内存屏障介绍

问题

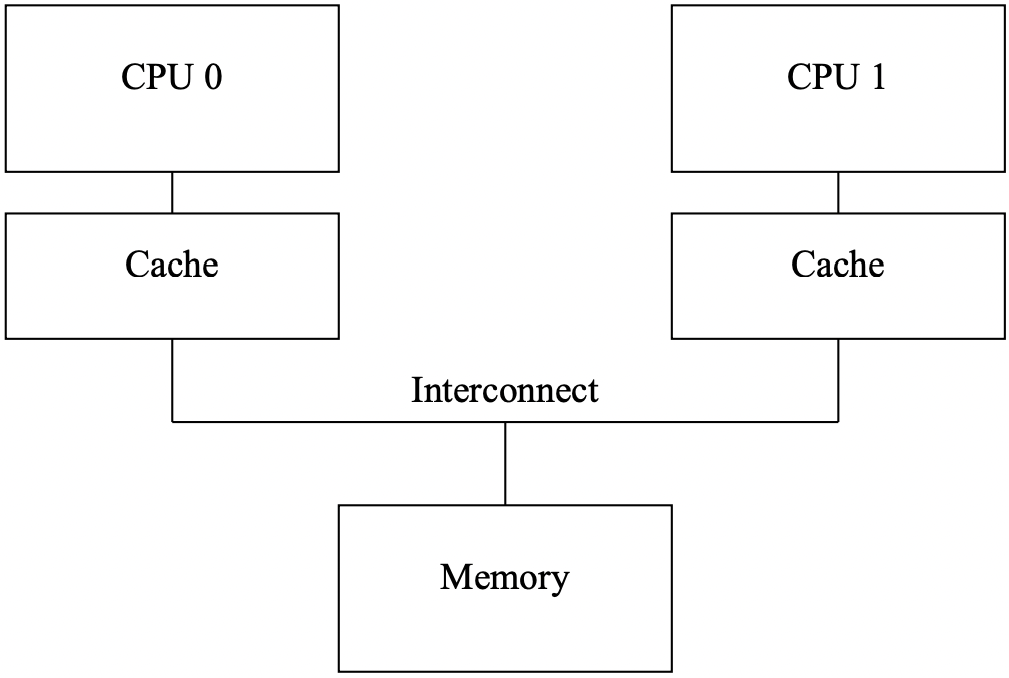
下面代码中的assert()是否会失败？为什么？

C | 复制代码

```c
int a = 0;
int b = 0;

// Thread A
void foo(void) {
    a = 1;
    b = 1;
}

// Thread B
void bar(void) {
    while (b == 0) continue;
    assert(a == 1);
}

```

多处理器体系

多CPU系统架构图中，不同CPU具有不同的内存视角，CPU实际访问内存的顺序跟代码顺序不一致（**前提：没有地址依赖**）。主要是CPU Cache、Inter-processor Communication、Device的实现决定的：https://www.puppetmastertrading.com/images/hwViewForSwHackers.pdf 。



简单来说就是为了提升性能。因为CPU并不知道上层的代码逻辑，很难判断独立内存地址之间的关系，所以提供单独的指令让软件工程师自己决定。

并且不同的CPU架构对memory ordering提供了不同程序的保证：

Type	Alpha	ARMv7	MIPS	RISC-V		PA-RISC	POWER	SPARC			x86	AMD64	IA-64	z/Architecture
				WMO	TSO			RMO	PSO	TSO				
Loads can be reordered after loads	Y	Y	depend on implem	Y		Y	Y	Y					Y	
Loads can be reordered after stores	Y	Y		Y		Y	Y	Y					Y	
Stores can be reordered after stores	Y	Y		Y		Y	Y	Y	Y				Y	
Stores can be reordered after loads	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Atomic can be reordered with loads	Y	Y	e nt at io n	Y			Y	Y					Y	
Atomic can be reordered with stores	Y	Y		Y			Y	Y	Y				Y	
Dependent loads can be reordered	Y													
Incoherent instruction cache/pipeline	Y	Y		Y	Y		Y	Y	Y	Y	Y		Y	

此外编译器也会做优化导致实际指令顺序发生变化。所以总结如下有两个原因或导致上述的结果。

- 编译优化
- 硬件乱序

内存屏障

如上所述，独立的内存操作实际上是以随机顺序执行的，但这可能会对CPU–CPU互动和I/O造成问题。需要的是某种干预方式，以指示编译器和CPU限制顺序。内存屏障就是这样的干预方式，它们在屏障两侧的内存操作上施加了一种感知到的部分顺序。这种强制执行非常重要，因为系统中的CPU和其他设备可以使用各种技巧来提高性能，包括重新排序、延迟和组合内存操作、推测加载、推测分支预测和各种类型的缓存。内存屏障用于覆盖或抑制这些技巧，使代码可以合理地控制多个CPU和/或设备之间的交互。

- Write memory barriers

写内存屏障保证在屏障之前指定的所有写操作将在屏障之后指定的所有写操作之前出现。写屏障仅对写进行部分排序，不需要对读产生任何影响。可以将CPU视为随着时间的推移向内存系统提交一系列存储操作。在写屏障之前的所有存储操作将在写屏障之后的所有存储操作之前发生。[！]请注意，写屏障通常应与读屏障配对使用

- Read memory barriers

在屏障之前指定的所有读操作将在屏障之后指定的所有读操作之前出现。读屏障仅对读进行部分排序，不需要对写产生任何影响。[！]请注意，读屏障通常应与写屏障配对使用。

- General memory barriers

通用内存屏障保证在屏障之前指定的所有读和写操作将在屏障之后指定的所有读和写操作之前出现，关于系统的其他组件。通用内存屏障是对加载和存储进行部分排序。通用内存屏障暗示读和写内存屏障，因此可以替换任何一个。

没有屏障

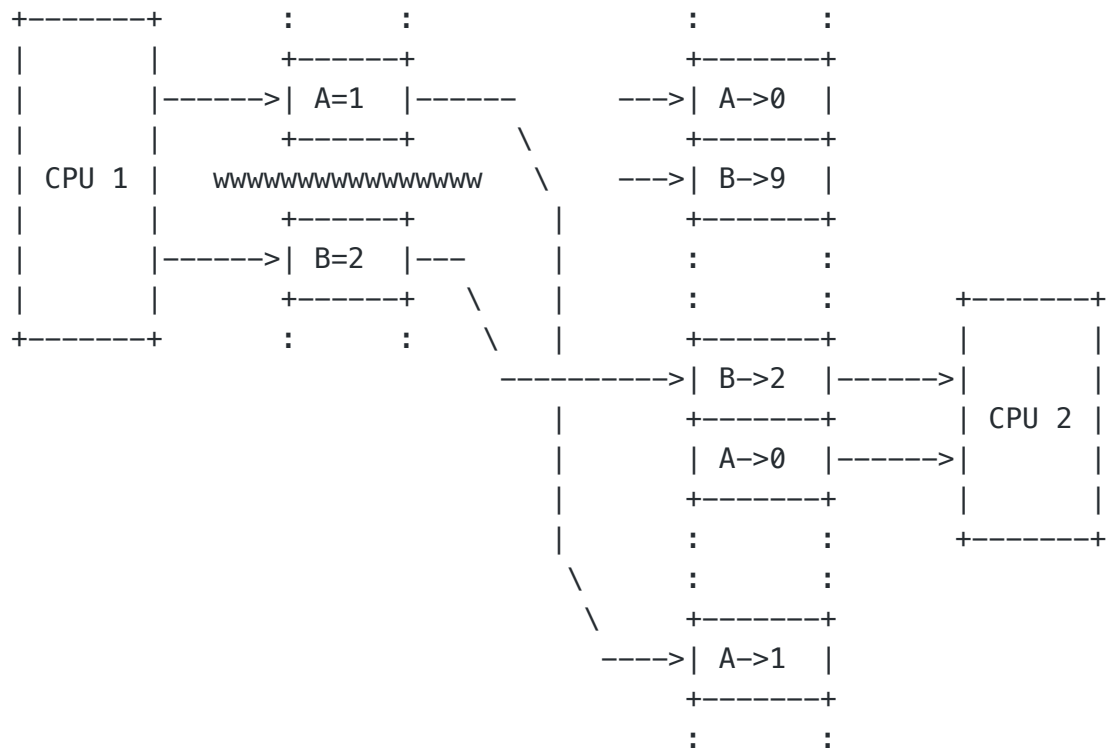
Plain Text | 复制代码

```
        CPU 1                   CPU 2
        ======================= =======================
                { A = 0, B = 9 }
        STORE A=1
        STORE B=2
                                LOAD B
                                LOAD A

Without intervention, CPU 2 may then choose to perceive the events on CPU 1 in
some effectively random order:

        +-------+       :      :                :       :
        |       |       +------+                +-------+
        |       |------>| A=1  |------      --->| A->0  |
        |       |       +------+      \         +-------+
        | CPU 1 |       |      |       \    --->| B->9  |
        |       |       +------+        |       +-------+
        |       |------>| B=2  |---     |       :       :
        |       |       +------+   \    |       :       :       +-------+
        +-------+       :      :    \   |       +-------+       |       |
                                     ---------->| B->2  |------>|       |
                                        |       +-------+       | CPU 2 |
                                        |       | A->0  |------>|       |
                                        |       +-------+       |       |
                                        |       :       :       +-------+
                                         \      :       :
                                          \     +-------+
                                           ---->| A->1  |
                                                +-------+
                                                :       :
```

写屏障

Plain Text | 复制代码

```
Consider the following sequence of events:

        CPU 1                   CPU 2
        ======================= =======================
               { A = 0, B = 9 }
        STORE A=1
        <write barrier>
        STORE B=2
                                LOAD B
                                LOAD A

Without intervention, CPU 2 may then choose to perceive the events on CPU 1 in
some effectively random order, despite the write barrier issued by CPU 1:
```

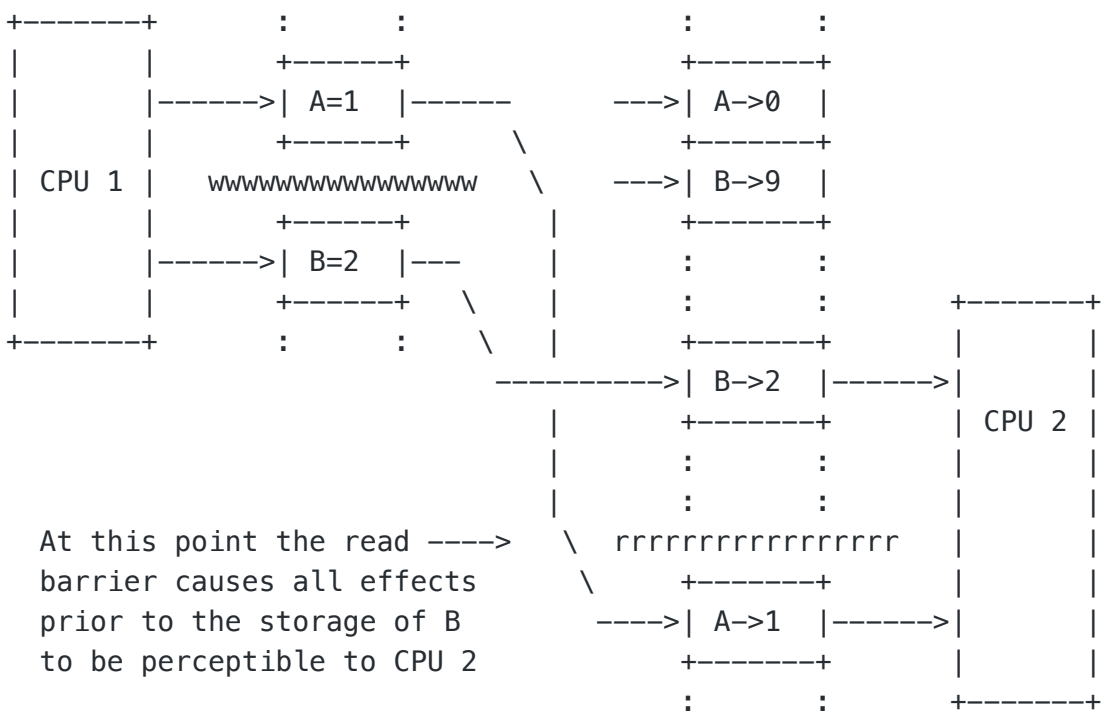


读写配对

Plain Text | 复制代码

```
        CPU 1                   CPU 2
        ======================= =======================
                { A = 0, B = 9 }
        STORE A=1
        <write barrier>
        STORE B=2
                                LOAD B
                                <read barrier>
                                LOAD A
```

then the partial ordering imposed by CPU 1 will be perceived correctly by CPU 2:



内核相关

- 编译器屏障

编译器屏障barrier()。比如单线程，但是可能有中断发生的场景。

- CPU内存屏障

显形：

- mb() smp_mb()
- wmb() smp_wmb()
- rmb() smp_rmb()

隐性：

- spin locks
- R/W spin locks
- mutexes
- semaphores
- R/W semaphores

注意所有的CPU屏障自带编译器屏障。此外还有其他一些特定的内存屏障，例如dma、pmem等，特定领域才需要关注这里不再赘述。

Rust相关

为了屏蔽底层差异，C++标准对内存顺序进行了抽象。Rust也复用了这个标准如下：

Relaxed 宽松	最宽松的规则，它对编译器和 CPU 不做任何限制，可以乱序
Release 释放	保证Release之前的写操作永远在它之前，但是它后面的操作可能被重排到它前面
Acquire 获取	保证在它之后的访问永远在它之后，但是它之前的操作却有可能被重排到它后面，往往和Release在不同线程中联合使用
AcqRel 获取释放	可以认为是 Release + Acquire，用在compare_and_swap，比较少见
SeqCst 顺序一致	保证SeqCst操作前的数据操作绝对不会被重新排在该SeqCst操作之后，且该SeqCst操作后的数据操作也绝对不会被重新排在SeqCst操作前。

- atomics

具体见std::sync::atomicstd::sync::atomic

- fence和compiler_fence

具体见std::sync::atomic::fence 和 std::sync::atomic::compiler_fence

所以到底该怎么用?

所以根据单线程一般不用考虑。但是多线程场景，尤其是以下场景，需要假设独立地址之间的内存操作顺序是随机的、并行的。

> It has to be assumed that the conceptual CPU is weakly-ordered but that it will maintain the appearance of program causality with respect to itself. Some CPUs (such as i386 or x86_64) are more constrained than others (such as powerpc or frv), and so the most relaxed case (namely DEC Alpha) must be assumed outside of arch-specific code. This means that it must be considered that the CPU will execute its instruction stream in any order it feels like - or even in parallel - provided that if an instruction in the stream depends on an earlier instruction, then that earlier instruction must be sufficiently complete[*] before the later instruction may proceed; in other words: provided that the appearance of causality is maintained. [*] Some instructions have more than one effect - such as changing the condition codes, changing registers or changing memory - and different instructions may depend on different effects. A CPU may also discard any instruction sequence that winds up having no ultimate effect. For example, if two adjacent instructions both load an immediate value into the same register, the first may be discarded. Similarly, it has to be assumed that compiler might reorder the instruction stream in any way it sees fit, again provided the appearance of causality is maintained.

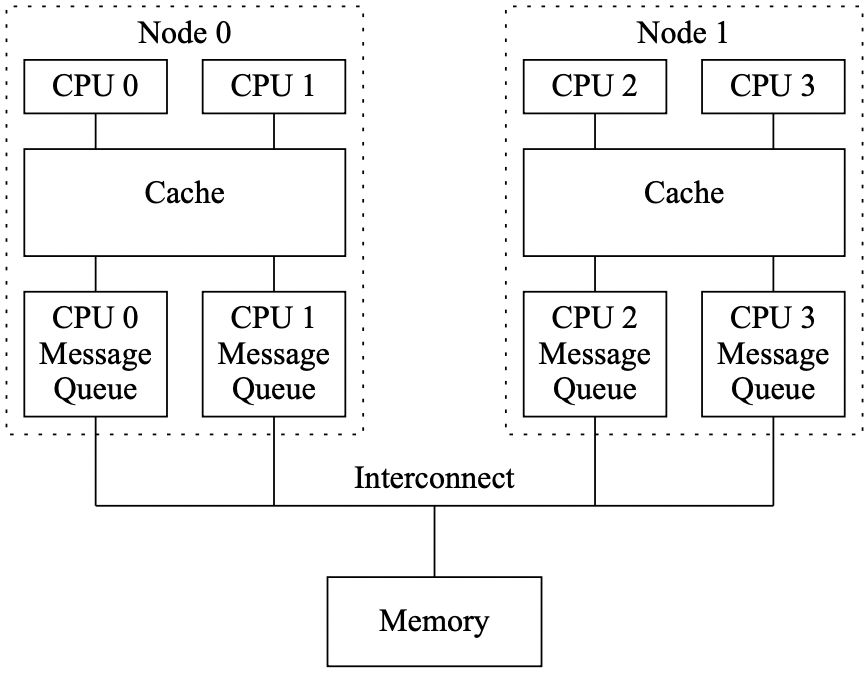
- 内核开发

(*) Interprocessor interaction. (*) Atomic operations. (*) Accessing devices. (*) Interrupts.

- 应用开发

凡是涉及多线程共享，但又没有用锁把临界区保护起来。比如rust的atomics的部分使用场景，简单来说就是除了对atomic变量本身感兴趣，还针对atomic做了判断并对随之二来的其他地址的共享内存有依赖。

需要注意的是：Release和Acquire要成对出现，单独使用是无效的。具体要看缓存一致性协议的实现。简单理解可以看下面的图：



一些误解

1. 既然x86_64是强一致模型，是否可以不考虑memory ordering？只有aarch64等cpu架构才需要考虑

所以说即便是最常用的x86_64架构，也无法保证所有情况下的顺序一致性，见 Intels Developer Manual, chapter 8.2.2:

可以保证：

> - Reads are not reordered with other reads.
> - Writes are not reordered with older reads.
> - Writes to memory are not reordered with other writes (with some exceptions)

但无法保证:

> - Reads may be reordered with older writes to different locations but not with older writes to the same location

2. 在使用了memory barrier之后，用了可以立即读到barrier前面的值?

不能，这只是一种顺序上的保证，内存屏障的指定执行完并不意味着立即能读到数据；反之，如果没有使用memory barrier，也不会永远读不到那个值，而是在一定时间内一定会读到

一些实例

内核的环形队列实现：https://www.kernel.org/doc/Documentation/core-api/circular-buffers.rst

问题答案

C | 复制代码

```c
int a = 0;
int b = 0;

// Thread A
void foo(void) {
    a = 1;
    b = 1;
}

// Thread B
void bar(void) {
    while (b == 0) continue;
    assert(a == 1);
}

```

应该改为：

```c
int a = 0;
int b = 0;

// Thread A
void foo(void) {
    a = 1;
    smp_wmb();
    b = 1;
}

// Thread B
void bar(void) {
    while (b == 0) continue;
    smp_rmb();
    assert(a == 1);
}

```

其它参考文献

https://www.kernel.org/doc/Documentation/memory-barriers.txt

https://cfsamsonbooks.gitbook.io/explaining-atomics-in-rust/

https://www.stroustrup.com/abstraction-and-machine.pdf

https://www.puppetmastertrading.com/images/hwViewForSwHackers.pdf