### thread

thread.detach()的话，句柄t1 已经无法管理这个线程了。当主线程结束，进程就结束了，线程没运行完。

thread是可以std::move(t0)的，转移控制权。

//this\_thread functions  
std::this\_thread::get\_id();  
//让出当前线程  
std::this\_thread::yield();  
std::this\_thread::sleep\_until (chrono::system\_clock::now()+chrono::seconds(3));  
std::this\_thread::sleep\_for(std::chrono::seconds(1));

### mutex

#include <functional>  
#include <iostream>  
#include <memory>  
#include <tuple>  
#include <chrono>  
#include <map>  
#include <future>  
#include <condition\_variable>  
  
using namespace std;  
  
class Tick {  
 public:  
 chrono::system\_clock::time\_point g\_time\_point;  
 void producer(chrono::milliseconds t) {  
 while (true) {  
 unique\_lock<mutex> lock(mutex\_);  
 g\_time\_point = chrono::system\_clock::now();  
 lock.unlock();  
 this\_thread::sleep\_for(t);  
 }  
 };  
 void consumer() {  
 while (true) {  
 cout << chrono::duration\_cast<chrono::milliseconds>(g\_time\_point.time\_since\_epoch()).count() << endl;  
 this\_thread::sleep\_for(500ms);  
 }  
 };  
 private:  
 mutex mutex\_;  
  
};  
  
int main() {  
 Tick tick;  
 thread t1(&Tick::producer, &tick, 300ms);  
 thread t3(&Tick::producer, &tick, 1000ms);  
 thread t2(&Tick::consumer, &tick);  
 t1.join();  
}

注意看这段代码，我创建了两个生产者，一个消费者，然后通过循环的方式取出数据，为了传递消息，我不得不使用全局变量，这样的实现会导致消费者取出数据存在时间偏差。

### std::condition\_variable

std::condition\_variable\_any更加通用,这就可能产生额外的开销,所以std::condition\_variable一般作为首选的类型

#include <functional>  
#include <iostream>  
#include <memory>  
#include <tuple>  
#include <chrono>  
#include <map>  
#include <future>  
#include <condition\_variable>  
#include <queue>  
  
using namespace std;  
  
class Tick {  
 public:  
 void producer (chrono::milliseconds t)  
 {  
 while (true)  
 {  
 this\_thread::sleep\_for (t);  
 unique\_lock<mutex> lock (mutex\_);  
 msg\_.push (chrono::system\_clock::now ());  
 con\_var\_.notify\_one ();  
 }  
 };  
 void consumer ()  
 {  
 while (true) {  
 unique\_lock<mutex> lock (mutex\_);  
 con\_var\_.wait (lock, [this] { return !msg\_.empty (); });  
 while (!msg\_.empty()) {  
 cout << chrono::duration\_cast<chrono::milliseconds>(msg\_.front().time\_since\_epoch()).count() << endl;  
 msg\_.pop();  
 }  
 lock.unlock();  
 this\_thread::sleep\_for(1000ms);  
 }  
 };  
 private:  
 mutex mutex\_;  
 queue<chrono::system\_clock::time\_point> msg\_;  
 condition\_variable con\_var\_;  
};  
  
int main ()  
{  
 Tick tick;  
 cout<<"start point\n"<<chrono::duration\_cast<chrono::milliseconds>(chrono::system\_clock::now().time\_since\_epoch()).count()<<endl;  
 thread t1 (&Tick::producer, &tick, 300ms);  
 thread t3 (&Tick::producer, &tick, 500ms);  
 thread t2 (&Tick::consumer, &tick);  
 t3.join ();  
}

这里我们把代码修改为使用消息队列和条件变量的形式。

生产者线程在上锁后，把数据写入到消息队列，然后解锁。

消费者线程先上锁，然后condition.wait()，wait传入锁和条件谓词函数，判断失败会释放锁进入等待状态，判断成功则会重新获取锁，进入下面的操作，从消息队列中取出数据，注：取出之后再处理可以减小锁的粒度。

等到下一次有notify的时候，wait()会被唤醒，重新做检查。

几种函数

image-20230425104956685

**wait\_for**

template <class Rep, class Period>  
cv\_status wait\_for (unique\_lock<mutex>& lck,  
const chrono::duration<Rep,Period>& rel\_time);  
template <class Rep, class Period, class Predicate>  
bool wait\_for (unique\_lock<mutex>& lck,  
const chrono::duration<Rep,Period>& rel\_time, Predicate  
pred);

和wait不同的是,wait\_for可以执行一个时间段,在线程收到唤醒通知或者时间超时之前,该线程都会  
处于阻塞状态,如果收到唤醒通知或者时间超时,wait\_for返回,剩下操作和wait类似。

与wait\_for类似,只是**wait\_until**可以指定一个时间点

TODO：线程安全队列？p100

### future

问题引入

mutex m\_result;  
condition\_variable cv\_result;  
  
void task(int a, int b, int &res) {  
 unique\_lock<mutex> lg(m\_result);  
 a \*= 2;  
 b \*= 3;  
 res = a + b;  
 cv\_result.notify\_one();  
}  
  
int main() {  
 int result{};  
 thread t1(task, 1, 2, ref(result));  
 cout << "result: " << result << endl;  
 unique\_lock<mutex> lg(m\_result);  
 cv\_result.wait(lg, [&result] { return result != 0; });  
 cout << "result: " << result << endl;  
 t1.join();  
}

为了异步地获取一个计算结果，我们需要开启一个线程，在函数中传递一个引用，为了保护变量需要加锁，为了不被阻塞，需要加条件变量，这样很**麻烦**。

我们介绍标准库的组件<future>.

由于运算速度问题我们有时候不能马上得到结果，这时需要启动一个异步任务，并在任务结束后取回结果，这就是async，async返回一个future对象，可以使用get()方法获取结果，此时会阻塞在caller线程。

int main() {  
 auto future\_time = async(Tick::Wait);//bind future and async:nonblocked here  
 cout << "current time point:\n" << Tick::Now() << endl;  
 this\_thread::sleep\_for(1s);  
 cout << "current time point:\n" << Tick::Now() << endl;  
 cout << "the time is: \n" << future\_time.get() << endl;//blocked  
}

使用wait()可以等待运行结束再继续，类似于thread.join()

int main() {  
 auto future\_time = async(Tick::Wait);  
 cout << "current time point:\n" << Tick::Now() << endl;  
 this\_thread::sleep\_for(1s);  
 future\_time.wait();  
 cout << "current time point:\n" << Tick::Now() << endl;  
 cout << "the time is: \n" << future\_time.get() << endl;  
}

注意：future.get()只能调用一次。

通过额外向 async()传入std::launch::deferred参数，可以使得任务被延迟到wait()或get()执行，而且是在调用者线程执行。

auto future\_time = async(launch::deferred,Tick::Wait);

如果不调用甚至不会创建执行函数，所以务必把需要修改的变量作为返回值。

优先选用基于任务而非基于线程：基于任务提供了比基于线程更高的抽象级别，不需要进行线程管理

#### std::package\_task

通过std::package\_task把各种callable obk包装起来，方便作为线程入口，packaged\_task可以让任务的封装和执行分开。

int main() {  
 packaged\_task task(Tick::Wait);  
 thread t1(std::ref(task));//task一定要执行，否则result.get()会持续阻塞。  
 future result = task.get\_future();  
 t1.join();  
 cout << result.get() << endl;  
}

包装一个函数，然后丢入线程，获取future对象，等待线程结束，从future拿到结果。

#### std::promise

int Calculate(promise<int> &pr) {  
 this\_thread::sleep\_for(2s);  
 pr.set\_value(1);  
 return 0;  
}  
int main() {  
 promise<int> pr;  
 thread t1(Calculate, std::ref(pr));  
 t1.join();  
 auto f = pr.get\_future();  
 cout << f.get() << endl;  
}

promise是一个承诺，当线程创建了promise时，它向线程承诺，这个值以后一定会被设置。

### 原子操作<atomic>

atomic<int> a = 0;  
//int a{} ;  
  
template<typename T>  
function<void(T &)> f(T &a) {  
 for (int i = 0; i < 1000000; i++) {  
 a++;  
 }  
}  
  
int main() {  
// thread t1(f<int>, ref(a));  
// thread t2(f<int>, ref(a));  
 thread t1(f<atomic<int>>, ref(a));  
 thread t2(f<atomic<int>>, ref(a));  
 t1.join(), t2.join();  
 cout << a << endl;  
}