# 题目

我们提供一个类：

class FooBar {

public void foo() {

    for (int i = 0; i < n; i++) {

      print("foo");

  }

}

public void bar() {

    for (int i = 0; i < n; i++) {

      print("bar");

    }

}

}

两个不同的线程将会共用一个 FooBar 实例。其中一个线程将会调用 foo() 方法，另一个线程将会调用 bar() 方法。

请设计修改程序，以确保 "foobar" 被输出 n 次。

示例 1:

输入: n = 1

输出: "foobar"

解释: 这里有两个线程被异步启动。其中一个调用 foo() 方法, 另一个调用 bar() 方法，"foobar" 将被输出一次。

示例 2:

输入: n = 2

输出: "foobarfoobar"

解释: "foobar" 将被输出两次。

# 分析

## 方法一：互斥锁

**思路：**

创建两个mutex类型变量，利用其lock和unlock函数实现（未遵循only the owner can unlock the mutex）。

**代码：**

class FooBar {

private:

int n;

mutex m1,m2;

public:

FooBar(int n) {

this->n = n;

m2.lock();

}

void foo(function<void()> printFoo) {

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

m1.lock();

// printFoo() outputs "foo". Do not change or remove this line.

printFoo();

m2.unlock();

}

}

void bar(function<void()> printBar) {

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

m2.lock();

// printBar() outputs "bar". Do not change or remove this line.

printBar();

m1.unlock();

}

}

};

## 方法二：互斥锁+条件变量

class FooBar {

private:

int n;

mutex mtx;

condition\_variable cv;

bool foo\_done = false;

public:

FooBar(int n) : n(n) {}

void foo(function<void()> printFoo) {

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

unique\_lock<mutex> lock(mtx);

cv.wait(lock, [&]() { return !foo\_done; });

printFoo();

foo\_done = true;

cv.notify\_one();

}

}

void bar(function<void()> printBar) {

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

unique\_lock<mutex> lock(mtx);

cv.wait(lock, [&]() { return foo\_done; });

printBar();

foo\_done = false;

cv.notify\_one();

}

}

};

另一种写法：

class FooBar {

private:

int n;

public:

FooBar(int n) {

this->n = n;

}

void foo(function<void()> printFoo) {

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

std::unique\_lock<std::mutex> lk(Mu);//获取Mu锁

v.wait(lk,[this](){return count == 1;});//看v是否满足条件，（锁和变量）

// printFoo() outputs "foo". Do not change or remove this line.

printFoo();

count++;//变量+1，=2，等于2时，foo会阻塞，bar的condition\_variable 满足条件

v.notify\_one();//通知并唤醒阻塞在v2里面的线程

}

}

void bar(function<void()> printBar) {

//注释同上

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

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

v.wait(lk,[this](){return count == 2;});

// printBar() outputs "bar". Do not change or remove this line.

printBar();

count--;

v.notify\_one();

}

}

private:

int count = 1;//条件变量

std::condition\_variable v;//条件变量对象

std::mutex Mu;//定义一个锁

};

## 方法三：信号量

#include <semaphore.h> // 需要手动包含信号量头文件

class FooBar {

private:

int n;

sem\_t foo\_done, bar\_done;

public:

FooBar(int n) : n(n) {

sem\_init(&foo\_done, 0 , 0);

sem\_init(&bar\_done, 0 , 1);

}

void foo(function<void()> printFoo) {

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

sem\_wait(&bar\_done);

printFoo();

sem\_post(&foo\_done);

}

}

void bar(function<void()> printBar) {

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

sem\_wait(&foo\_done);

printBar();

sem\_post(&bar\_done);

}

}

};

## 方法四：原子操作

class FooBar {

private:

int n;

atomic<bool> foo\_done = false;

public:

FooBar(int n) : n(n) {}

void foo(function<void()> printFoo) {

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

while (foo\_done) {

this\_thread::yield();

}

printFoo();

foo\_done = true;

}

}

void bar(function<void()> printBar) {

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

while (!foo\_done) {

this\_thread::yield();

}

printBar();

foo\_done = false;

}

}

};

另一种写法：

class FooBar {

private:

int n;

atomic<bool> fooed = false;

public:

FooBar(int n) {

this->n = n;

}

void foo(function<void()> printFoo) {

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

while(fooed.load())this\_thread::yield();

// printFoo() outputs "foo". Do not change or remove this line.

printFoo();

fooed.store(true);

}

}

void bar(function<void()> printBar) {

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

while(!fooed.load())this\_thread::yield();

// printBar() outputs "bar". Do not change or remove this line.

printBar();

fooed.store(false);

}

}

};

## 方法五：异步