C++11互斥体

1 什么是互斥体

互斥体是用于多线程编程中，防止多个线程对同一个共享数据同时访问形成资源竞争的同步原语（原语就是执行过程中不能被中断的语句）。

2 为何使用互斥体

示例1

```
#include <thread>
#include <iostream>

static int share_data = 0;

void SelfIncreaseProc()
{
        for(int i=0; i<100000; ++i) // 迭代次数太少可能会不能观察到现象
        {
                ++share_data;
        }
}

void SelfDecreaseProc()
{
        for(int i=0; i<100000; ++i) // 迭代次数太少可能会不能观察到现象
        {
                --share_data;
        }
}

int main(int argc, char** argv)
{
```

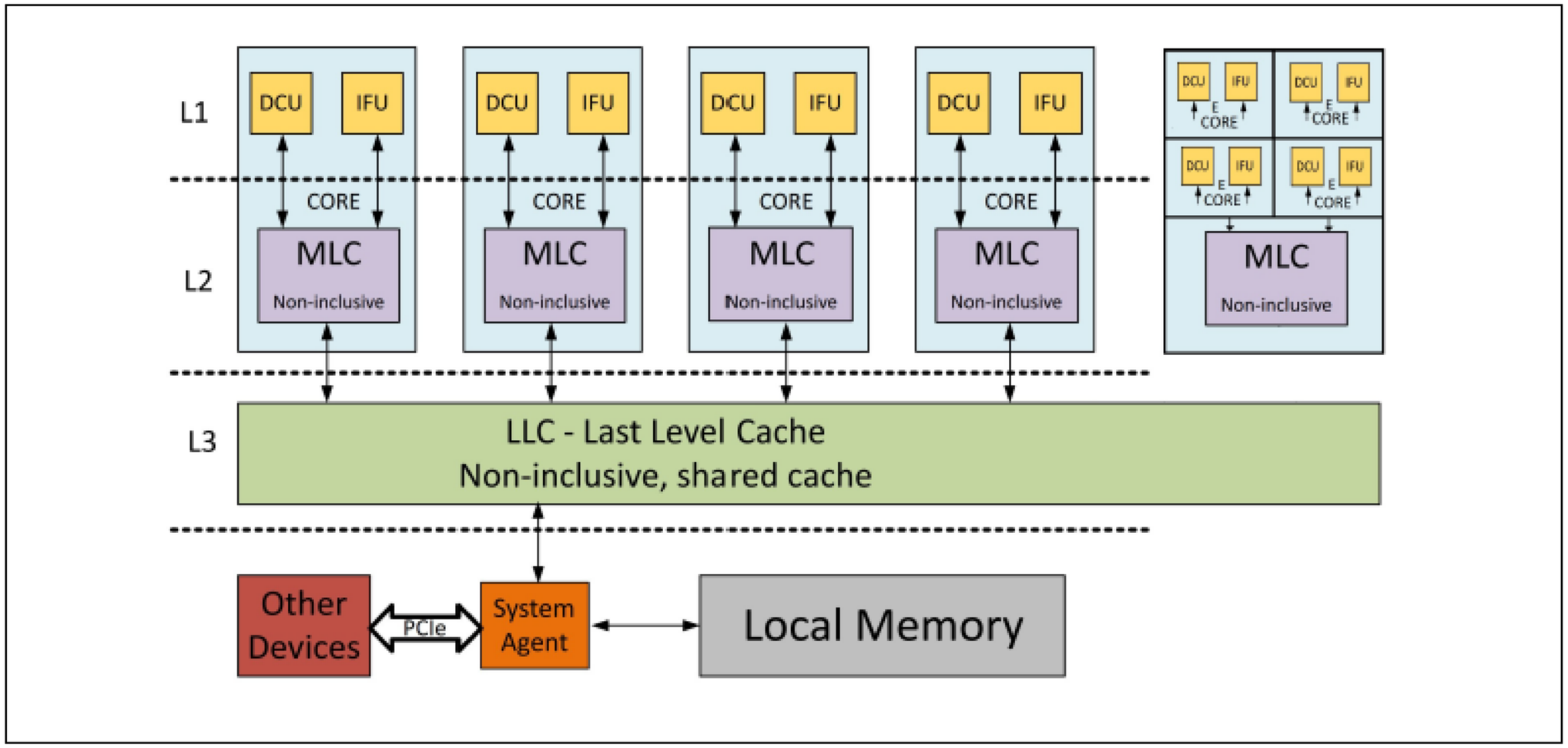
```
        std::thread threadA(SelfIncreaseProc);
        std::thread threadB(SelfDecreaseProc);

        threadA.join();
        threadB.join();

        std::cout << "share_data: " << share_data << std::endl;
}
```

结果

```
bob@bob-VirtualBox:~/Code/mutex$ ./exampl1
a: -182517
bob@bob-VirtualBox:~/Code/mutex$ ./exampl1
a: -211887
bob@bob-VirtualBox:~/Code/mutex$ ./exampl1
a: -25818
bob@bob-VirtualBox:~/Code/mutex$ ./exampl1
a: -210783
bob@bob-VirtualBox:~/Code/mutex$ ./exampl1
a: -131592
bob@bob-VirtualBox:~/Code/mutex$ ./exampl1
```

为何会有这样的结果

缓存一致性&顺序一致性



3 C++中使用互斥体

从C++11标准开始，我们可以使用头文件<mutex>所提供的多种互斥体，接下来我们一起探索下几种常用的互斥体和它们常用的方法。

3.1 std::mutex

`mutex` 提供**排他性非递归所有权**语义。

std::mutex::lock

给互斥体上锁，如果有其他线程已经锁定这个互斥体，那当前线程会被阻塞，直到另外的线程解锁这个互斥体。

std::mutex::try_lock

与lock相比，返回一个bool值，如果尝试锁定互斥体失败则立即返回false。当然也可能会虚假失败，即互斥体未被锁定的情况下返回false。

std::mutex::unlock

解锁互斥体，使得因为尝试锁定而被阻塞的其他线程可以继续往下执行。

示例2

```
#include <thread>
#include <iostream>
#include <mutex>

static int share_data = 0;
static std::mutex mtx;

void SelfIncreaseProc()
{
        // 为何不写在for外面？
        for(int i=0; i<100000; ++i)
        {
                mtx.lock();
                ++share_data;
                mtx.unlock();
        }
}

void SelfDecreaseProc()
{
        for(int i=0; i<100000; ++i)
        {
                mtx.lock();
                --share_data;
                mtx.unlock();
        }
}
```

```
int main(int argc, char** argv)
{
        std::thread threadA(SelfIncreaseProc);
        std::thread threadB(SelfDecreaseProc);

        threadA.join();
        threadB.join();
        std::cout << "share_data: " << share_data << std::endl;
}
```

结果

```
bob@bob-VirtualBox:~/Code/mutex$ ./example2
a: 0
bob@bob-VirtualBox:~/Code/mutex$ ./example2
a: 0
bob@bob-VirtualBox:~/Code/mutex$ ./example2
a: 0
bob@bob-VirtualBox:~/Code/mutex$ ./example2
a: 0
bob@bob-VirtualBox:~/Code/mutex$ ./example2
a: 0
```

3.2 std::timed_mutex

以类似 mutex 的行为， `timed_mutex` 提供**排他性非递归所有权**语义。
除了std::mutex提供的成员方法，timed_mutex还提供try_lock_for和try_lock_until这两个带时间类型的方法。

std::timed_mutex::try_lock_for

在给定的时间范围内尝试对互斥体加锁，如果加锁成功则返回true，如果超时未加锁成功则返回false。

std::timed_mutex::try_lock_until

在给定的时间点到达之前尝试对互斥体加锁，如果加锁成功则返回true，如果超时未加锁成功则返回false。

3.3 std::recursive_mutex

成员函数和mutex一样，但是区别是：`recursive_mutex` **提供排他性递归所有权语义。**
1. *多个线程同一时间只能有一个线程对互斥体加锁。*
2. *当前线程可以多次对互斥体加锁。*

示例3

```
#include <thread>
#include <mutex>
#include <iostream>
#include <chrono>

static int shared_data = 0;
static std::recursive_mutex rcv_mtx;

void RecursiveCall(int& num)
{
        if(--num == 0)
    {
                return;
    }

        rcv_mtx.lock(); // 如果是std::mutex，第二次锁定在这会死锁

        ++shared_data;
        RecursiveCall(num);

        rcv_mtx.unlock();
}

int main()
{
        int num = 1000;
        // 线程创建是以值传递进行的
        std::thread threadA(RecursiveCall, std::ref(num));
```

```
        // 确保线程A先执行
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

        std::thread threadB([]()
                {
                        rcv_mtx.lock();
                        std::cout << "shared_data: " << shared_data << std::endl;
                        rcv_mtx.unlock();
                });

        threadA.join();
    threadB.join();
}
```

PS: C++11额外提供了std::recursive_timed_mutex，主要是提供了递归使用的带超时设置的互斥体。

3.4 std::lock_guard<Mutex>

因为C++互斥体每次都要手动调用lock/unlock很麻烦，所以C++11提供了lock_guard来自动获取和释放互斥体，它是符合RAII(Resource Acquisition Is Initialization)风格的，及由构造函数完成互斥体的锁定，析构函数完成互斥体的解锁。

示例4

```
#include <mutex>

static std::mutex mtx;
static int shared_data = 0;

void IncreaseData()
{
        std::lock_guard guard(mtx); // 会在构造时调用lock，析构时调用unlock
        ++shared_data;
}
```

下期预告->条件变量