C++ 的 6 种内存顺序，你都知道吗？

CPP开发者 2020-11-30 03:50

(给CPP开发者加星标，提升C/C++技能)

> 来源：盐焗咸鱼
> https://blog.csdn.net/qq_33215865/article/details/88089927

原子操作的内存顺序

有六个内存顺序选项可应用于对原子类型的操作：

1. memory_order_relaxed

2. memory_order_consume

3. memory_order_acquire

4. memory_order_release

5. memory_order_acq_rel

6. memory_order_seq_cst。

除非你为特定的操作指定一个顺序选项，否则内存顺序选项对于所有原子类型默认都是memory_order_seq_cst。

6个内存顺序可以分为3类：

1. 自由顺序
(memory_order_relaxed)

2.获取-释放顺序
(memory_order_consume, memory_order_acquire, memory_order_release 和 memory_order_acq_rel)

3.排序一致顺序
(memory_order_seq_cst)

1、std::memory_order_relaxed “自由”内存顺序

在原子类型上的操作以自由序列执行，没有任何同步关系，仅对此操作要求原子性。例如，在某一线程中，先写入A，再写入B。但是在多核处理器中观测到的顺序可能是先写入B，再写入A。自由内存顺序对于不同变量可以自由重排序。

这是因为不同的CPU缓存和内部缓冲区，在同样的存储空间中可以存储不同的值。对于非一致排序操作，线程没有必要去保证一致性。

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
  x.store(true,std::memory_order_relaxed);
  y.store(true,std::memory_order_relaxed);
}
void read_y_then_x()
{
  while(!y.load(std::memory_order_relaxed));
  if(x.load(std::memory_order_relaxed))
    ++z;
}
int main()
{
  x=false;
  y=false;
  z=0;
  std::thread a(write_x_then_y);
```

```
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

上述代码，z.load()!=0有可能会返回false。在b线程中，多核处理器观测到的顺序是随机的。b线程中的观测到的变量的并不会与线程a中的变量做同步，没有任何顺序要求。

2、std::memory_order_release “释放”内存顺序

使用memory_order_release的原子操作，当前线程的读写操作都不能重排到此操作之后。例如，某一线程先写入A，再写入B，再以memeory_order_release操作写入C，再写入D。在多核处理器中观测到的顺序AB只能在C之前，不能出现C写入之后，A或B再写入的情况。但是，可能出现D重排到C之前的情况。

memory_order_release用于发布数据，放在写操作的最后。

3、std::memory_order_acquire “获取”内存顺序

使用memory_order_acquire的原子操作，当前线程的读写操作都不能重排到此操作之前。例如，某一线程先读取A，再读取B，再以memeory_order_acquire操作读取C，再读取D。在多核处理器中观测到的顺序D只能在C之前，不能出现先读取D，最后读取C的情况。但是，可能出现A或B重排到C之后的情况。

memory_order_acquire用于获取数据，放在读操作的最开始 。

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
```

```
    x.store(true,std::memory_order_relaxed);
    y.store(true,std::memory_order_release);
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));  // 自旋，等待y被设置为true
    if(x.load(std::memory_order_relaxed))
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

上述代码是使用“释放-获取"模型对“自由”模型的改进。z.load() != 0 返回的一定是true。首先，a线程中，y使用memory_order_release释放内存顺序，在多核处理器观测到的顺序，x的赋值肯定会位于y之前。b线程中，y的获取操作是同步操作，x的访问顺序必定在y之后，观测到的x的访问值一定为true。

“获取”与“释放”一般会成对出现，用来同步线程。

4、std::memory_order_acq_rel "获取释放"内存顺序

memory_order_acq_rel带此内存顺序的读-改-写操作既是获得加载又是释放操作。没有操作能够从此操作之后被重排到此操作之前，也没有操作能够从此操作之前被重排到此操作之后。

```
std::atomic<int> sync(0);
void thread_1()
{
    // ...
    sync.store(1,std::memory_order_release);
```

```
}

void thread_2()
{
  int expected=1;
  while(!sync.compare_exchange_strong(expected,2,
              std::memory_order_acq_rel))
    expected=1;
}
void thread_3()
{
  while(sync.load(std::memory_order_acquire)<2);
  // ...
}
```

上述代码，使用memory_order_acq_rel来实现3个线程的同步。thread1执行写入功能，thread2执行读取功能。3个线程的执行顺序是确定的。compare_exchange_strong，当*this值与expected相同时，会将2赋值*this，返回true，不同时，将*this赋值expected，返回flase。

5、std::memory_order_consume 依赖于数据的内存顺序

memory_order_consume只会对其标识的对象保证该对象存储先行于那些需要加载该对象的操作。

```
struct X
{
int i;
std::string s;
};

std::atomic<X*> p;
std::atomic<int> a;

void create_x()
{
  X* x=new X;
  x->i=42;
  x->s="hello";
```

```
    a.store(99,std::memory_order_relaxed);
    p.store(x,std::memory_order_release);
}

void use_x()
{
    X* x;
    while(!(x=p.load(std::memory_order_consume)))
        std::this_thread::sleep(std::chrono::microseconds(1));
    assert(x->i==42);
    assert(x->s=="hello");
    assert(a.load(std::memory_order_relaxed)==99);   /
}

int main()
{
    std::thread t1(create_x);
    std::thread t2(use_x);
    t1.join();
    t2.join();
}
```

x->i ==42，和x-> == "hello"会被确保已被赋值。但是a的值却是不确定的。加载p的操作标记为memory_order_consume，这就意味着存储p仅先行那些需要加载p的操作，对于a是没有保障的。

6、std::memory_order_seq_cst “顺序一致”内存顺序

memory_order_seq_cst比std::memory_order_acq_rel更为严格。memory_order_seq_cst不仅是一个"获取释放"内存顺序，它还会对所有拥有此标签的内存操作建立一个单独全序。memory_order_acq_rel的顺序保障，是要基于同一个原子变量的。memory_order_acq_rel使用了两个不同的原子变量x1, x2，那在x1之前的读写，重排到x2之后，是完全可能的，在x1之后的读写，重排到x2之前，也是被允许的。然而，如果两个原子变量x1,x2，是基于memory_order_seq_cst在操作，那么即使是x1之前的读写，也不能被重排到x2之后，x1之后的读写，也不能重排到x2之前，也就说，如果都用memory_order_seq_cst，那么程序代码顺序(Program Order)就将会是你在多个线程上都实际观察到的顺序(Observed Order)。

顺序一致是最简单、直观的序列，但是它也是最昂贵的内存序列，它需要对所有线程进行全局同步，比其他的顺序造成更多的消耗。因为保证一致顺序，需要添加额外的指令。

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x()
{
  x.store(true,std::memory_order_seq_cst);
}

void write_y()
{
  y.store(true,std::memory_order_seq_cst);
}
void read_x_then_y()
{
  while(!x.load(std::memory_order_seq_cst));
  if(y.load(std::memory_order_seq_cst))
    ++z;
}
void read_y_then_x()
{
  while(!y.load(std::memory_order_seq_cst));
  if(x.load(std::memory_order_seq_cst))
    ++z;
}
int main()
{
  x=false;
  y=false;
  z=0;
  std::thread a(write_x);
```

```
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);
}
```

z.load() != 0 一定会为true。memory_order_seq_cst的语义会为所有操作都标记为memory_order_seq_cst建立一个单独全序。线程c和d总会有一个执行z++，x和y的赋值顺序，不管谁先谁后，在所有线程的眼中顺序都是确定的。

- EOF -

推荐阅读 — 点击标题可跳转

1、C++在嵌入式中表现如何?

2、C++ 堆栈工作机制

3、如何优雅地实现 C++ 编译期静态反射

关于 C++ 显式缺省和显式删除，欢迎在评论中和我探讨。觉得文章不错，请点赞和在看支持我继续分享好文。谢谢!