# 题目

我们提供了一个类：

public class Foo {

  public void first() { print("first"); }

  public void second() { print("second"); }

  public void third() { print("third"); }

}

三个不同的线程 A、B、C 将会共用一个 Foo 实例。

一个将会调用 first() 方法

一个将会调用 second() 方法

还有一个将会调用 third() 方法

请设计修改程序，以确保 second() 方法在 first() 方法之后被执行，third() 方法在 second() 方法之后被执行。

示例 1:

输入: [1,2,3]

输出: "firstsecondthird"

解释:

有三个线程会被异步启动。

输入 [1,2,3] 表示线程 A 将会调用 first() 方法，线程 B 将会调用 second() 方法，线程 C 将会调用 third() 方法。

正确的输出是 "firstsecondthird"。

示例 2:

输入: [1,3,2]

输出: "firstsecondthird"

解释:

输入 [1,3,2] 表示线程 A 将会调用 first() 方法，线程 B 将会调用 third() 方法，线程 C 将会调用 second() 方法。

正确的输出是 "firstsecondthird"。

提示：

尽管输入中的数字似乎暗示了顺序，但是我们并不保证线程在操作系统中的调度顺序。

你看到的输入格式主要是为了确保测试的全面性。

# 分析

## 方法一：原子操作

我们平时进行的数据修改都是非原子操作，如果多个线程同时以非原子操作的方式修改同一个对象可能会发生数据争用，从而导致未定义行为；而原子操作能够保证多个线程顺序访问，不会导致数据争用，其执行时没有任何其它线程能够修改相同的原子对象。

针对这道题，我们可以让 second 和 third 函数等待原子变量被修改为某个值后再执行，然后分别在 first 和 second 函数中来修改这个原子变量。

c++ 11 提供了 std::atomic<T> 模板类来构造原子对象：

class Foo {

std::atomic<bool> a{ false };

std::atomic<bool> b{ false };

public:

void first(function<void()> printFirst) {

printFirst();

a = true;

}

void second(function<void()> printSecond) {

while (!a)

this\_thread::sleep\_for(chrono::milliseconds(1));

printSecond();

b = true;

}

void third(function<void()> printThird) {

while (!b)

this\_thread::sleep\_for(chrono::milliseconds(1));

printThird();

}

};

值得注意的是，原子操作的实现跟处理器和操作系统内核相关，因此 c++ 标准并没有规定 atomic 的实现是否是无锁的（lock-free），只规定了需要提供一个 is\_lock\_free() 来查询当前编译器对 atomic 的实现是否是无锁的。

## 方法二：互斥锁

互斥锁是用来防止多个线程同时访问共享资源对象的机制，在同一时间只有一个线程可以拥有一个特定的锁对象，其他线程如果尝试获取锁会阻塞直到锁资源被释放或直接返回失败。

针对这道题我们可以用两个互斥锁来阻塞second和third函数，分别在 first 和 second 执行结束后解锁。

**代码：**

class Foo {

mutex mtx1, mtx2;

public:

Foo() {

mtx1.lock(), mtx2.lock();

}

void first(function<void()> printFirst) {

printFirst();

mtx1.unlock();

}

void second(function<void()> printSecond) {

mtx1.lock();

printSecond();

mtx1.unlock();

mtx2.unlock();

}

void third(function<void()> printThird) {

mtx2.lock();

printThird();

mtx2.unlock();

}

};

这段代码能够 ac，但实际上这种使用 mutex 的方法是错误的，因为根据 c++ 标准，在一个线程尝试对一个 mutex 对象进行 unlock 操作时，mutex 对象的所有权必须在这个线程上；也就是说，应该由同一个线程来对一个 mutex 对象进行 lock 和 unlock 操作，否则会产生未定义行为。题目中提到了 first, second, third 三个函数分别是由三个不同的线程来调用的，但我们是在 Foo 对象构造时（可以是在 create 这几个线程的主线程中，也可以是在三个线程中的任意一个）对两个 mutex 对象进行 lock 操作的，因此，调用 first 和 second 函数的两个线程中至少有一个在尝试获取其他线程所拥有的 mutex 对象的所有权。

另外，如果非要讨论这个解法有什么优化的余地的话，因为 mutex 对象本身是不保护任何数据的，我们只是通过 mutex 的机制来保护数据被同时访问，所以最好使用 lock\_guard 或者 unique\_lock 提供的 RAII 机制来管理 mutex 对象，而不是直接操作 mutex 对象；其中 lock\_guard 只拥有构造和析构函数，用来实现 RAII 机制，而 unique\_lock 是一个完整的 mutex 所有权包装器，封装了所有 mutex 的函数：

class Foo {

mutex mtx\_1, mtx\_2;

unique\_lock<mutex> lock\_1, lock\_2;

public:

Foo() : lock\_1(mtx\_1, try\_to\_lock), lock\_2(mtx\_2, try\_to\_lock) {

}

void first(function<void()> printFirst) {

printFirst();

lock\_1.unlock();

}

void second(function<void()> printSecond) {

lock\_guard<mutex> guard(mtx\_1);

printSecond();

lock\_2.unlock();

}

void third(function<void()> printThird) {

lock\_guard<mutex> guard(mtx\_2);

printThird();

}

};

## 方法三：条件变量

条件变量一般和互斥锁搭配使用，互斥锁用于上锁，条件变量用于在多线程环境中等待特定事件发生。

针对这道题我们可以分别在 first 和 second 执行完之后修改特定变量的值（例如修改成员变量 k 为特定值），然后通知条件变量，唤醒下一个函数继续执行。

std::condition\_variable 是一种用来同时阻塞多个线程的同步原语（synchronization primitive），std::condition\_variable 必须和 std::unique\_lock 搭配使用：

class Foo {

condition\_variable cv;

mutex mtx;

int k = 0;

public:

void first(function<void()> printFirst) {

printFirst();

k = 1;

cv.notify\_all(); // 通知其他所有在等待唤醒队列中的线程

}

void second(function<void()> printSecond) {

unique\_lock<mutex> lock(mtx); // lock mtx

cv.wait(lock, [this](){ return k == 1; }); // unlock mtx，并阻塞等待唤醒通知，需要满足 k == 1 才能继续运行

printSecond();

k = 2;

cv.notify\_one(); // 随机通知一个（unspecified）在等待唤醒队列中的线程

}

void third(function<void()> printThird) {

unique\_lock<mutex> lock(mtx); // lock mtx

cv.wait(lock, [this](){ return k == 2; }); // unlock mtx，并阻塞等待唤醒通知，需要满足 k == 2 才能继续运行

printThird();

}

};

std::condition\_variable::wait 函数会执行三个操作：先将当前线程加入到等待唤醒队列，然后 unlock mutex 对象，最后阻塞当前线程；它有两种重载形式，第一种只接收一个 std::mutex 对象，此时线程一旦接受到唤醒信号（通过 std::condition\_variable::notify\_one 或 std::condition\_variable::notify\_all 进行唤醒），则无条件立即被唤醒，并重新 lock mutex；第二种重载形式还会接收一个条件（一般是 variable 或者 std::function），即只有当满足这个条件时，当前线程才能被唤醒，它在 gcc 中的实现也很简单，只是在第一种重载形式之外加了一个 while 循环来保证只有在满足给定条件后才被唤醒，否则重新调用 wait 函数：

template<typename \_Predicate>

void wait(unique\_lock<mutex>& \_\_lock, \_Predicate \_\_p)

{

while (!\_\_p())

wait(\_\_lock);

}

## 方法四：信号量

信号量是用来实现对共享资源的同步访问的机制，其使用方法和条件变量类似，都是通过主动等待和主动唤醒来实现的。

c++ 标准库中并没有信号量的实现和封装，我们可以用 c 语言提供的 <sempahore.h> 库来解题 ：

#include <semaphore.h>

class Foo {

private:

sem\_t sem\_1, sem\_2;

public:

Foo() {

sem\_init(&sem\_1, 0, 0), sem\_init(&sem\_2, 0, 0);

}

void first(function<void()> printFirst) {

printFirst();

sem\_post(&sem\_1);

}

void second(function<void()> printSecond) {

sem\_wait(&sem\_1);

printSecond();

sem\_post(&sem\_2);

}

void third(function<void()> printThird) {

sem\_wait(&sem\_2);

printThird();

}

};

## 方法五：异步

异步操作是一种，在不需要等待被调用方返回结果之前，就让操作继续进行下去的方法。针对这道题可以使用基于 future/promise 的异步编程模型。

future 和 promise 起源于函数式编程，其目的是将值（future）和计算方式（promise）分离，使得 promise 可以异步地修改 future，从而提高代码的可读性，并减少通信延迟。

std::future 是用来获取异步操作结果的模板类；std::packaged\_task, std::promise, std::async 都可以进行异步操作，并拥有一个 std::future 对象，用来存储它们所进行的异步操作返回或设置的值（或异常），这个值会在将来的某一个时间点，通过某种机制被修改后，保存在其对应的 std::future 对象中：

对于 std::promise，可以通过调用 std::promise::set\_value 来设置值并通知 std::future 对象：

class Foo {

promise<void> pro1, pro2;

public:

void first(function<void()> printFirst) {

printFirst();

pro1.set\_value();

}

void second(function<void()> printSecond) {

pro1.get\_future().wait();

printSecond();

pro2.set\_value();

}

void third(function<void()> printThird) {

pro2.get\_future().wait();

printThird();

}

};

std::future<T>::wait 和 std::future<T>::get 都会阻塞地等待拥有它的 promise 对象返回其所存储的值，后者还会获取 T 类型的对象；这道题只需要利用到异步通信的机制，所以并没有返回任何实际的值。

std::packaged\_task 是一个拥有 std::future 对象的 functor，将一系列操作进行了封装，在运行结束之后会将返回值保存在其所拥有的 std::future<T> 对象中；同样地，在这道题中只需要利用到其函数运行结束之后通知 std::future 对象的机制：

class Foo {

function<void()> task = []() {};

packaged\_task<void()> pt\_1{ task }, pt\_2{ task };

public:

void first(function<void()> printFirst) {

printFirst();

pt\_1();

}

void second(function<void()> printSecond) {

pt\_1.get\_future().wait();

printSecond();

pt\_2();

}

void third(function<void()> printThird) {

pt\_2.get\_future().wait();

printThird();

}

};