## 基于C++中互斥锁和条件变量的同步互斥

### 互斥锁/互斥量mutex及程序示例

C++中的互斥锁属于教科书中的初值为1的二元互斥信号量，用于控制对资源、共享变量的互斥访问，包括：

（1）std::mutex：最基本的互斥量，不支持递归地对std::mutex 对象上锁；

（2）std::recursive\_lock：允许递归地对互斥量对象上锁。

对互斥锁的操作有lock()、unlock()、try\_lock()。

成员函数

* + 构造函数

std::mutex不允许拷贝构造和move拷贝，刚创建的mutex对象处于unlocked状态。

* + lock()

调用lock()的线程将锁住互斥锁，线程调用该函数会发生3 种情况：

(i)如果该互斥量当前没有被锁住，表示互斥锁对应的共享资源空闲，则调用线程将该互斥量锁住，拥有该锁，直到调用unlock去释放锁。

(ii)如果互斥量已经被其它线程锁住，表示其它线程已经占有共享资源，则当前的调用线程被阻塞。

(iii)如果当前互斥量被当前调用线程锁住，将产生死锁。

* + unlock()

解锁，释放对互斥量的所有权，即释放对共享资源的控制权。

* + try\_lock()

尝试锁住互斥量，如果互斥量已经被其它线程占有，当前线程**不会**被阻塞。线程调用该函数会发生3 种情况：

(i)如果该互斥量当前没有被锁住，表示互斥锁对应的共享资源空闲，则调用线程将该互斥量锁住，拥有该锁，直到调用unlock去释放锁。

(ii)如果互斥量已经被其它线程锁住，表示其它线程已经占有共享资源，则当前的调用线程返回false，并不会被阻塞。

(iii)如果当前互斥量被当前调用线程锁住，将产生死锁。

* + lock\_guard

在lock\_guard对象构造时，传入的Mutex对象（即它所管理的Mutex对象）将被当前线程锁住。

在lock\_guard对象被析构时，它所管理的Mutex对象会自动解锁。

由于不需要程序显示调用lock和unlock对Mutex进行上锁和解锁操作，因此是最简单安全的上锁和解锁方式，尤其是在程序抛出异常后先前已被上锁的Mutex对象可以正确进行解锁操作，大大简化了编写与Mutex 相关的异常处理代码。

**程序示例1：**

#include <iostream>

#include <mutex>

#include <thread>

using namespace std;

volatile int counter(0); // non-atomic counter

//volatile 表示不进行优化 多次循环不把i保存在寄存器，每次从内存获取。

mutex **mtx**; // locks access to counter

void increase10Ktime()

{

for (int i = 0; i < 10000; i++)

{

**mtx.lock()**;

counter++;

**mtx.unlock()**;

}

}

int main()

{

thread ths[10];

for (int i = 0; i < 10; i++)

{

ths[i] = thread(increase10Ktime);

}

for (auto& th: ths)

th.join();

cout << "after successful increase :" << counter << endl;

return 0;

}

运行结果：

运行结果

**程序示例2：**

#include <iostream>

#include <mutex>

#include <thread>

using namespace std;

volatile int counter(0); // non-atomic counter

//volatile 表示不进行优化 多次循环不把i保存在寄存器，每次从内存获取。

mutex mtx; // locks access to counter

void increase10Ktime()

{

for (int i = 0; i < 10000; i++)

{

counter++;

}

}

int main()

{

thread ths[10];

for (int i = 0; i < 10; i++)

{

ths[i] = thread(increase10Ktime);

}

for (auto& th : ths)

th.join();

cout << "after successful increase :" << counter << endl;

return 0;

}

运行结果：

运行结果

**程序示例3：try\_lock/unclok**

#include <iostream>

#include <mutex>

#include <thread>

using namespace std;

volatile int counter(0); // non-atomic counter

mutex mtx; // locks access to counter

void increase10Ktime()

{

for (int i = 0; i < 10000; i++)

{

while (!mtx.try\_lock()); //没有锁死等

counter++;

mtx.unlock();

}

}

int main()

{

thread ths[10];

for (int i = 0; i < 10; i++)

{

ths[i] = thread(increase10Ktime);

}

for (auto& th : ths)

th.join();

cout << "after successful increase :" << counter << endl;

return 0;

}

运行结果：

运行结果

**程序示例4**：lock\_guard

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

mutex mtx;

void printEven(int i)

{

if (i % 2 == 0)

cout << i << " is even" << endl;

else

throw logic\_error("not even"); //抛出异常

}

void printThreadId(int id)

{

try {

//RAII： mtx资源获取即初始化。

lock\_guard<mutex> lck(mtx); //栈自旋抛出异常时栈对象自我析构。

printEven(id);

}

catch (logic\_error&) {

cout << "exception caught" << endl;

}

//离开当前作用于自动释放mtx锁。

}

int main()

{

thread ths[10]; //spawn 10 threads

for (int i = 0; i < 10; i++)

{

ths[i] = thread(printThreadId, i + 1);

}

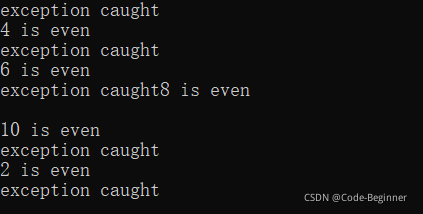
for (auto& th : ths)

th.join();

return 0;

}

运行结果：



可以看到：**当前线程抛出异常不影响其他线程。**

**程序示例5：**

与程序示例4对比，不使用lock\_guard<mutex>

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

mutex mtx;

void printEven(int i)

{

if (i % 2 == 0)

cout << i << " is even" << endl;

else

throw logic\_error("not even"); //抛出异常

}

void printThreadId(int id)

{

try {

mtx.lock();

printEven(id);

mtx.unlock();

}

catch (logic\_error&) {

cout << "exception caught" << endl;

}

}

int main()

{

thread ths[10]; //spawn 10 threads

for (int i = 0; i < 10; i++)

{

ths[i] = thread(printThreadId, i + 1);

}

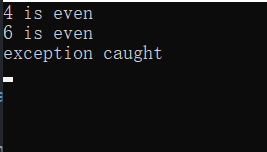
for (auto& th : ths)

th.join();

return 0;

}

运行结果：



抛出异常之后，抛出异常的线程只有加锁，没有释放锁的过程。只要有一个线程没有释放，和这个锁相关的所有线程都会阻塞。

### 条件变量

在C++11中，线程使用条件变量（condition\_variable）实现相互间的同步操作。条件变量利用线程间共享全局变量（即条件变量）进行同步，主要包括两个动作：

（1）一个线程等待“条件变量的条件成立”，当该条件不满足时，被阻塞、挂起，即进入等待状态；

（2）其它线程使“定义在条件变量上的条件成立”，给出信号，通知唤醒等待的线程。

条件变量与互斥锁/互斥量mutex、用户提供的判定条件相互配合，一起组合使用，可以实现生产者-消费者等复杂的同步互斥问题。条件变量可以原子地使得线程在唤醒时检查用户定义的判定条件，如果条件不满足就释放互斥锁，并入阻塞。

线程修改条件变量的动作为：

* + （1）获得一个定义在条件变量std::condition\_variable上的互斥锁std::mutex；
  + （2）当持有互斥锁后，修改条件变量；
  + （3）对条件变量执行notify\_one或notify\_all。说明：当执行notify动作时，不必持有锁

共享变量是原子性的，必须在mutex的保护下被修改，以便能够将对条件变量的修改改动正确notify发布到正在等待的线程。

等待条件变量std::condition\_variable的线程必须：

* + （1）获取std::unique\_lock<std::mutex>，此处的互斥量mutex用于保护作为条件的共享变量；
  + （2）执行wait, wait\_for或者wait\_until. 这些等待动作原子性地释放mutex，并使得线程的执行暂停；
  + （3）当获得条件变量的通知notify，或者超时，或者一个虚假的唤醒，线程被唤醒，并且获得mutex. 然后线程检查条件是否成立，如果是虚假唤醒，就继续等待。

与条件变量相关的函数如下。



1. 等待函数wait：

(1) wait(unique\_lock <mutex>＆lck)

当前线程的执行会被阻塞，直到收到notify为止。

(2) wait(unique\_lock <mutex>＆lck，Predicate pred)

当前线程仅在pred=false时阻塞；如果pred=true时，不阻塞。

wait()执行包括三个步骤：释放互斥锁、等待在条件变量上、再次获取互斥锁。

2. 通知函数notify\_one：

notify\_one()：没有参数，也没有返回值。

解除阻塞当前正在等待此条件的线程之一。如果没有线程在等待，则函数不执行任何操作。如果超过一个，不会指定具体哪一线程。

### 基于condition\_variable和mutex的信号量实现

使用条件变量和互斥锁，通过计数实现信号量：



### 基于条件变量的生产者-消费者问题

示例程序1：

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <thread>

#include <queue>

#include <mutex>

std::mutex g\_cvMutex; //二元互斥信号量

std::condition\_variable g\_cv; //条件变量

//缓冲区队列/buffer

std::deque<int> g\_data\_deque;

//缓冲区buffer最大数目

const int MAX\_NUM = 30;

//缓冲区指针

int g\_next\_index = 0;

//生产者，消费者线程个数

const int PRODUCER\_THREAD\_NUM = 3;

const int CONSUMER\_THREAD\_NUM = 3;

void producer\_thread(int thread\_id)

{

while (true)

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(500)); //线程延时、睡眠

//对缓冲区队列加锁

std::unique\_lock <std::mutex> lk(g\_cvMutex);

//当缓冲区队列未满时，继续添加数据

g\_cv.wait(lk, [](){ return g\_data\_deque.size() <= MAX\_NUM; }); //队列未满

g\_next\_index++; //指针下移

g\_data\_deque.push\_back(g\_next\_index); //数据加入队列

std::cout << "producer\_thread: " << thread\_id << " producer data: " << g\_next\_index;

std::cout << " queue size: " << g\_data\_deque.size() << std::endl;

//唤醒其他线程

g\_cv.notify\_all();

//自动释放锁

}

}

void consumer\_thread(int thread\_id)

{

while (true)

{

std::this\_thread::sleep\_for(std::chrono::milliseconds(550));

//对缓冲区队列加锁

std::unique\_lock <std::mutex> lk(g\_cvMutex);

//检测条件是否达成：队列不为空，有数据

g\_cv.wait(lk, []{return !g\_data\_deque.empty(); });

//互斥操作，从队列中取数据

int data = g\_data\_deque.front();

g\_data\_deque.pop\_front();

std::cout << "\tconsumer\_thread: " << thread\_id << " consumer data: ";

std::cout << data << " deque size: " << g\_data\_deque.size() << std::endl;

//唤醒其他线程

g\_cv.notify\_all();

//自动释放锁

}

}

nt main()

{

std::thread arrRroducerThread[PRODUCER\_THREAD\_NUM];

std::thread arrConsumerThread[CONSUMER\_THREAD\_NUM];

for (int i = 0; i < PRODUCER\_THREAD\_NUM; i++)

{

arrRroducerThread[i] = std::thread(producer\_thread, i);

}

for (int i = 0; i < CONSUMER\_THREAD\_NUM; i++)

{

arrConsumerThread[i] = std::thread(consumer\_thread, i);

}

for (int i = 0; i < PRODUCER\_THREAD\_NUM; i++)

{

arrRroducerThread[i].join();

}

for (int i = 0; i < CONSUMER\_THREAD\_NUM; i++)

{

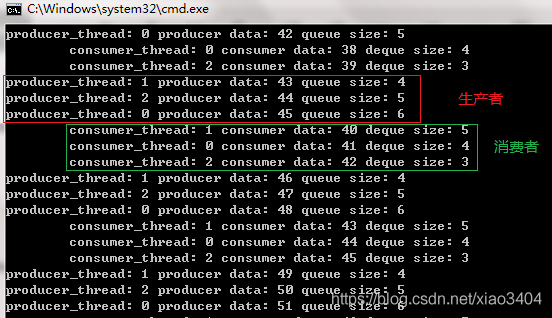
arrConsumerThread[i].join();

}

return 0;

}

运行结果：



**示例程序2：**

// condition\_variable::notify\_one

#include <iostream> // std::cout

#include <thread> // std::thread

#include <mutex> // std::mutex, std::unique\_lock

#include <condition\_variable> // std::condition\_variable

std::mutex mtx;

std::condition\_variable produce,consume;

int cargo = 0; // shared value by producers and consumers

void consumer () {

std::unique\_lock<std::mutex> lck(mtx);

while (cargo==0) consume.wait(lck);

std::cout << cargo << '\n';

cargo=0;

produce.notify\_one();

}

void producer (int id) {

std::unique\_lock<std::mutex> lck(mtx);

while (cargo!=0) produce.wait(lck);

cargo = id;

consume.notify\_one();

}

int main ()

{

std::thread consumers[10],producers[10];

// spawn 10 consumers and 10 producers:

for (int i=0; i<10; ++i) {

consumers[i] = std::thread(consumer);

producers[i] = std::thread(producer,i+1);

}

// join them back:

for (int i=0; i<10; ++i) {

producers[i].join();

consumers[i].join();

}

return 0;

}