第 12 讲：Scalable Synchronization on Shared-Memory Multiprocessors - II

Multiprocessor Memory Model & Multiprocessor Programming

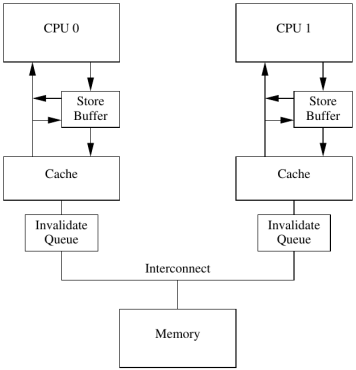
陈渝

清华大学计算机系

yuchen@tsinghua.edu.cn

2020 年 5 月 9 日

Recap

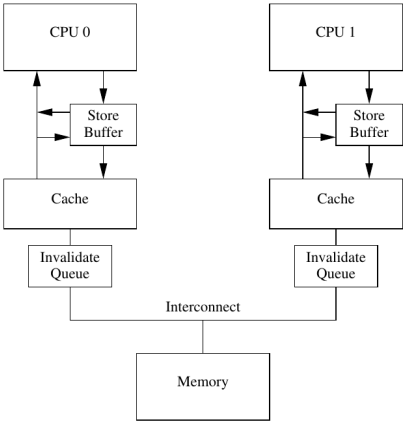


Cache Coherent in Multi-processor

- MSI 一致性协议
- MESI 一致性协议

ref: Some info are from
Paul McKenney (IBM) Tom Hart (University of Toronto), Frans Kaashoek (MIT), Daniel J. Sorin "A Primer on Memory Consistency and Cache Coherence", Fabian Giesen "Cache coherency primer", Mingyu Gao(Tsinghua),Yubin Xia(SJTU) "The C/C++ Memory Model: Overview and Formalization",Mark Batty,etc. "C++ 11 Memory Consistency Model", Sebastian Gerstenberg;"HOW UBISOFT MONTREAL DEVELOPS GAMES FOR MULTICORE Before & After C++11", Jeff Preshing;etc.

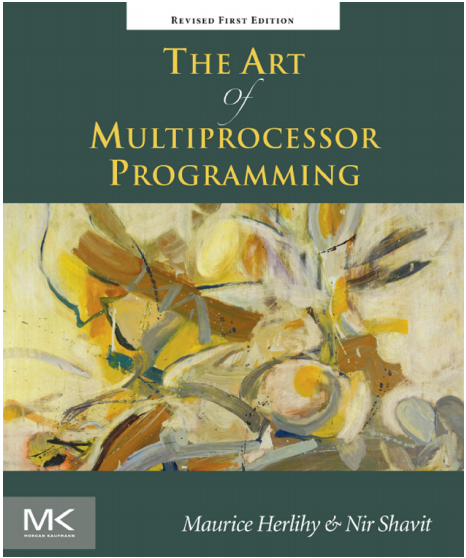
Recap

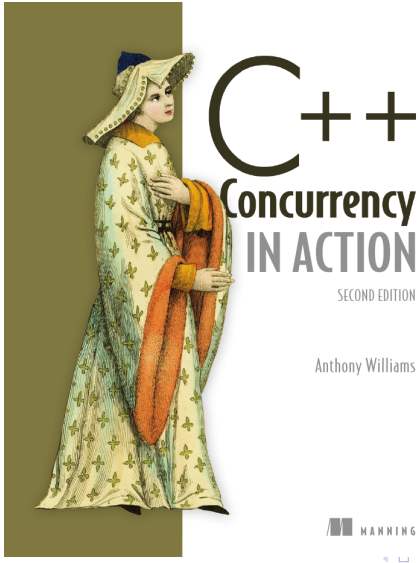


Memory Consistency in Multi-processor

- Sequential Consistency
- Total Store Order Consistency
- Acquire/Release Consistency
- Relaxed/Weak Consistency

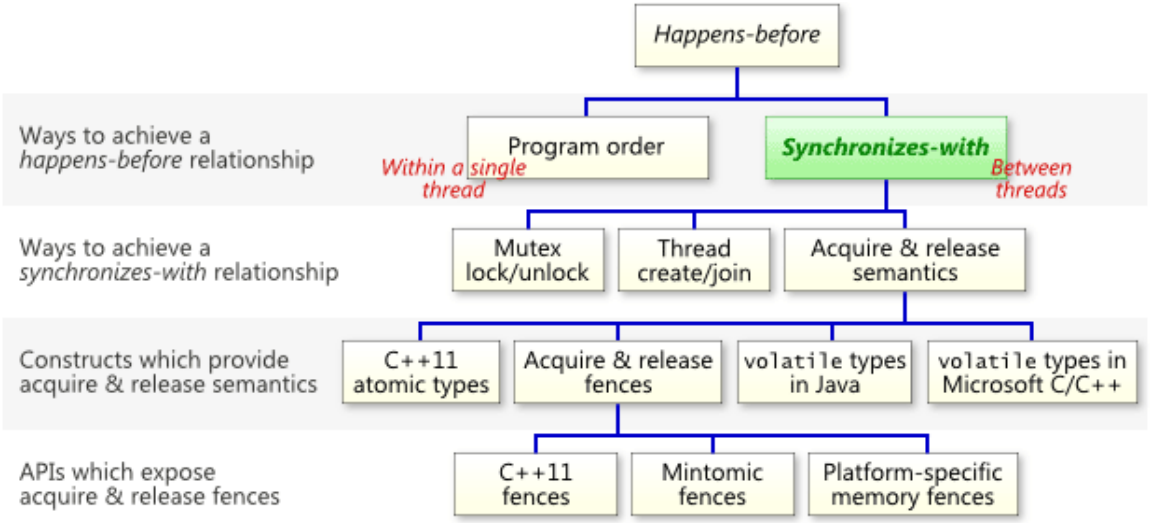
Multiprocessor Programming





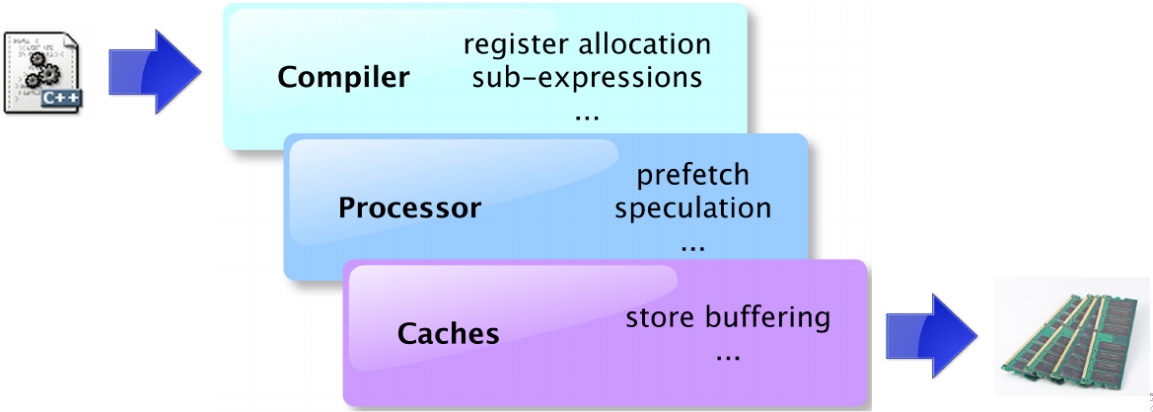
Ways to Achieve Synchronizes-With

C++11/17/20, Go, RUST, Java, ...



Real Hardware/Compiler

Real hardware doesn't run the code that you wrote.
Real compiler doesn't produce the code that you wrote.



Real Hardware/Compiler

Real hardware doesn't run the code that you wrote.
Real compiler doesn't produce the code that you wrote.

http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

Real Hardware/Compiler

Real hardware doesn't run the code that you wrote.
Real compiler doesn't produce the code that you wrote.

Example (Dekker's algorithm)

$$x = 0;\ y = 0;$$

$$\begin{array}{l|l} x = 1; & y = 1; \\ r1 = y; & r2 = x; \end{array}$$

$$\text{assert}\,(r1 == 1 \,||\, r2 == 1);$$

根据英特尔的规范: 在本示例的末尾，r1 和 r2 都等于 0 是合法的

Real Hardware/Compiler

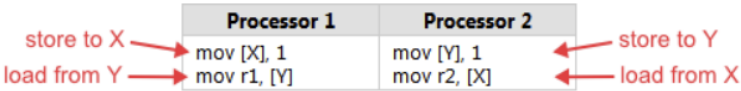
Real hardware doesn't run the code that you wrote.
Real compiler doesn't produce the code that you wrote.

Example (Dekker's algorithm)

$$x = 0;\ y = 0;$$

$$\begin{array}{l|l} x = 1; & y = 1; \\ r1 = y; & r2 = x; \end{array}$$

$$\text{assert}\,(r1 == 1 \,||\, r2 == 1);$$



Real Hardware/Compiler

Real hardware doesn't run the code that you wrote.
Real compiler doesn't produce the code that you wrote.

Example (Dekker's algorithm)

$$x = 0; y = 0;$$

$$\begin{array}{l|l} x = 1; & y = 1; \\ r1 = y; & r2 = x; \end{array}$$

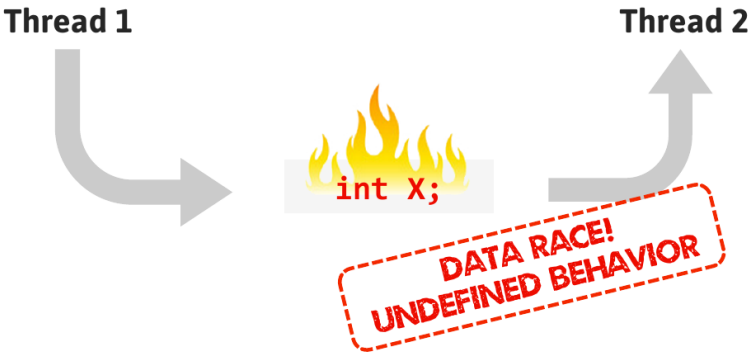
$$\text{assert}\,(r1 == 1 \,||\, r2 == 1);$$

Processor 1	Processor 2
mov r1, [Y]	mov r2, [X]
mov [X], 1	mov [Y], 1

https://preshing.com/20120515/memory-reordering-caught-in-the-act/

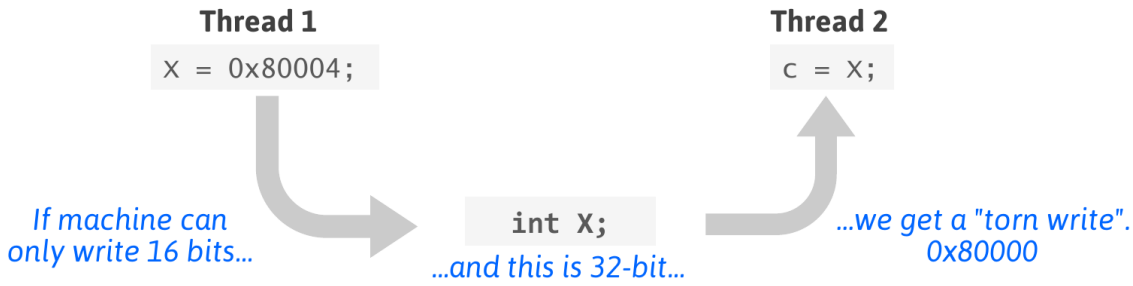
Real Hardware/Compiler

Real hardware doesn't run the code that you wrote.
Real compiler doesn't produce the code that you wrote.



Real Hardware/Compiler

Real hardware doesn't run the code that you wrote.
Real compiler doesn't produce the code that you wrote.

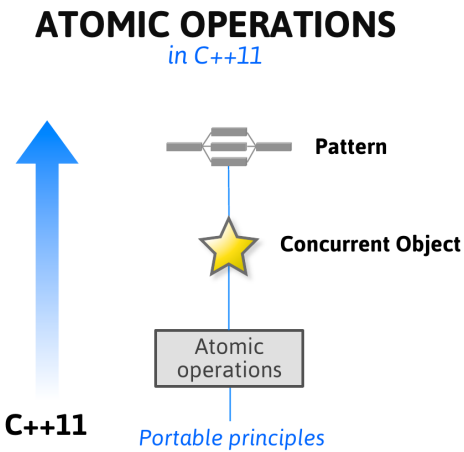


Real Hardware/Compiler

Real hardware doesn't run the code that you wrote.
Real compiler doesn't produce the code that you wrote.
Dekkers Algorithm, g++ -O2

```
owner@vmTest:~/src$ cat dekker.cpp

int flag;
int flag2;
int data;

int main() {
        if(flag2 == 0)
                data = 42;
        flag = 1;
        if(flag2 == 0)
                return 0;
```

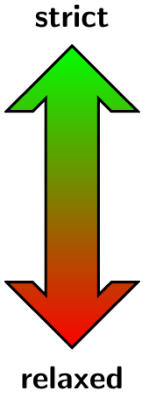
```
main:
.LFB0:
        .cfi_startproc
        movl    flag2, %eax     read flag2
        testl   %eax, %eax      comp flag2
        jne     .L2
        movl    $42, data
.L2:
        movl    $1, flag        write flag
        xorl    %eax, %eax
        ret
        .cfi_endproc
```



History

- In 2011, new versions of the ISO standards for C and C++, informally known as C11 and C++11, were ratified.
- These standards define a memory model for C/C++
- Support for this model has recently become available in popular compilers (GCC 4.4, Intel C++ 13.0, MSVC 11.0, Clang 3.1).

C++11 Memory Model



seq_cst

release /acquire

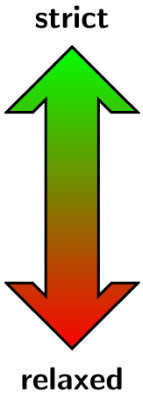
release /consume

relaxed

Why C++11 Memory Model

- 在 C++11 之前，其实是没有定义内存模型的，我们所使用的都是一些处理器/编译器暴露的同步原语，比如 GCC 的内联汇编，内建函数之类的, 有着不同的实现。
- C++11 在标准库中引入了 memory model 的意义在于在 High Level Language 层面实现对在多处理器中多线程共享内存的访问，实现跨编译器,OS 和硬件的差异性。

C++11 Memory Model



seq_cst

release /acquire

release /consume

relaxed

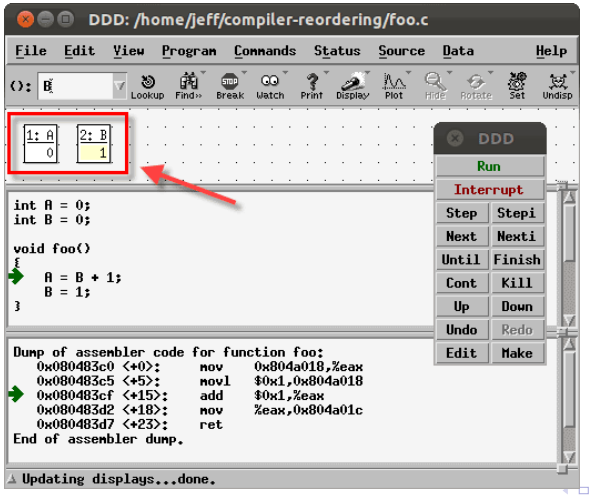
Happens-before

- An important fundamental concept in understanding the memory model
- A guarantee that memory writes by one specific statement are visible to another specific statement

Happens-before

```
int A = 0;
int B = 0;

void foo()
{
    A = B + 1;                  // (1)
    B = 1;                      // (2)
}
```

Happens-before



Happens-before

```
int A, B;

void foo()
{
    A = B + 1;
    B = 0;
}
```

Happens-before

GCC 4.6.1

```
$ gcc -S -masm=intel foo.c
$ cat foo.s
    ...
    mov     eax, DWORD PTR _B  (redo this at home...)
    add     eax, 1
    mov     DWORD PTR _A, eax
    mov     DWORD PTR _B, 0
    ...
```

Happens-before

GCC 4.6.1

```
$ gcc -O2 -S -masm=intel foo.c
$ cat foo.s
    ...
    mov     eax, DWORD PTR B
    mov     DWORD PTR B, 0
    add     eax, 1
    mov     DWORD PTR A, eax
    ...
```

Happens-before

GCC 4.6.1

```
int A, B;

void foo()
{
    A = B + 1;
    asm volatile("" ::: "memory");
    B = 0;
}
```

Happens-before

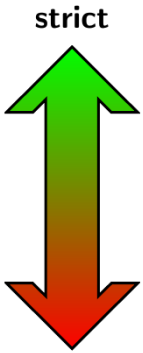
GCC 4.6.1

```
$ gcc -O2 -S -masm=intel foo.c
$ cat foo.s
    ...
    mov     eax, DWORD PTR _B
    add     eax, 1
    mov     DWORD PTR _A, eax
    mov     DWORD PTR _B, 0
    ...
```

Happens-before

```
//x86
#define COMPILER_BARRIER() asm volatile("" ::: "memory")
//PowerPC
#define RELEASE_FENCE() asm volatile("lwsync" ::: "memory")
================================================
int Value;
std::atomic<int> IsPublished(0);
void sendValue(int x)
{
    Value = x;
    // <-- reordering is prevented here!
    IsPublished.store(1, std::memory_order_release);
}
```

C++11 Memory Model



seq_cst

release / acquire

release / consume

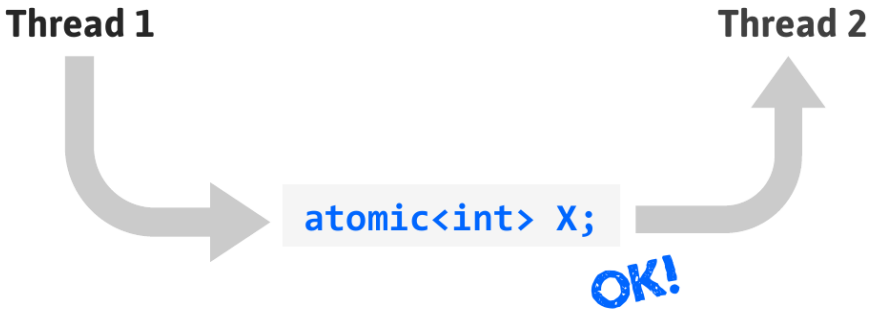
relaxed

std :: atomic<T>

- x.load(memory order)
- x.store (T, memory order)

Concurrent accesses on atomic locations do not race. The memory order argument specifies ordering constraints between atomic and non-atomic memory accesses in different threads.

If multiple threads access the same variable concurrently, and at least one thread modifies it, all threads must use C++11 atomic operations.



std :: atomic<T>

```
#include <atomic>

std::atomic_int flag;
std::atomic_int flag2;
int data;

int main() {
    if(flag2 == 0)


        data = 42;

    flag = 1;

    if(flag2 == 0)

        return 0;
}
```

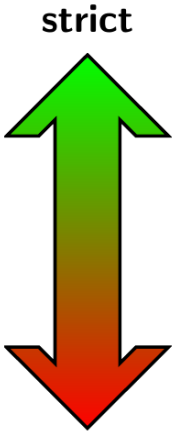
```
main:
.LFB329:
        .cfi_startproc
        movl    flag2(%rip), %eax
        testl   %eax, %eax
        jne     .L2
        movl    $42, data(%rip)
.L2:
        movl    $1, flag(%rip)
        mfence
        movl    flag2(%rip), %eax
        xorl    %eax, %eax
        ret
        .cfi_endproc
```

std :: atomic<T>

```
#include <atomic>

std::atomic_int flag(0);
std::atomic_int flag2;
int data;

int main() {
    if(flag2 == 0)


        data = 42;

    flag++;
    if(flag2 == 0)

        return 0;
}
```

```
main:
.LFB329:
        .cfi_startproc
        movl    flag2(%rip), %eax
        testl   %eax, %eax
        jne     .L2
        movl    $42, data(%rip)
.L2:
        lock addl       $1, flag(%rip)
        movl    flag2(%rip), %eax
        xorl    %eax, %eax
        ret
        .cfi_endproc
```

std :: memory_order



seq_cst	total order (SC for DRF code)
release / acquire	message passing
release / consume	message passing
relaxed	no synchronization

std :: memory_order_seq_cst

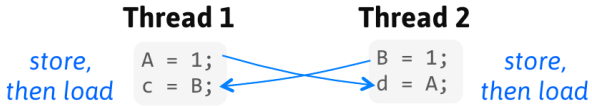
There is a total order over all seq cst operations. This order contributes to inter-thread ordering constraints.

Example (Dekker's algorithm)

```
atomic_int x(0); atomic_int y(0);

x.store(1, seq_cst);          ||  y.store(1, seq_cst);
int r1 = y.load(seq_cst);     ||  int r2 = x.load(seq_cst);

assert(r1 == 1 || r2 == 1);
```

std :: memory_order_seq_cst

```
atomic<int> A(0);
atomic<int> B(0);
```



Possible Interleavings:

```
A = 1;
c = B;
B = 1;
d = A;
```

```
A = 1;
B = 1;
c = B;
d = A;
```

```
B = 1;
d = A;
A = 1;
c = B;
```

c	d	
~~0~~	~~0~~	*Impossible!*
0	1	
1	0	
1	1	

std :: memory_order_seq_cst

sequential consistency

直观上，读操作应该返回“最后”一次写入的值。

- 在单处理器系统中，“最后”由程序次序定义。
- 在多处理器系统中，我们称之为顺序连贯 (sequential consistency, SC).

约束条件

- 在每个处理器内，维护每个处理器的程序次序；
- 在所有处理器间，维护单一的表征所有操作的次序。对于写操作 W1, W2, 不能出现从处理器 P1 看来，执行次序为 W1->W2; 从处理器 P2 看来，执行次序却为 W2->W1 这种情况。

std :: memory_order_release/acquire

An acquire load makes prior writes to other memory locations made by the thread that did the release visible in the loading thread.

Example (message passing)

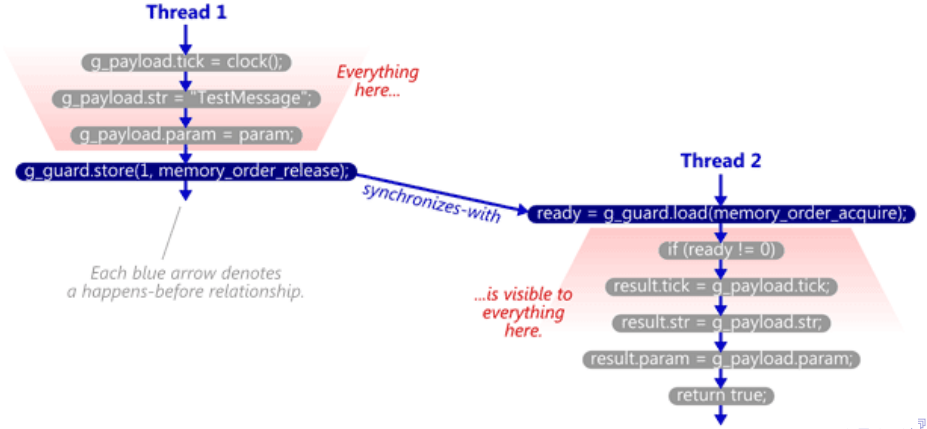
```
int data(0); atomic_bool flag(false);
```

```
// sender
data = 42;
flag.store(true, release);
```

```
// receiver
while (!flag.load(acquire))
    {};
assert(data == 42);
```

std :: memory_order_release/acquire

An acquire load makes prior writes to other memory locations made by the thread that did the release visible in the loading thread.



std :: memory_order_release/acquire

Acquire semantics

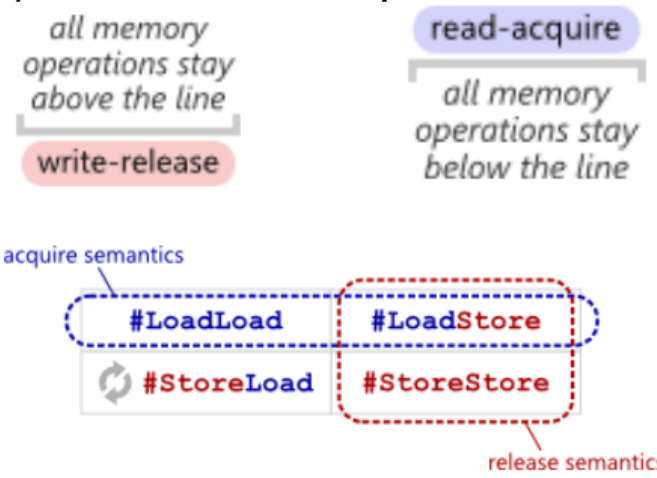
Acquire semantics is a property that can only apply to operations that read from shared memory, whether they are read-modify-write operations or plain loads. The operation is then considered a **read-acquire**. Acquire semantics prevent memory reordering of the read-acquire with any read or write operation that follows it in program order.

std :: memory_order_release/acquire

Release semantics

Release semantics is a property that can only apply to operations that write to shared memory, whether they are read-modify-write operations or plain stores. The operation is then considered a write-release. Release semantics prevent memory reordering of the write-release with any read or write operation that precedes it in program order.

std :: memory_order_release/acquire



std :: memory_order_release/acquire

```
void produce() {
    payload = 42;
    guard.store(1, std::memory_order_release)
}
void consume(int iterations) {
    for(int i = 0; i < iterations; i++){
      if(guard.load(std::memory_order_acquire))
        result[i] = payload;
    }
}
```

std :: memory_order_release/acquire

Intel x86

```
mov     ecx, dword ptr [rip + _Guard]
test    ecx, ecx
cmovne  eax, dword ptr [rip + _Payload]
```

load from Guard

load from Payload

ARM V7

```
add     r3, pc
ldr.w   r4, [r9]
dmb     ish
ldr     r5, [r3]
cmp     r4, #0
it      ne
movne   r2, r5
```

load from Guard

memory barrier

load from Payload

PowerPC

```
lis     r8, Guard@ha
addi    r8, r8, Guard@l
lis     r7, Payload@ha
lwz     r9, 0(r8)
cmpw    cr7, r9, r9
bne-    cr7, $+4
isync
cmpwi   cr7, r9, 0
beq-    cr7, .L0
lwz     r10, Payload@l(r7)
```

load from Guard

memory barrier

load from Payload

http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

std :: memory_order_release/acquire

1000 iterations:

Intel x86: strong memory model
implicit acquire-release consistency

~~#LoadLoad~~	~~#LoadStore~~
#StoreLoad	~~#StoreStore~~

ARM v7, PowerPC: weak memory model
casual consistency
needs memory barriers for acquire-release consistency

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

std :: memory_order_consume

(**Data Dependency Ordering**)A consume load makes prior writes to data-dependent memory locations made by the thread that did the release visible in the loading thread.

Data Dependency

C++ Source Code Level

```
a = Guard.load(memory_order_consume);
b = *a;
c = *b;
```

carries-a-dependency

Machine Instruction Level

```
lis     r8, Guard@ha
la      r8, Guard@l(8)
lwz     r9, 0(r8)
lwz     r10, 0(r9)
lwz     r11, 0(r10)
```

std :: memory_order_consume

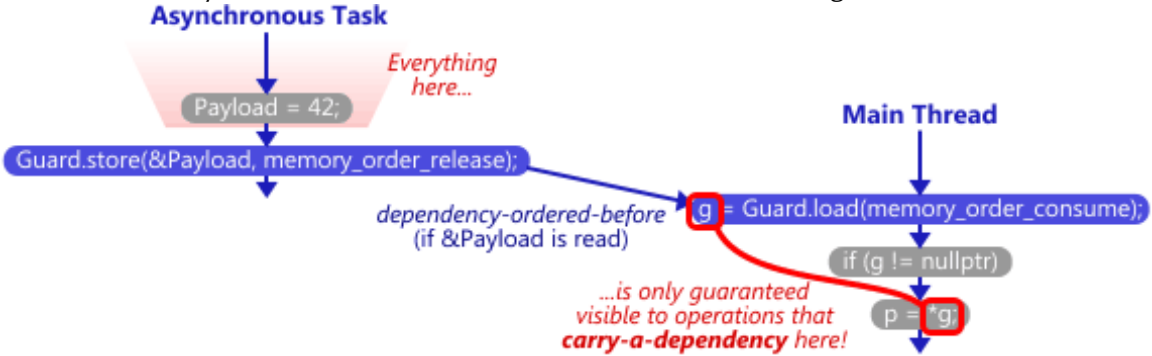
(**Data Dependency Ordering**)A consume load makes prior writes to data-dependent memory locations made by the thread that did the release visible in the loading thread.

Example (message passing)

```
int data(0); atomic<int*> p(0);

// sender                          // receiver
data = 42;                         while (p.load(consume) == 0)
p.store(&data, release );              {};
                                   assert(*p == 42);
```

std :: memory_order_consume

(**Data Dependency Ordering**)A consume load makes prior writes to data-dependent memory locations made by the thread that did the release visible in the loading thread.



std :: memory_order_consume

- x86-64 machine code loads Guard into register rcx, then, if rcx is not null, uses rcx to load the payload, thus creating a data dependency between the two load instructions.
- 86-64’ s strong memory model already guarantees that loads are performed in-order, even if there isn’ t a data dependency.

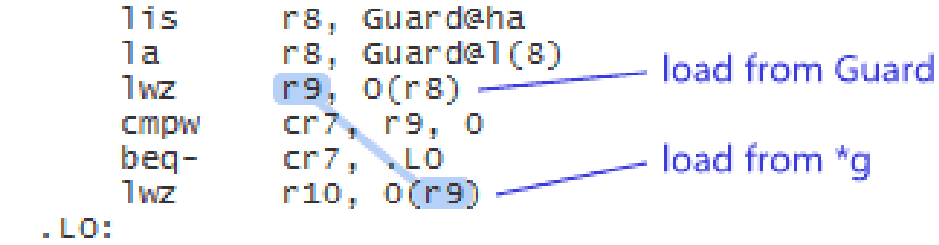
```
    mov     rcx, qword ptr [rip + _Guard]
    test    rcx, rcx
    je      L0
    mov     eax, dword ptr [rcx]
L0:
```

load from Guard

load from *g

std :: memory_order_consume

- PowerPC machine code loads Guard into register r9, then uses r9 to load the payload, thus creating a data dependency between the two load instructions.
- completely avoid the "*cmp;bne;isync*" sequence of instructions that formed a memory barrier in the original example, while still ensuring that the two loads are performed in-order.



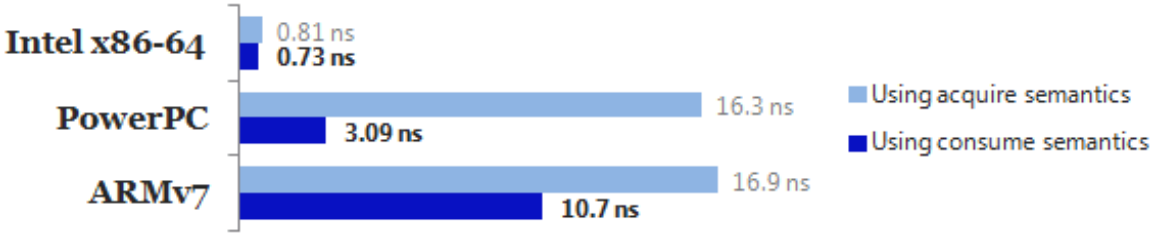
std :: memory_order_consume

- ARMV7 machine code loads Guard into register r4, then uses r4 to load the payload, thus creating a data dependency between the two load instructions.
- completely avoid the "*dmb ish* " instruction that was present in the original example, while still ensuring that the two loads are performed in-order.

```
    movw    r3, :lower16:(_Guard-(L0+4))
    movt    r3, :upper16:(_Guard-(L0+4))
L0:
    add     r3, pc
    ldr     r4, [r3]
    cmp     r4, #0
    it      ne
    ldrne   r2, [r4]
```

std :: memory_order_consume

(**Data Dependency Ordering**)A consume load makes prior writes to data-dependent memory locations made by the thread that did the release visible in the loading thread.



std :: memory_order_consume

real world – RCU

在真实世界中使用这一技术–利用数据依赖顺序以避免内存栅栏的例子就是 Linux 内核。Linux 提供了读-复制-更新（RCU）的同步机制，适合构建在多个线程中需要多读少写的共享变量 (包括指针等) 数据结构。Linux 实际上没有使用 C++11 消费语义来去除那些内存栅栏，而是依靠它自己的 API 和规范。其实起初 RCU 就被看作是给 C++11 添加消费语义的动机。

std :: memory_order_relaxed

The memory_order_relaxed ensure these operations are atomic, but don't impose any ordering constraints/memory barriers that aren't already there.

relaxed order 允许单线程上不同的 memory location 进行 reorder，但是对于同一个 memory location 不能进行 reorder。

```
atomic<int> A(0);
atomic<int> B(0);
```

Thread 1

```
A.store(1, memory_order_relaxed);
c = B.load(memory_order_relaxed);
```

Thread 2

```
B.store(1, memory_order_relaxed);
d = A.load(memory_order_relaxed);
```

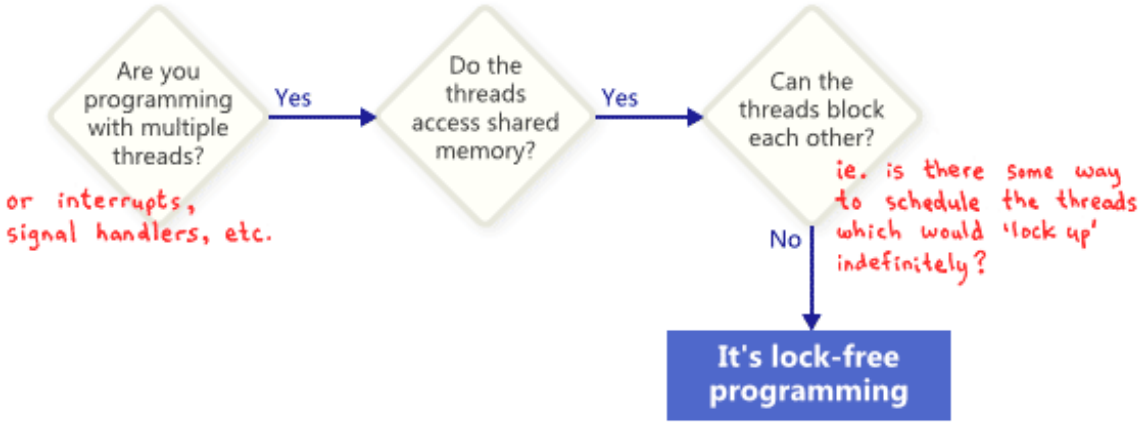
Doing the same thing

You can prevent it with "full memory fences":

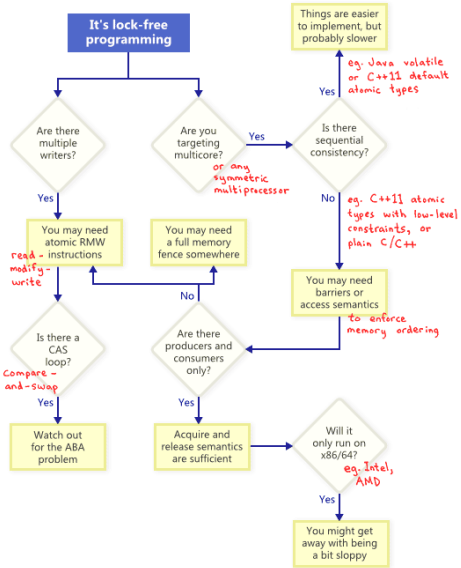
```
atomic_thread_fence(memory_order_seq_cst);
```

c	d	
0	0	*Possible!*
0	1	
1	0	
1	1	

Lock-Free Programming



Lock-Free Programming



Lock-free Stack

```
template<typename T>
class lock_free_stack
{
    private:
        struct node
        {
            T data;
            node* next;
            node(T const& data_):  // 1
                data(data_)
        {}
    };
```

Lock-free Stack

```
std::atomic<node*> head;

public:
void push(T const& data)
{
    node* const new_node=new node(data); // 2
    new_node->next=head.load();  // 3
    while(!head.compare_exchange_weak(new_node->next,new_node));  // 4
}
```

使用“比较/交换”操作：返回 false 时，因为比较失败 (例如，head 被其他线程修改)，会使用 head 中的内容更新 new_node->next(第一个参数) 的内容。

Lock-free Stack

```
while(!head.compare_exchange_weak(new_node->next,new_node));  // 4

    if ( head == new_node->next){
        head = new_node;
        return true;
    }
    else{
        new_node->next = head;
        return false;
    }
```

使用“比较/交换”操作：返回 false 时，因为比较失败 (例如，head 被其他线程修改)，会使用 head 中的内容更新 new_node->next(第一个参数) 的内容。

Lock-free Stack

```
void pop(T& result)
{
    node* old_head=head.load();
    while(!head.compare_exchange_weak(old_head,old_head->next));
    result=old_head->data;
}
```

这段代码很优雅，但有关于节点泄露的两个问题。首先，这段代码在空链表时不工作：当 head 指针是空指针时，要访问 next 指针时，将引起未定义行为。