程（Process）边界 | 可以 | 不可 |
| 定义写法 | HANDLE hmtx; | CRITICAL\_SECTION cs; |
| 初始化写法 | hmtx= CreateMutex (NULL, FALSE, NULL); | InitializeCriticalSection(&cs); |
| 结束清除写法 | CloseHandle(hmtx); | DeleteCriticalSection(&cs); |
| 无限期等待的写法 | WaitForSingleObject (hmtx, INFINITE); | EnterCriticalSection(&cs); |
| 0等待（状态检测）的写法 | WaitForSingleObject(hmtx,0); | TryEnterCriticalSection(&cs); |
| 任意时间等待的写法 | WaitForSingleObject (hmtx, dwMilliseconds); | 不支持 |
| 锁释放的写法 | ReleaseMutex(hmtx); | LeaveCriticalSection(&cs); |
| 能否被一道用于等待其他内核对象 | 可以（使用  WaitForMultipleObjects，  WaitForMultipleObjectsEx，  MsgWaitForMultipleObjects，  MsgWaitForMultipleObjectsEx等等） | 不可 |
| 当拥有锁的线程死亡时 | Mutex变成abandoned状态，其他的等待线程可以获得锁。 | Critical Section的状态不可知（undefined），  以后的动作就不能保证了。 |
| 自己会不会锁住自己 | 不会（对已获得的Mutex,重复调用WaitForSingleObject不会锁住自己。但最后你别忘了要调用同样次数的ReleaseMutex） | 不会（对已获得的Critical Section,重复调用EnterCriticalSection不会锁住自己。但最后你别忘了要调用同样次数的  LeaveCriticalSection） |

下面是一些补充：

1.请先检查你的设计，把不必要的全局或共享对象改为局部对象。全局的东西越少，出问题的可能就越小。

2.每次你使用EnterCriticalSection时，请不要忘了在函数的所有可能返回的地方都加上LeaveCriticalSection。对于Mutex也同样。若你把这个问题和Win32 structured exception或C++ exception一起考虑，你会发现问题并不是那么简单。自定义一个封装类可能是一种解决方案，以Critical Section为例的代码如下所示：

class csholder

{

CRITICAL\_SECTION \*cs;

public:

csholder(CRITICAL\_SECTION \*c): cs(c)

{ EnterCriticalSection(cs); }

~csholder() { LeaveCriticalSection(cs); }

};

CRITICAL\_SECTION some\_cs;

void foo()

{

// ...

csholder hold\_some(&some\_cs);

// ... CS protected code here

// at return or if an exception happens

// hold\_some's destructor is automatically called

}

1.根据你的互斥范围需求的不同，把Mutex或Critical Section定义为类的成员变量，或者静态类变量。

2.若你想限制访问的全局变量只有一个而且类型比较简单（比如是LONG或PVOID型），你也可以使用InterlockedXXX系列函数来保证一个线程写多个线程读。

Demo程序

#include <Windows.h>

#include <iostream>

using namespace std;

int index = 0;

// 临界区结构对象

CRITICAL\_SECTION g\_cs;

HANDLE hMutex = NULL;

void changeMe()

{

cout << index++ << endl;

}

void changeMe2()

{

cout << index++ << endl;

}

void changeMe3()

{

cout << index++ << endl;

}

DWORD WINAPI th1(LPVOID lpParameter)

{

while(1)

{

Sleep(1600); //sleep 1.6 s

// 进入临界区

EnterCriticalSection(&g\_cs);

// 等待互斥对象通知

//WaitForSingleObject(hMutex, INFINITE);

// 对共享资源进行写入操作

//cout << "a" << index++ << endl;

changeMe();

changeMe2();

changeMe3();

// 释放互斥对象

//ReleaseMutex(hMutex);

// 离开临界区

LeaveCriticalSection(&g\_cs);

}

return 0;

}

DWORD WINAPI th2(LPVOID lpParameter)

{

while(1)

{

Sleep(2000); //sleep 2 s

// 进入临界区

EnterCriticalSection(&g\_cs);

// 等待互斥对象通知

//WaitForSingleObject(hMutex, INFINITE);

//cout << "b" << index++ << endl;

changeMe();

changeMe2();

changeMe3();

// 释放互斥对象

//ReleaseMutex(hMutex);

// 离开临界区

LeaveCriticalSection(&g\_cs);

}

return 0;

}

int main(int argc, char\* argv[])

{

// 创建互斥对象

//hMutex = CreateMutex(NULL, TRUE, NULL);

// 初始化临界区

InitializeCriticalSection(&g\_cs);

HANDLE hThread1;

HANDLE hThread2;

hThread1 = CreateThread(NULL, 0, th1, NULL, 0, NULL);

hThread2 = CreateThread(NULL, 0, th2, NULL, 0, NULL);

int k;

cin >> k;

printf("Hello World!\n");

return 0;

}