

多核多处理器与性能可扩展性

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
 - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
 - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

Review: 如何解决死锁？

解决死锁

出问题再处理：死锁的检测与恢复

设计时避免：死锁预防

运行时避免死锁：死锁避免

Review: 死锁预防：四个方向

1. 避免互斥访问：通过其他手段（如代理执行）
2. 不允许持有并等待：一次性申请所有资源
3. 资源允许抢占：需要考虑如何恢复
4. 打破循环等待：按照特定顺序获取资源

Review: 不同优先级反转解决方案对比

- 不可打断临界区协议 (Non-preemptive Critical Sections, NCP)

进入临界区后**不允许其他进程打断**：禁止操作系统调度

- 优先级继承协议 (Priority Inheritance Protocol, PIP)

高优先级进程被阻塞时，继承给锁持有者**自己的优先级**：锁给操作系统调度hint

- 即时优先级置顶协议 (Immediate Priority Ceiling Protocols, IPCP)

获取锁时，给持有者该锁竞争者中**最高优先级**：锁给操作系统调度hint

- 原生优先级置顶协议 (Original Priority Ceiling Protocols, OPCP)

高优先级进程被阻塞时，给锁持有者该锁竞争者中**最高优先级**：锁给操作系统调度hint

Review: 不同优先级反转解决方案对比

- 不可打断临界区协议 (Non-preemptive Critical Sections, NCP)

易实现，但会阻塞系统正常运行（更高优先级的程序正常执行）

- 优先级继承协议 (Priority Inheritance Protocol, PIP)

难实现，且每次有更高优先级的竞争者出现时都会被打断然后重新继承

- 即时优先级置顶协议 (Immediate Priority Ceiling Protocols, IPCP)

易实现，但需要知道有哪些竞争者会竞争锁。直接给最高与NCP相同

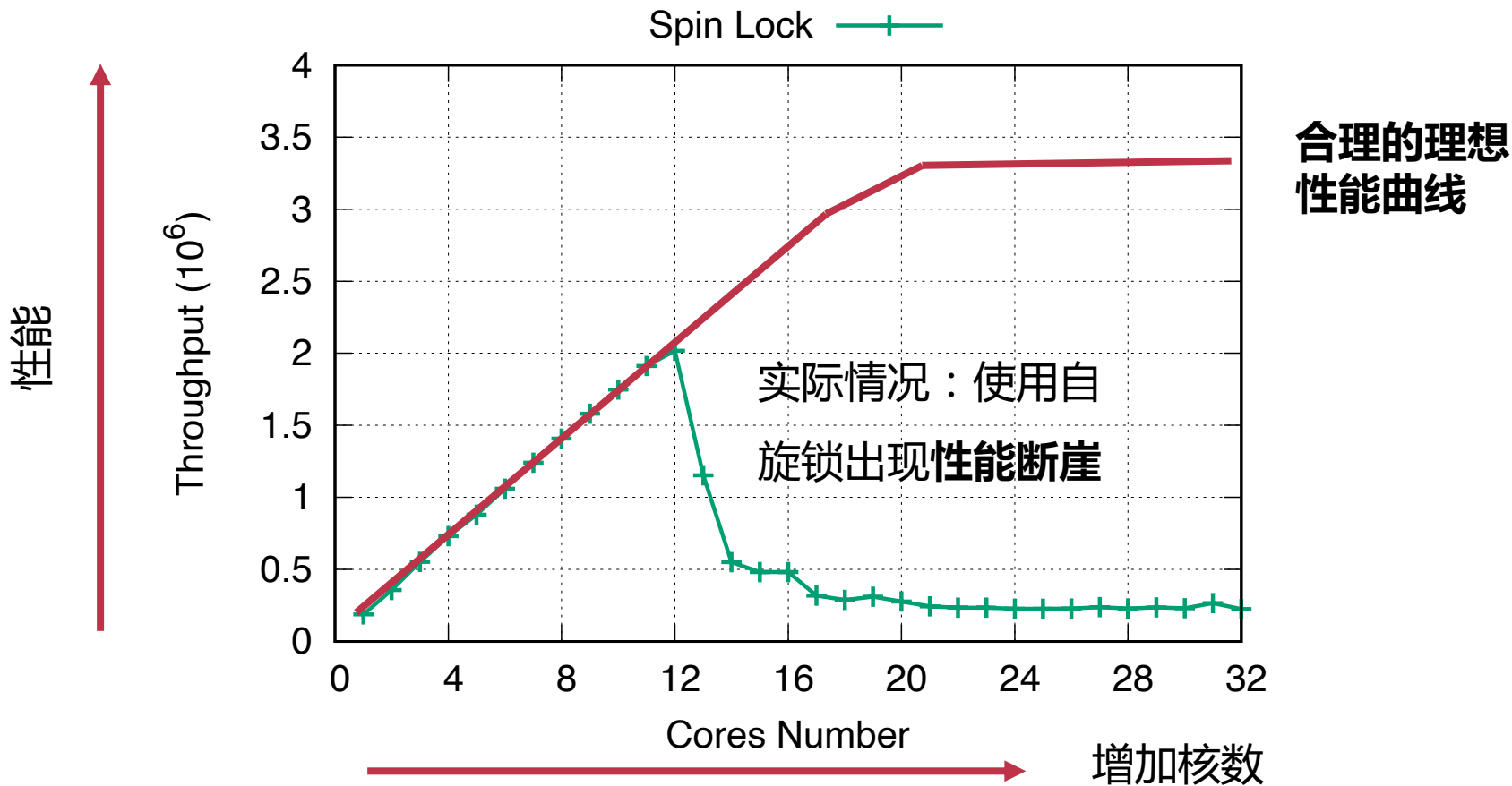
- 原生优先级置顶协议 (Original Priority Ceiling Protocols, OPCP)

难实现，需要知道有哪些竞争者会竞争锁，一旦发生置顶便不会再被其他竞争者打断

Review: 互斥锁微基准测试

```
struct lock *glock;
unsigned long gcnt = 0;
char shared_data[CACHE_LINE_SIZE];
void *thread_routine(void *arg) {
    while(1) {
        lock(glock);
        /* Critical Section */
        gcnt = gcnt + 1;
        /* Read Modify Write 1 * shared cacheline */
        visit_shared_data(shared_data, 1);
        unlock(glock);
        interval();
    }
}
```

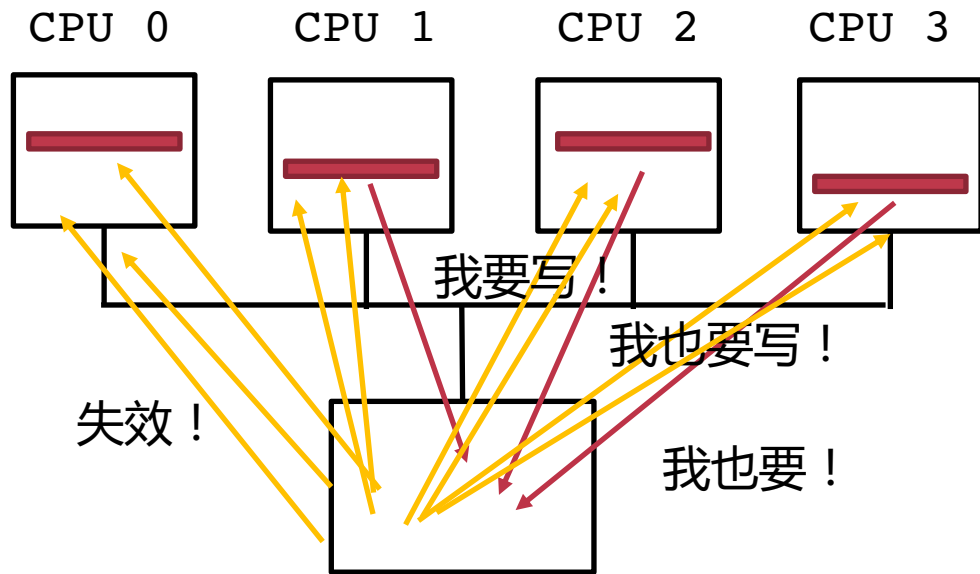
Review: 可扩展性断崖



Review: 可扩展性断崖背后的原因

```
void lock(int *lock) {  
    while(atomic_CAS(lock, 0, 1)  
        != 0)  
        /* Busy-looping */;  
}  
  
void unlock(int *lock) {  
    *lock = 0;  
}
```

自旋锁实现



全局目录项

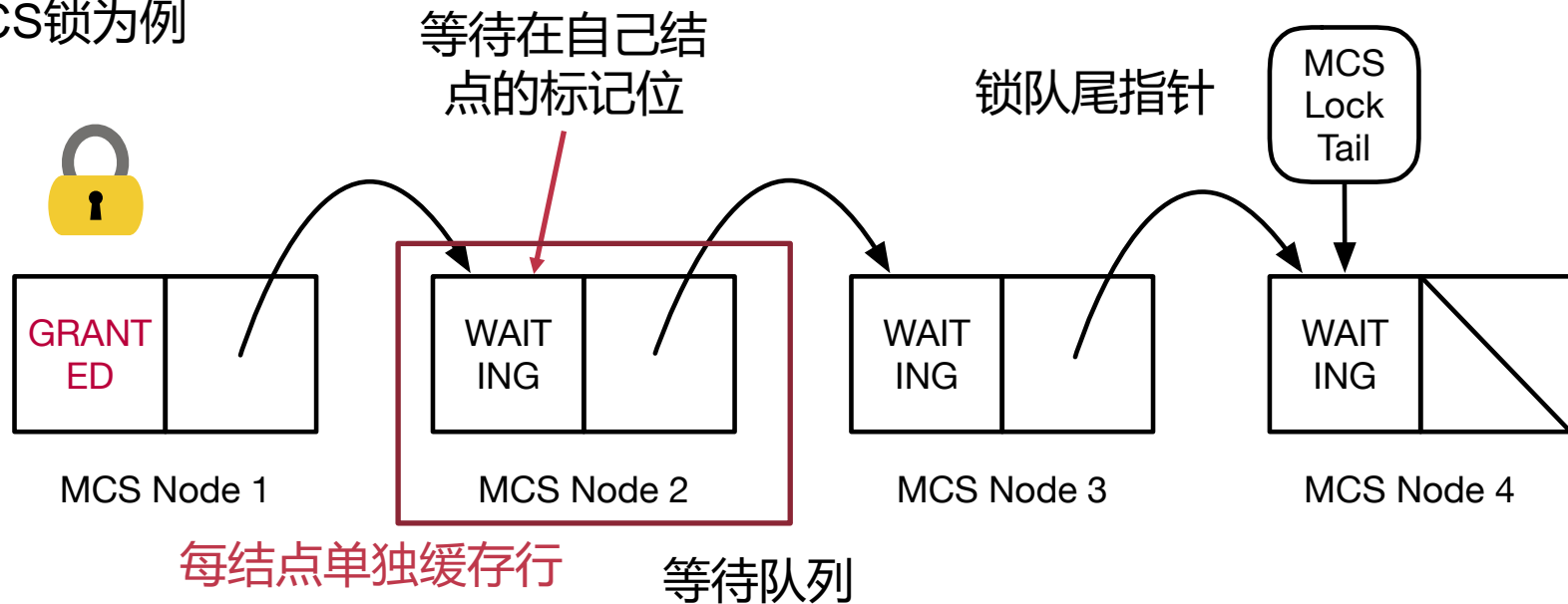
Lock所在缓存行状态

对**单一缓存行**的竞争导致**严重的性能**开销

Review: 解决可扩展性问题：MCS锁

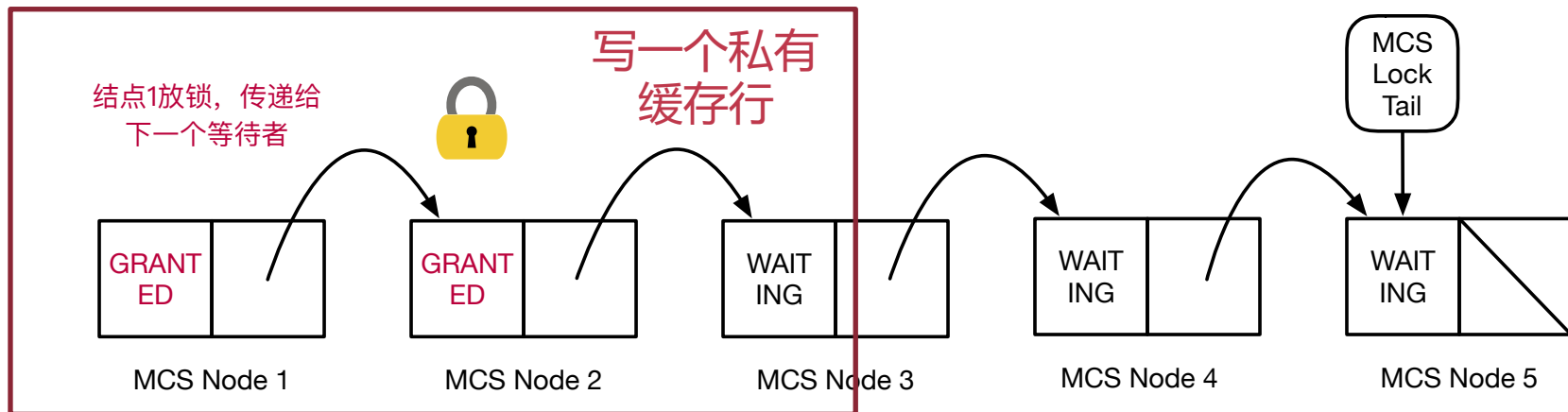
核心思路：在**关键路径**上避免对单一缓存行的高度竞争

以MCS锁为例

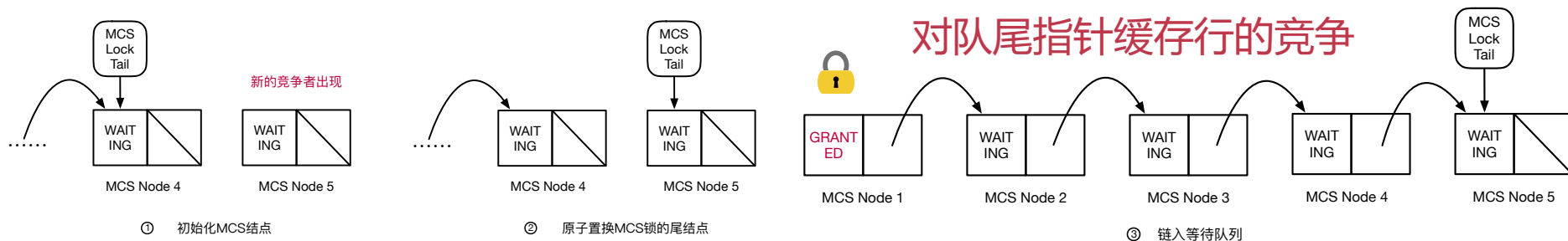


Review: MCS锁性能分析

不再会高频竞争全局缓存行

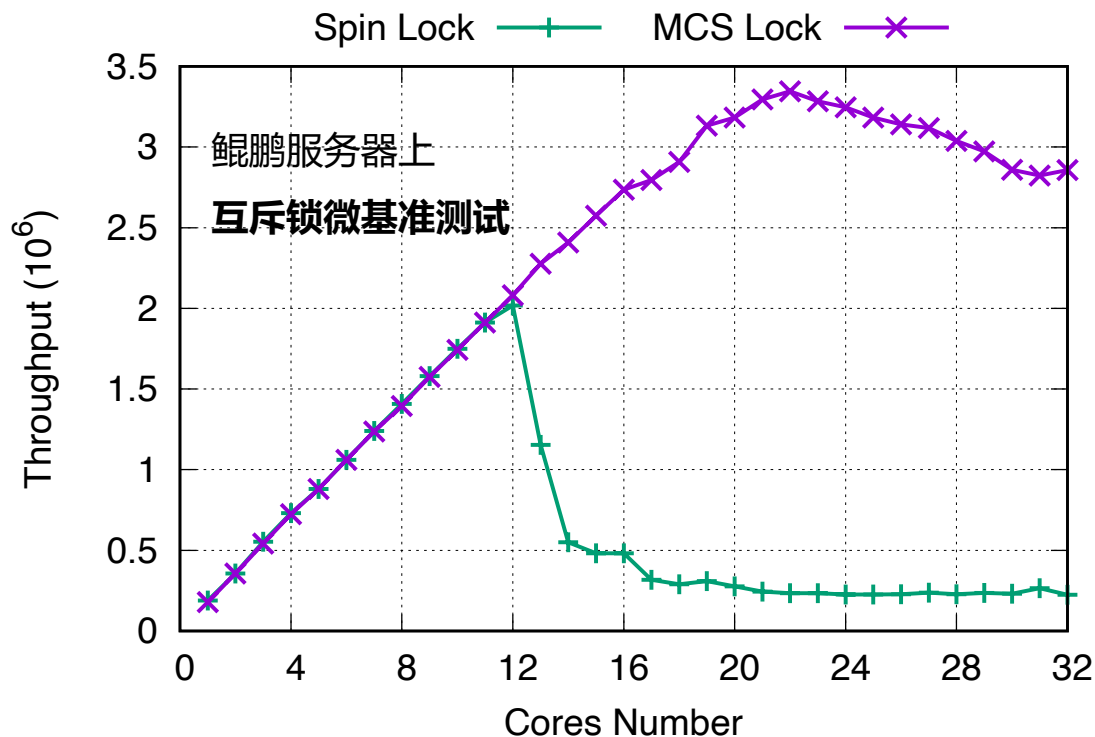


高竞争程度时的关键路径



高竞争程度时关键路径之外

Review: MCS锁性能评测



在**关键路径上**避免
对单一缓存行的高度竞争

Non-scalable locks are
dangerous, use scalable
locks instead!

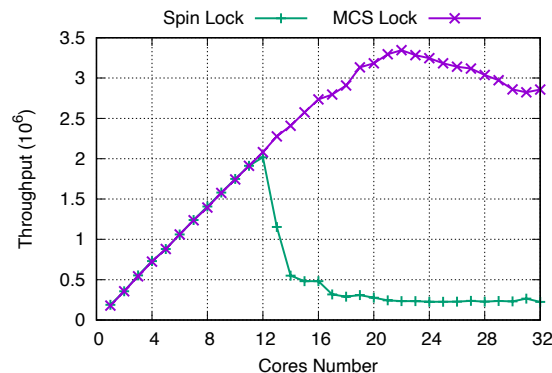
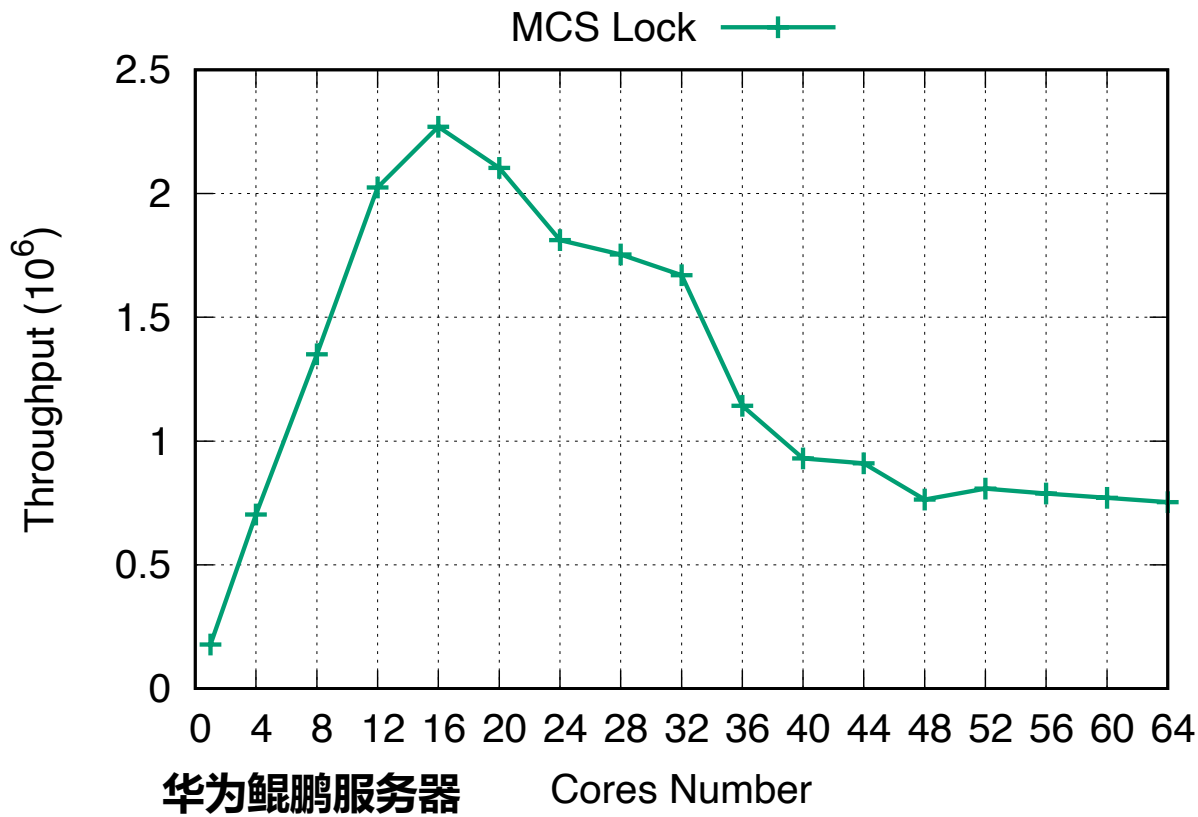
非一致内存访问 (NUMA)

互斥锁微基准测试

```
struct lock *glock;
unsigned long gcnt = 0;
char shared_data[CACHE_LINE_SIZE * 10];
void *thread_routine(void *arg) {
    while(1) {
        lock(glock);
        /* Critical Section */
        gcnt = gcnt + 1;
        /* Read Modify Write 10 * shared cacheline */
        visit_shared_data(shared_data, 10);
        unlock(glock);
        interval();
    }
}
```

在临界区访问更多缓存行，是否会影响 mcs 锁的可扩展性？

MCS锁在NUMA上的表现

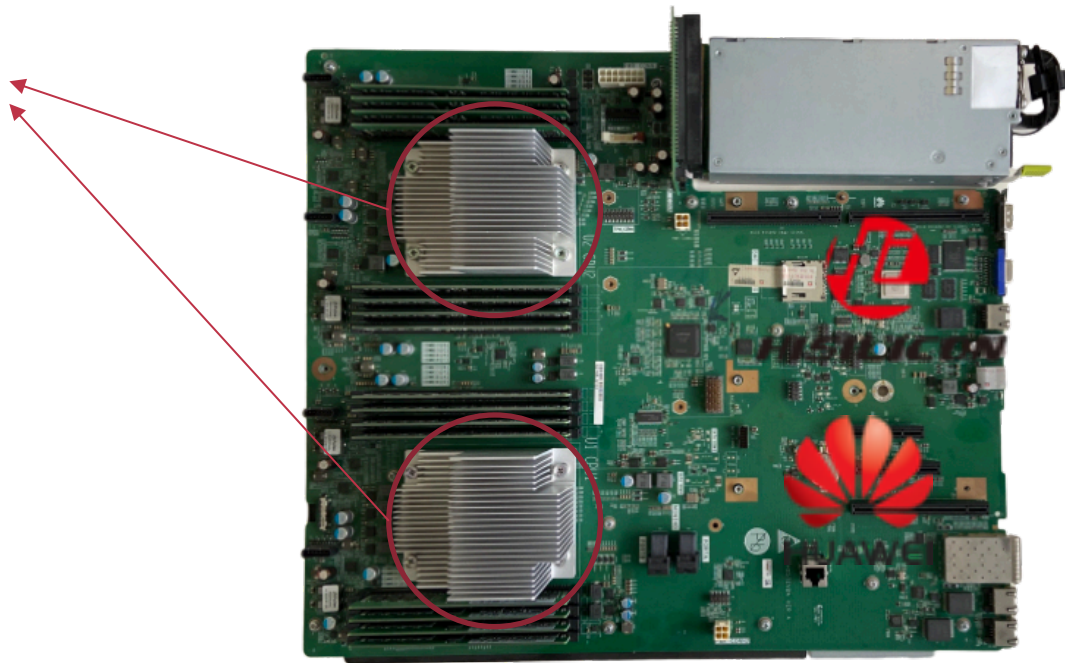


与之前测试有两点不同：

1. 临界区访问共享缓存行数量 1 => 10
2. 测试核心数扩大到 64个核心

鲲鹏服务器

2*处理器



华为鲲鹏服务器（ARM架构）

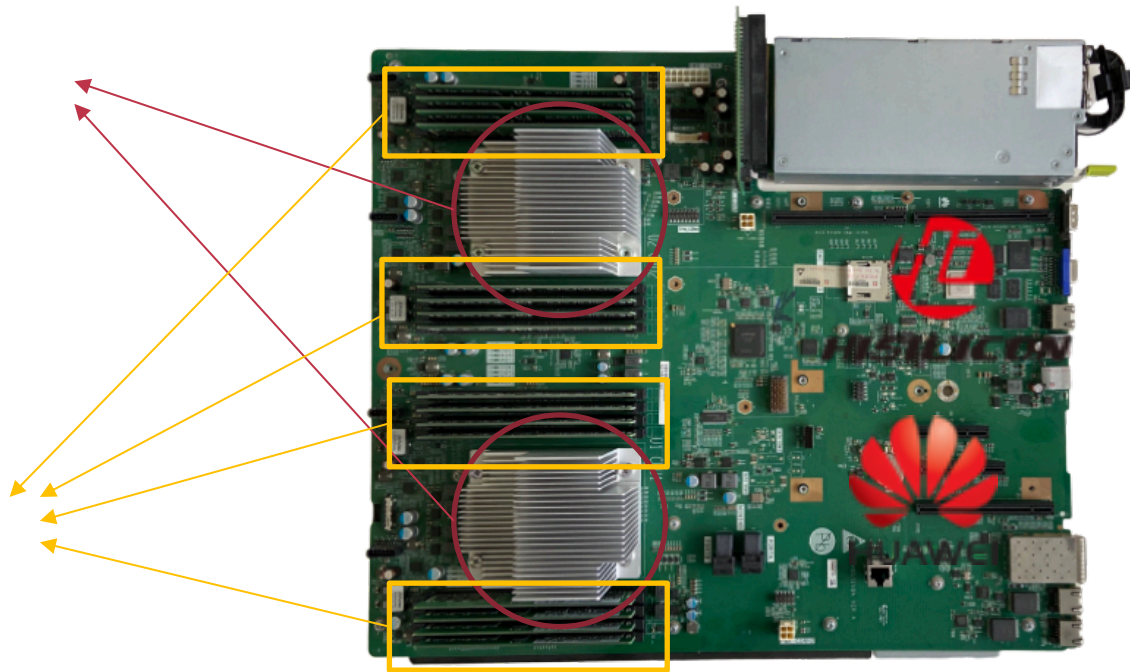
常见于多处理器（多插槽）机器

上海交通大学并行与分布式系统研究所（IPADS@SJTU）

鲲鹏服务器

2*处理器

内存条

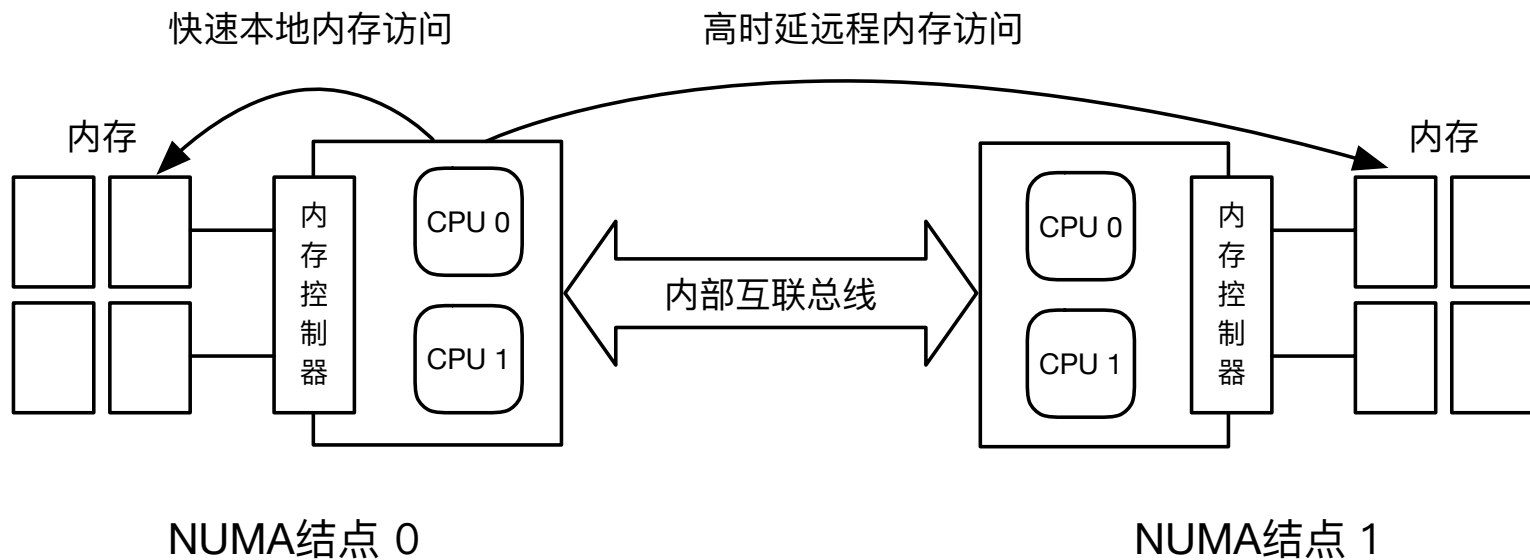


华为鲲鹏服务器（ARM架构）

常见于多处理器（多插槽）机器

上海交通大学并行与分布式系统研究所（IPADS@SJTU）

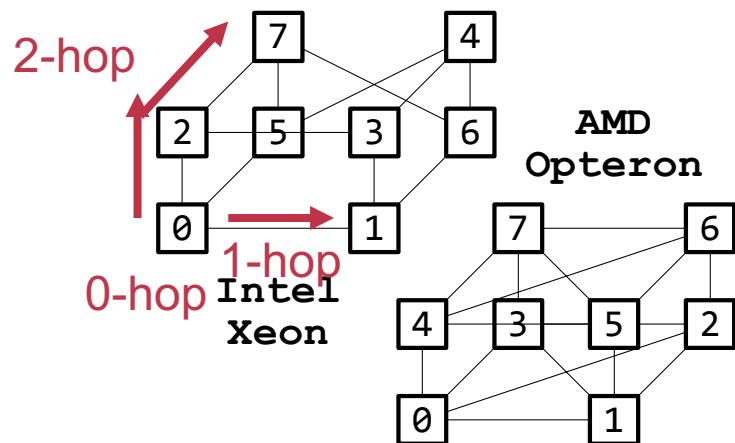
非一致内存访问



避免单内存控制器成为瓶颈，减少内存访问距离

常见于多处理器（多插槽）机器 单处理器众核系统也有可能使用，如Intel Xeon Phi

Intel与AMD的NUMA系统架构与特性



Intel与AMD多插槽NUMA架构
结构复杂

| Inst. | 0-hop | 1-hop | 2-hop |
|-----------------------------|-------|-------|-------|
| 80-core Intel Xeon machine | | | |
| Load | 117 | 271 | 372 |
| Store | 108 | 304 | 409 |
| 64-core AMD Opteron machine | | | |
| Load | 228 | 419 | 498 |
| Store | 256 | 463 | 544 |

Intel与AMD NUMA访存时延特性
跳数(hop)越多, 延迟越高

Intel与AMD的NUMA系统架构与特性

| Access | 0-hop | 1-hop | 2-hop | Interleaved |
|-----------------------------|-------|-----------|-------|-------------|
| 80-core Intel Xeon machine | | | | |
| Sequential | 3207 | 2455 | 2101 | 2333 |
| Random | 720 | 348 | 307 | 344 |
| 64-core AMD Opteron machine | | | | |
| Sequential | 3241 | 2806/2406 | 1997 | 2509 |
| Random | 533 | 509/487 | 415 | 466 |

Intel与AMD NUMA访存带宽特性 (MB/s)

跳数越多，带宽受限

NUMA环境中新的挑战

```
while(TRUE) {
```

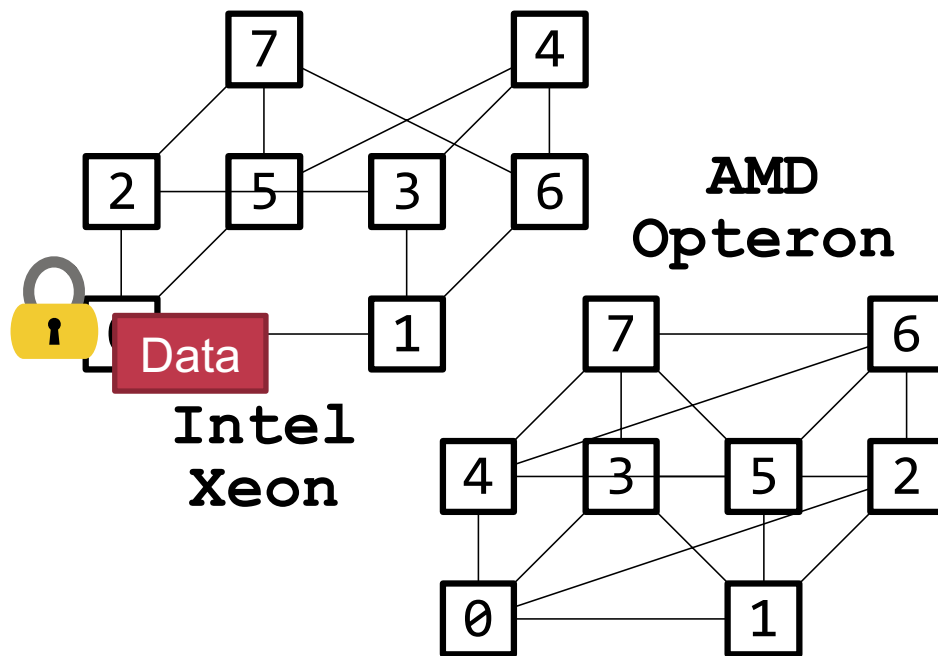
申请进入临界区

临界区部分

通知退出临界区

其他代码

```
}
```



NUMA环境中新的挑战

除了锁的元数据，
主要是临界区中
访问的共享数据

```
while(TRUE) {
```

申请进入临界区

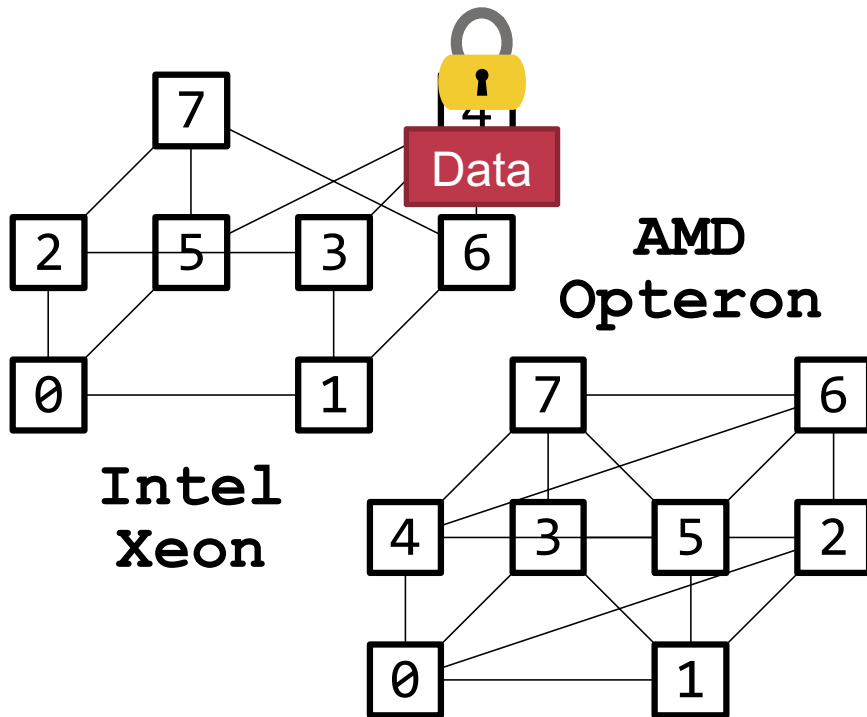
临界区部分

通知退出临界区

其他代码

```
}
```

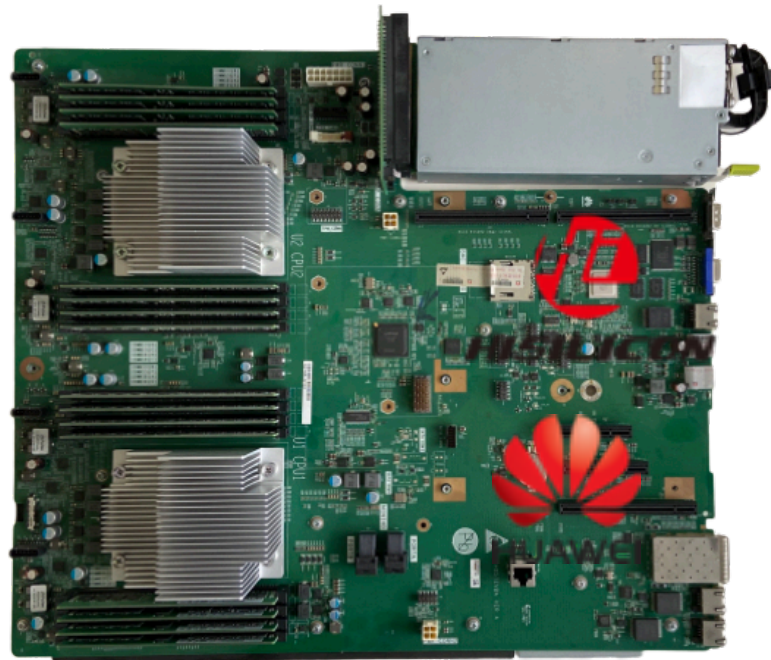
Challenge: 锁**不知道**临界区
中需要访问的内容！



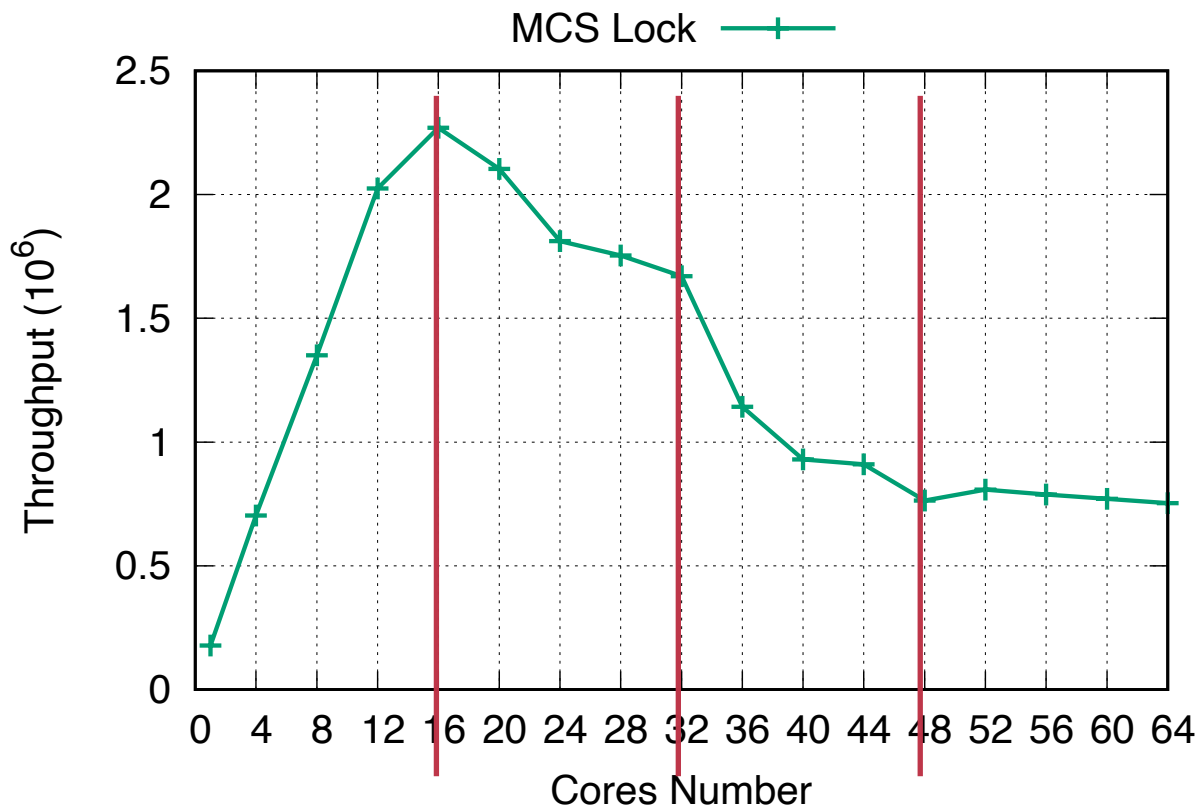
即使在cc-NUMA中没有出现缓存失效 **跨结点的缓存一致性协议开销巨大**

鲲鹏服务器的NUMA结构

```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15
node 1 cpus: 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31
node 2 cpus: 32 33 34 35 36 37 38
39 40 41 42 43 44 45 46 47
node 3 cpus: 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63
node distances:
node    0    1    2    3
  0:   10   15   20   20
  1:   15   10   20   20
  2:   20   20   10   15
  3:   20   20   15   10
```



MCS锁可扩展性



鲲鹏服务器

```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
6 7 8 9 10 11 12 13 14
15
node 1 cpus: 16 17 18 19
20 21 22 23 24 25 26 27
28 29 30 31
node 2 cpus: 32 33 34 35
36 37 38 39 40 41 42 43
44 45 46 47
node 3 cpus: 48 49 50 51
52 53 54 55 56 57 58 59
60 61 62 63
node distances:
node   0    1    2    3
0:   10   15   20   20
1:   15   10   20   20
2:   20   20   10   15
3:   20   20   15   10
```


NUMA-aware设计：以cohort锁*为例

核心思路：在一段时间内将访存限制在本地

先获取**每结点本地锁**

再获取全局锁

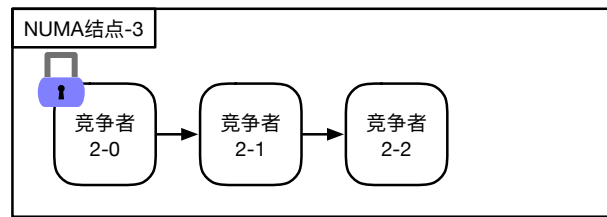
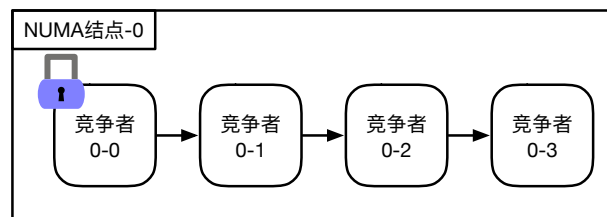
成功获取全局锁

释放时将其传递给

本地等待队列的下一位

全局锁在一段时间内

只在一个结点内部传递



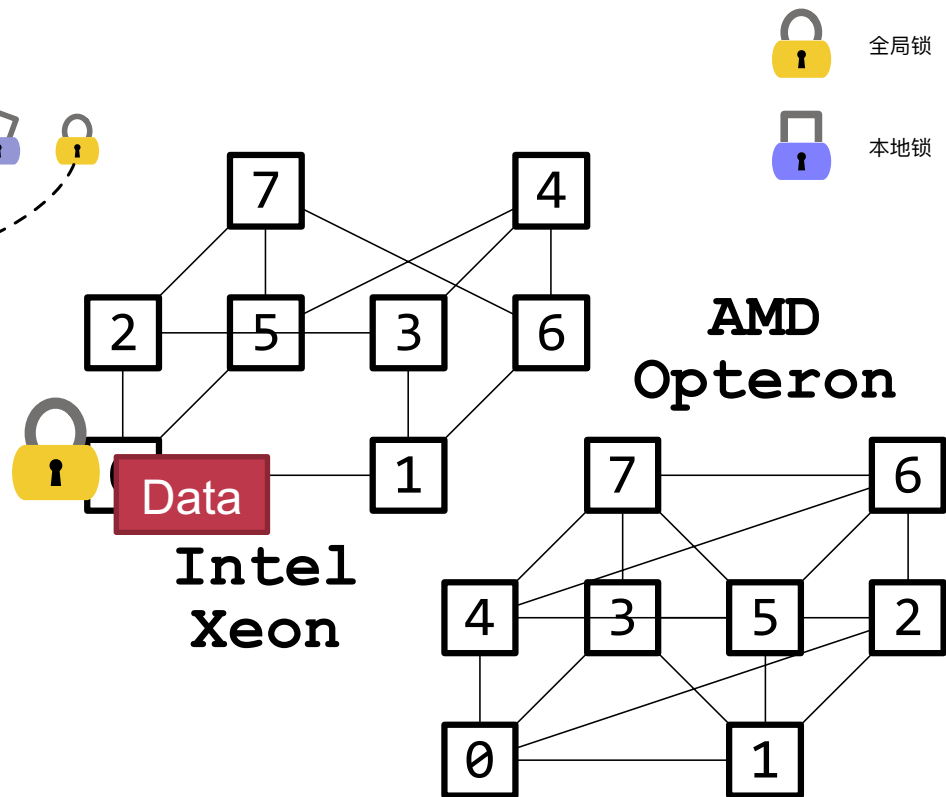
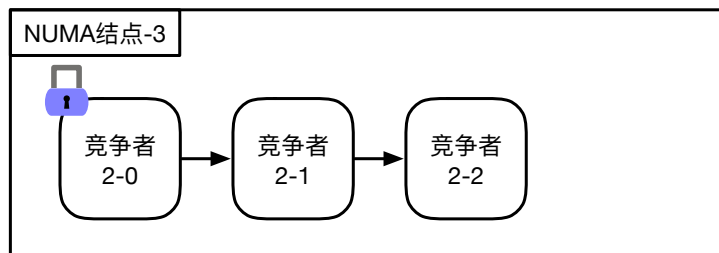
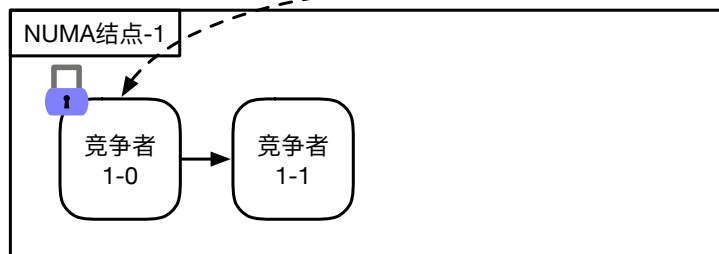
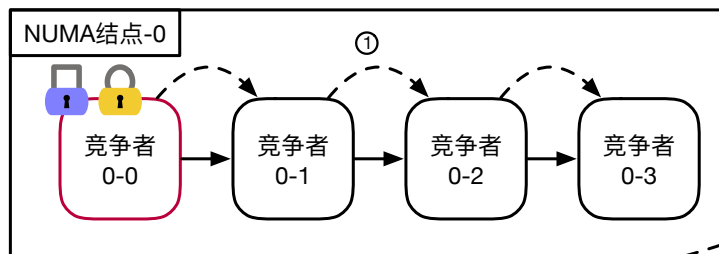
全局锁



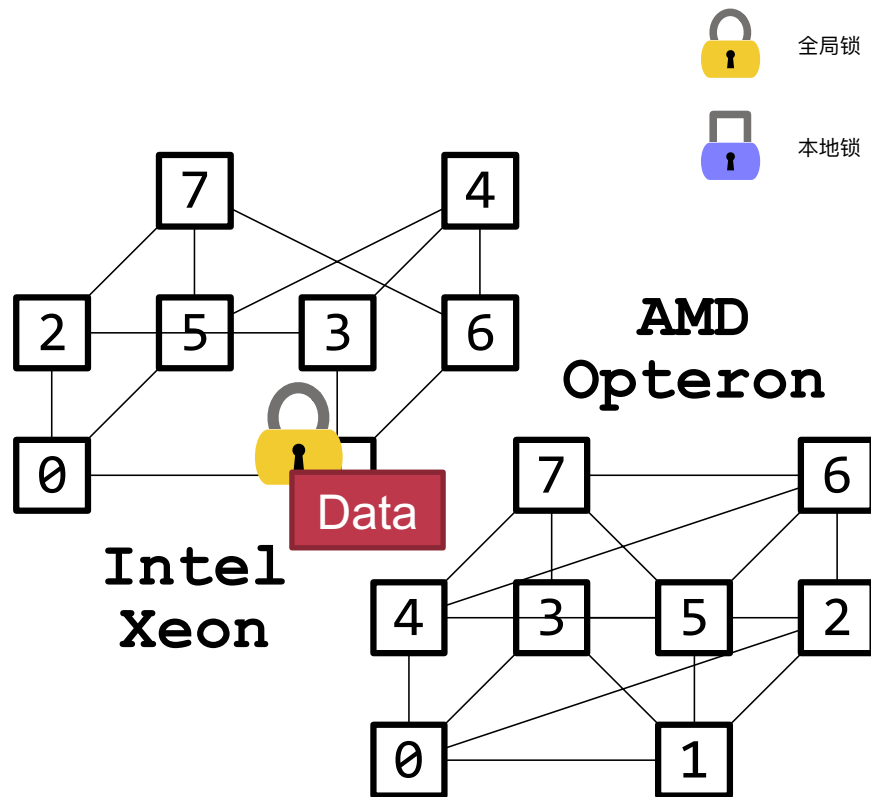
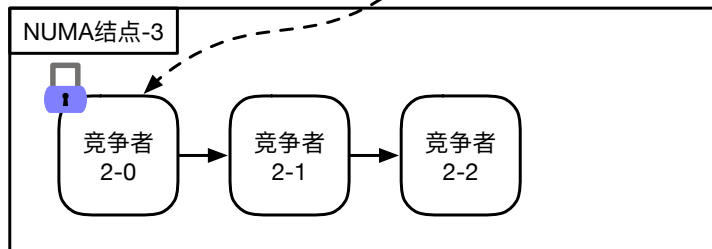
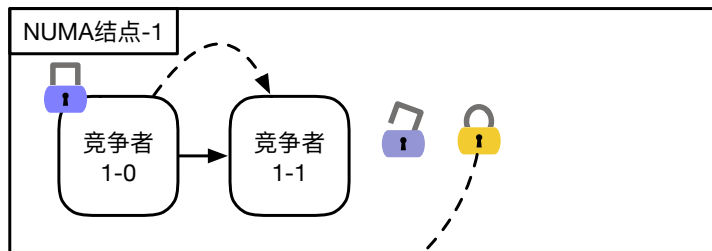
本地锁

**Dice, David, Virendra J. Marathe, and Nir Shavit. "Lock cohorting: a general technique for designing NUMA locks."*

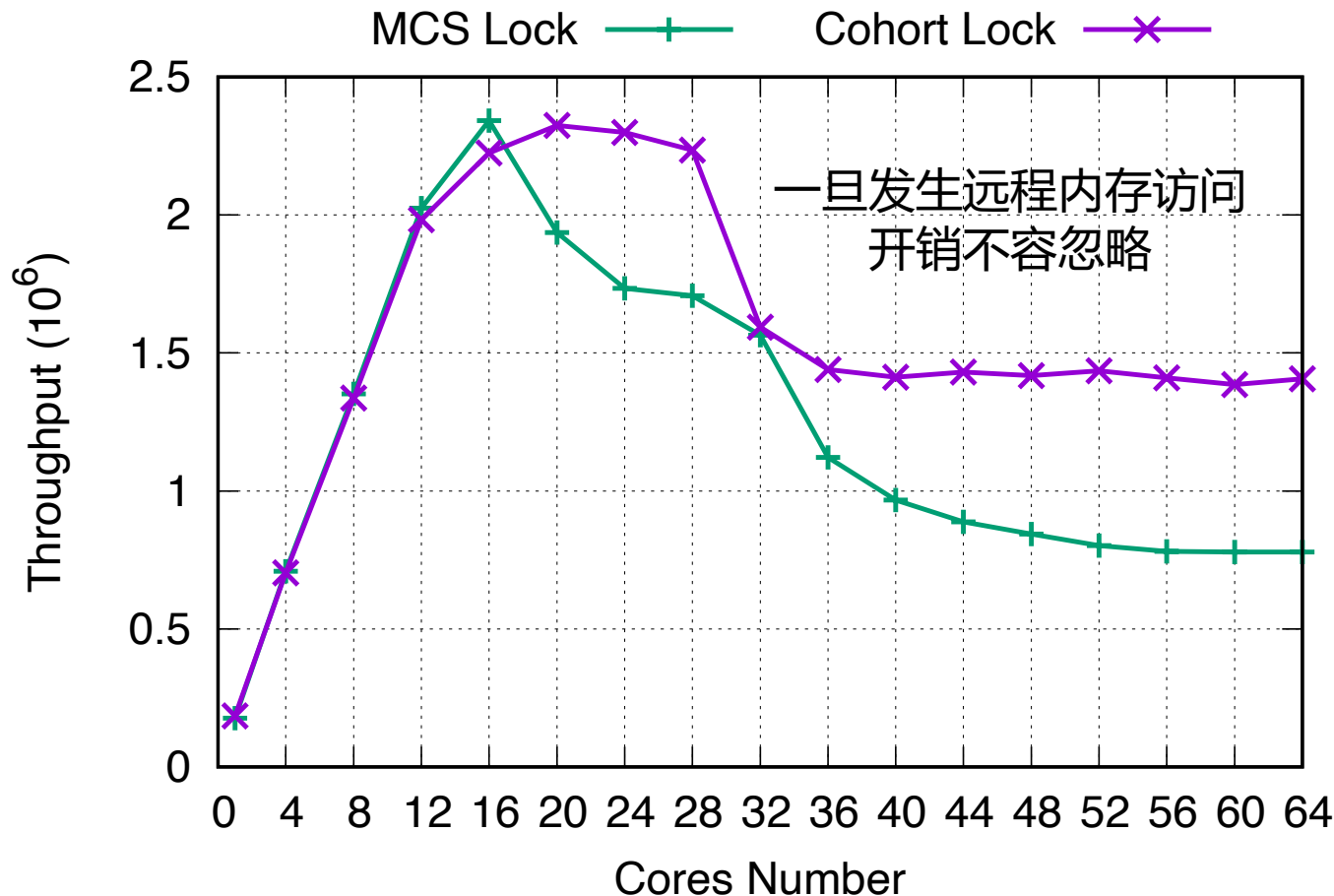
NUMA-aware设计：以cohort锁*为例



NUMA-aware设计：以cohort锁*为例



NUMA-aware设计：以cohort锁*为例



系统软件开发视角下的NUMA架构

- NUMA会暴露给操作系统，操作系统可以选择暴露给软件
- 软件可以用接口来分配本地的内存，也可不用直接分配，如（libnuma）
- 访问**远程内存**会带来严重**时延/带宽**问题造成性能瓶颈
- 对于所有进程：调度时避免跨NUMA结点迁移
- 对于没有NUMA-aware的应用：尽可能保证其分配的内存的本地性



内存模型

LockOne : 皮特森算法的前身

线程 - 0

```
1. while(TRUE) {  
2.     flag[0] = true;  
4.     while (flag[1] == true);
```

临界区部分

```
6.     flag[0] = false;
```

其他代码

```
7. }
```

线程 - 1

```
1. while(TRUE) {  
2.     flag[1] = true;  
4.     while (flag[0] == true);
```

临界区部分

```
6.     flag[1] = false;
```

其他代码

```
7. }
```

多核环境下能够互斥访问吗？

LockOne在现实硬件中能够保证互斥访问吗？

缓存一致性耗时：**阻塞**处理器**流水线**，造成巨大性能开销

处理器允许部分访存操作**乱序执行**，从而提供更好的并行性

```
void proc_A(void) {  
    flag[0] = 1;  
    while(flag[1]);  
}  
  
void proc_B(void) {  
    flag[1] = 1;  
    while(flag[0]);  
}
```



Read From , 必须读到对方设置的最新值

LockOne在现实硬件中能够保证互斥访问吗？

缓存一致性耗时：**阻塞**处理器**流水线**，造成巨大性能开销

处理器允许部分访存操作**乱序执行**，从而提供更好的并行性

```
void proc_A(void) {  
    flag[0] = 1;  
    while(flag[1]);  
}
```

乱序

```
void proc_B(void) {  
    flag[1] = 1;  
    while(flag[0]);  
}
```

乱序

LockOne在现实硬件中能够保证互斥访问吗？

缓存一致性耗时：**阻塞**处理器**流水线**，造成巨大性能开销

处理器允许部分访存操作**乱序执行**，从而提供更好的并行性

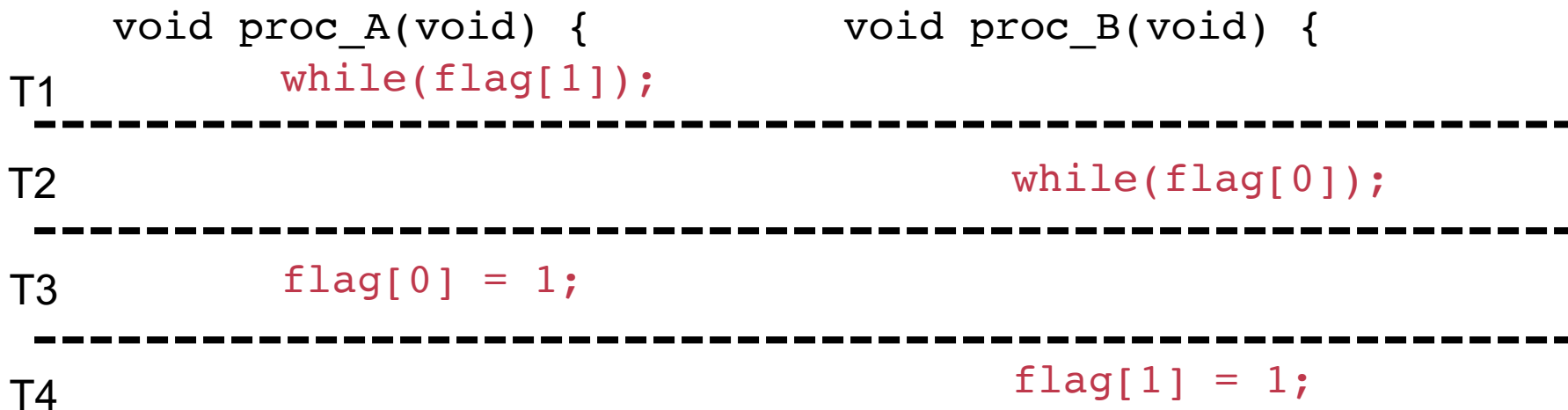
```
void proc_A(void) {  
    while(flag[1]);  
    flag[0] = 1;  
}
```

```
void proc_B(void) {  
    while(flag[0]);  
    flag[1] = 1;  
}
```

LockOne在现实硬件中能够保证互斥访问吗？

缓存一致性耗时：**阻塞**处理器**流水线**，造成巨大性能开销

处理器允许部分访存操作**乱序执行**，从而提供更好的并行性

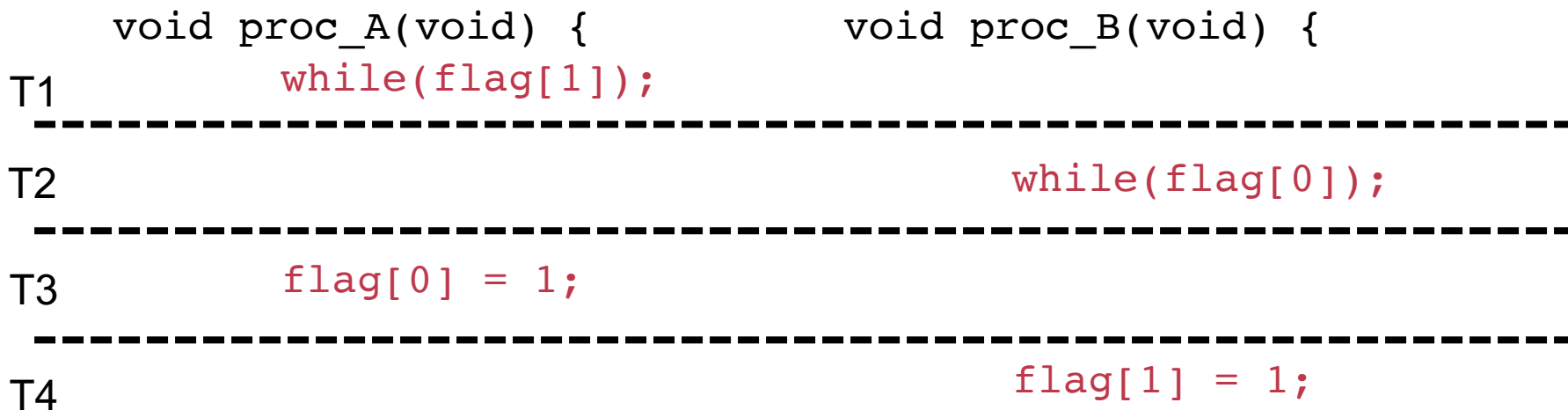


LockOne在现实硬件中能够保证互斥访问吗？

缓存一致性耗时：**阻塞**处理器**流水线**，造成巨大性能开销

处理器允许部分访存操作**乱序执行**，从而提供更好的并行性

proc_A 与 proc_B 都读到对方flag为0，同时进入临界区



LockOne在现实硬件中能够保证互斥访问吗？

会发生这个现象



不会发生这个现象

dual 386 (circa 1989)



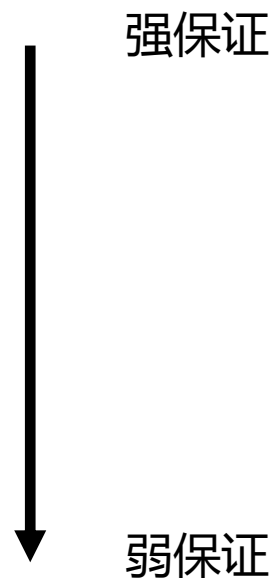
几种内存模型

严格一致性模型

顺序一致性模型

TSO一致性模型

弱序一致性模型



内存模型：严格一致性模型

- 严格一致性模型 (Strict Consistency)

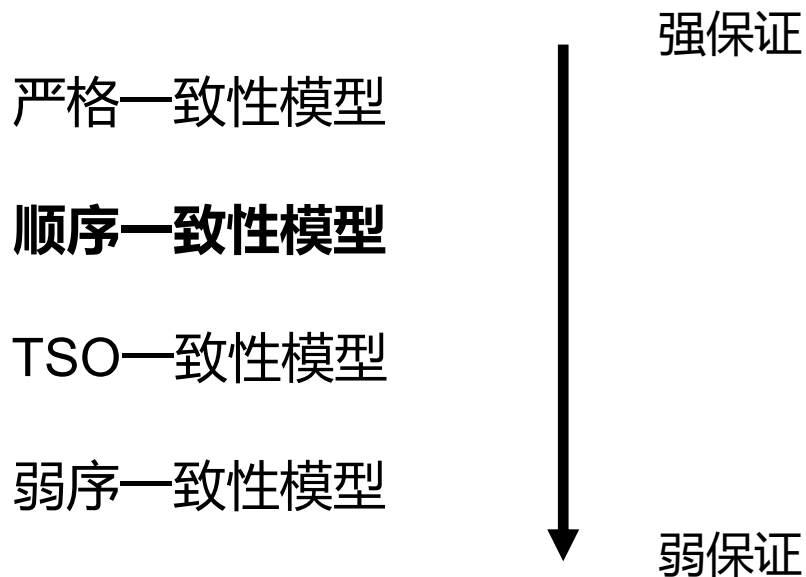
对一个地址的任意的**读操作**都能读到这个地址**最近一次写**的数据

访存操作顺序与**全局时钟**的顺序完全一致

| | <code>void proc_A(void) {</code> | <code>void proc_B(void) {</code> |
|----|----------------------------------|----------------------------------|
| T1 | <code>flag[0] = 1;</code> | |
| | | |
| T2 | | <code>flag[1] = 1;</code> |
| | | |
| T3 | <code>A = flag[1];</code> | |
| | | |
| T4 | <code>}</code> | <code>B = flag[0];</code> |

唯一可能：(A,B) = (1,1)

几种内存模型



内存模型：顺序一致性模型

- 顺序一致性模型 (Sequential Consistency)

不要求操作按照真实发生的时间顺序 (全局时钟) 全局可见

执行结果必须与**一个全局的顺序**执行一致

且这个全局顺序中一个核心的读写操作与其**程序顺序**保持一致。

```
void proc_A(void) {                                void proc_B(void) {
```

| | | |
|----|--------------|--------------|
| T1 | flag[0] = 1; | |
| | | |
| T2 | | flag[1] = 1; |
| | | |
| T3 | A = flag[1]; | |
| | | |
| T4 | } | B = flag[0]; |

多种可能结果

}

内存模型：顺序一致性模型

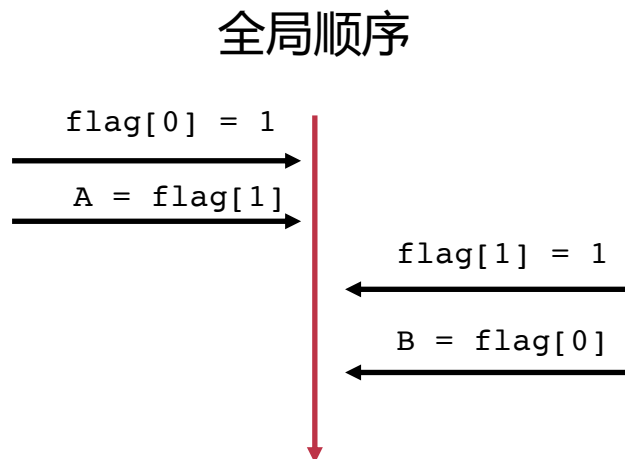
- 顺序一致性模型 (Sequential Consistency)

不要求操作按照真实发生的时间顺序 (全局时钟) 全局可见

执行结果必须与一个**全局的顺序**执行一致

且这个全局顺序中一个核心的读写操作与其**程序顺序**保持一致。

第一种可能：



实际发生顺序

T1 CPU0 flag[0] = 1
T2 CPU1 flag[1] = 1
T3 CPU0 A = flag[1]
T4 CPU1 B = flag[0]

(A , B) = (0 , 1)

内存模型：顺序一致性模型

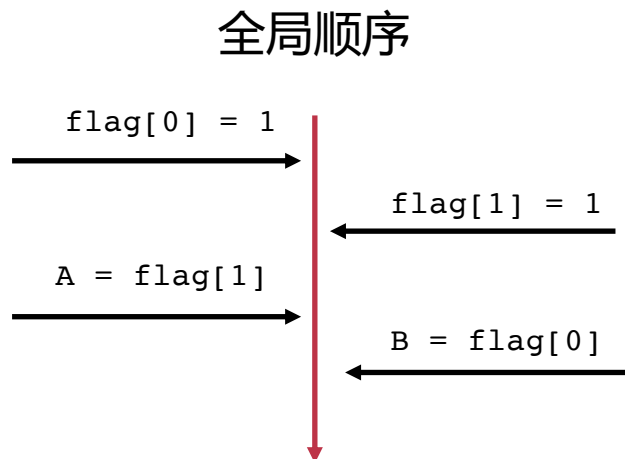
- 顺序一致性模型 (Sequential Consistency)

不要求操作按照真实发生的时间顺序 (全局时钟) 全局可见

执行结果必须与一个**全局的顺序**执行一致

且这个全局顺序中一个核心的读写操作与其**程序顺序**保持一致。

第二种可能：



实际发生顺序

T1 CPU0 flag[0] = 1
T2 CPU1 flag[1] = 1
T3 CPU0 A = flag[1]
T4 CPU1 B = flag[0]

$(A, B) = (1, 1)$

内存模型：顺序一致性模型

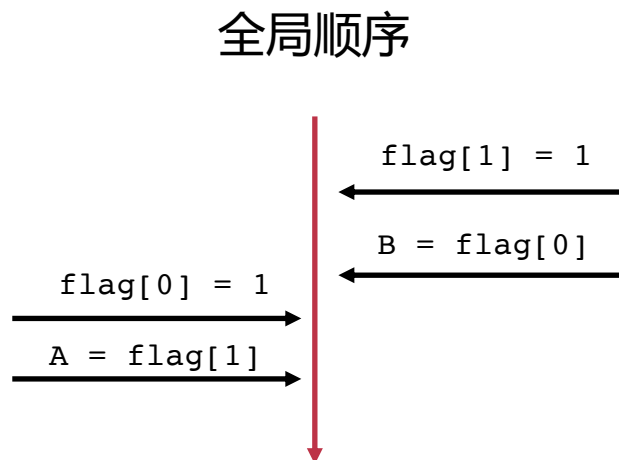
- 顺序一致性模型 (Sequential Consistency)

不要求操作按照真实发生的时间顺序 (全局时钟) 全局可见

执行结果必须与一个**全局的顺序**执行一致

且这个全局顺序中一个核心的读写操作与其**程序顺序**保持一致。

第三种可能：



实际发生顺序

T1 CPU0 flag[0] = 1
T2 CPU1 flag[1] = 1
T3 CPU0 A = flag[1]
T4 CPU1 B = flag[0]

$(A, B) = (1, 0)$

内存模型：顺序一致性模型

- 顺序一致性模型 (Sequential Consistency)

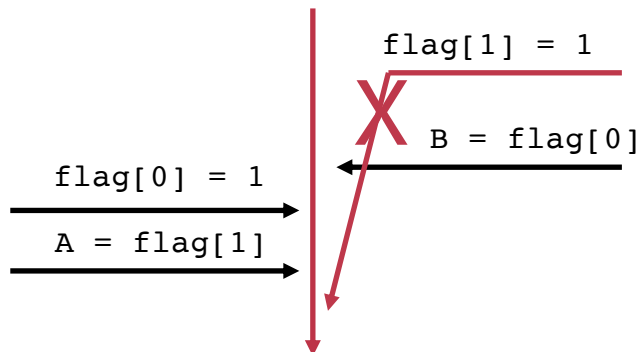
不要求操作按照真实发生的时间顺序 (全局时钟) 全局可见

执行结果必须与一个**全局的顺序**执行一致

且这个全局顺序中一个核心的读写操作与其**程序顺序**保持一致。

禁止：

全局序列中，proc_B的程序顺序被打破了

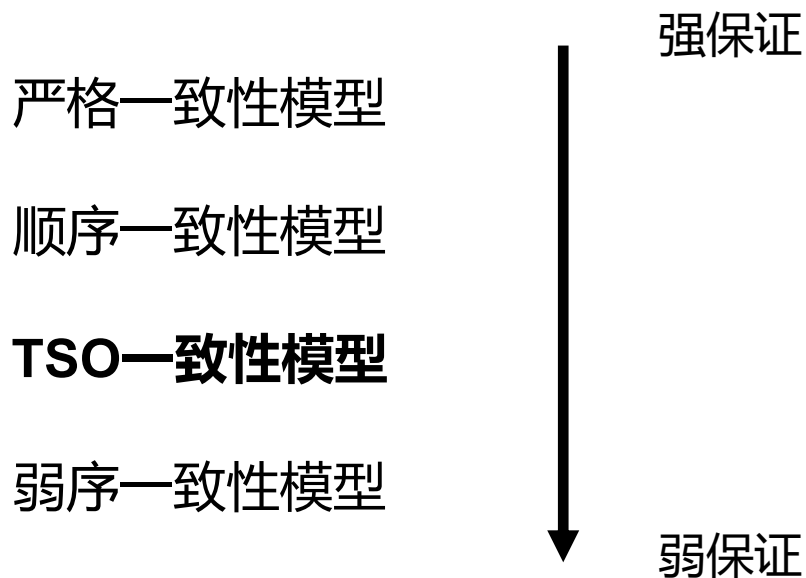


```
T1 CPU0 flag[0] = 1
T2 CPU1 flag[1] = 1
T3 CPU0 A = flag[1]
T4 CPU1 B = flag[0]
```

(A, B) **不可能为** (0, 0)

不会同时进入临界区

几种内存模型



内存模型：TSO一致性模型

- TSO 一致性模型 (Total Store Ordering)

针对**不同地址**的**读-读**、**读-写**、**写-写**顺序都能得到保证

只有**写-读**的顺序**不能**够得到保证

初始值：flag[0] = 0 flag[1] = 0

| | <code>void proc_A(void) {</code> | <code>void proc_B(void) {</code> |
|----|----------------------------------|----------------------------------|
| T1 | <code>flag[0] = 1; 写</code> | |
| T2 | | <code>flag[1] = 1; 写</code> |
| T3 | <code>A = flag[1]; 读</code> | |
| T4 | <code>}</code> | <code>B = flag[0]; 读</code> |

内存模型：TSO一致性模型

- TSO 一致性模型 (Total Store Ordering)

针对**不同地址**的**读-读**、**读-写**、**写-写**顺序都能得到保证

只有**写-读**的顺序**不能**够得到保证

```
void proc_A(void) {  
    flag[0] = 1;  
    A = flag[1];  
}
```



```
void proc_B(void) {  
    flag[1] = 1;  
    B = flag[0];  
}
```



内存模型：TSO一致性模型

- TSO 一致性模型 (Total Store Ordering)

针对**不同地址**的**读-读**、**读-写**、**写-写**顺序都能得到保证

只有**写-读**的顺序**不能**够得到保证

初始值：flag[0] = 0 flag[1] = 0

```
void proc_A(void) {
```

```
    A = flag[1];
```

```
void proc_A(void) {
```

```
    B = flag[0];
```

```
    flag[0] = 1;
```

```
}
```

最终允许结果：(A,B) = (0,0)

```
    flag[1] = 1;
```

```
}
```

内存模型：TSO一致性模型

- TSO 一致性模型 (Total Store Ordering)

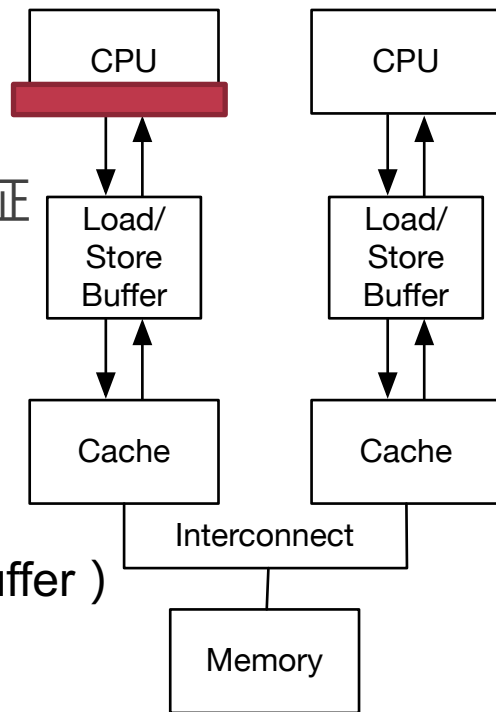
针对**不同地址**的**读-读**、**读-写**、**写-写**顺序都能得到保证

只有**写-读**的顺序不能够得到保证

为什么看起来很奇怪？规定这么细致？

- 需要允许**乱序执行**来提升性能！
- 要保证顺序：使用了**内存保序缓存** (Memory Ordering Buffer)
 - 包括**写缓存**以及**读缓存** (Store/Load Buffer)
 - 依序进出，保证顺序
- TSO：Intel (AMD) 硬件**复杂度、性能、软件使用场景**权衡的结果

写操作**需要等待**
缓存一致性



内存模型：TSO一致性模型

- TSO 一致性模型 (Total Store Ordering)

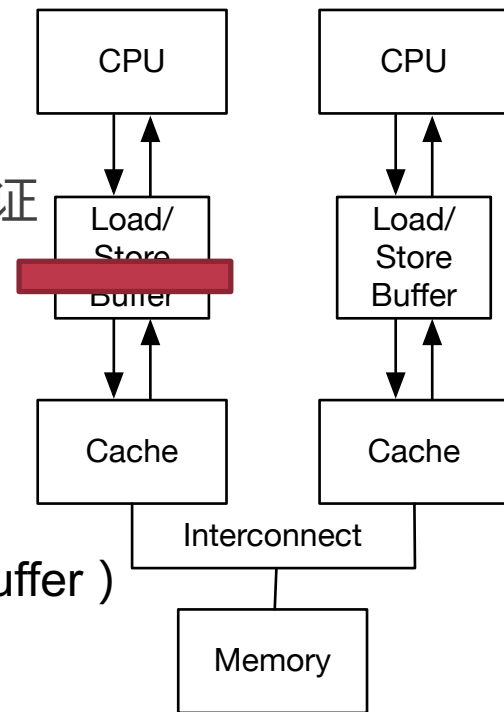
针对**不同地址**的**读-读**、**读-写**、**写-写**顺序都能得到保证

只有**写-读**的顺序不能够得到保证

为什么看起来很奇怪？规定这么细致？

- 需要允许**乱序执行**来提升性能！
- 要保证顺序：使用了**内存保序缓存** (Memory Ordering Buffer)
 - 包括**写缓存**以及**读缓存** (Store/Load Buffer)
 - 依序进出，保证顺序
- TSO：Intel (AMD) 硬件**复杂度、性能、软件使用场景**权衡的结果

放入store buffer，
CPU继续执行



内存模型：TSO一致性模型

- TSO 一致性模型 (Total Store Ordering)

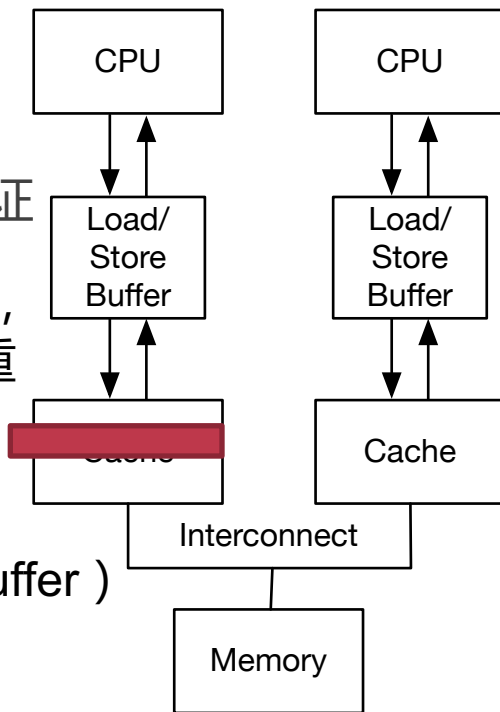
针对**不同地址**的**读-读**、**读-写**、**写-写**顺序都能得到保证

只有**写-读**的顺序不能够得到保证

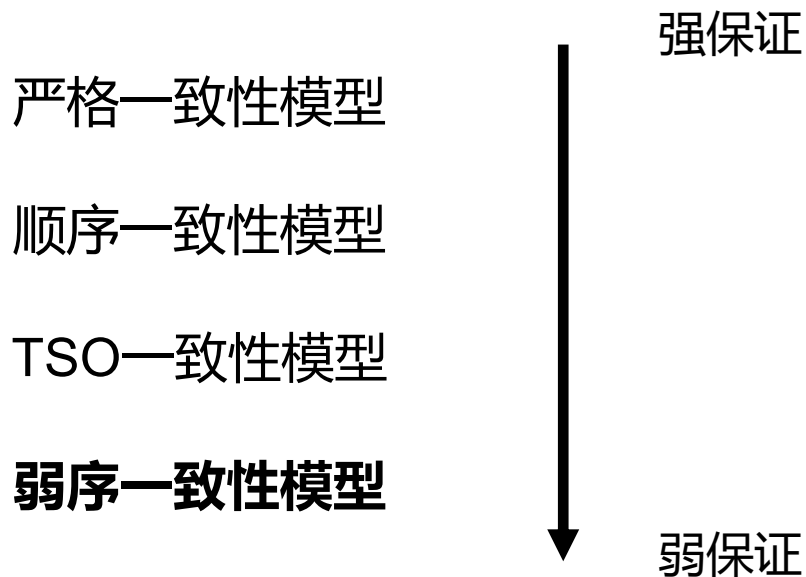
为什么看起来很奇怪？规定这么细致？

缓存一致性满足，
真实写到cache重

- 需要允许**乱序执行**来提升性能！
- 要保证顺序：使用了**内存保序缓存** (Memory Ordering Buffer)
 - 包括**写缓存**以及**读缓存** (Store/Load Buffer)
 - 依序进出，保证顺序
- TSO：Intel (AMD) 硬件**复杂度、性能、软件使用场景**权衡的结果



几种内存模型



内存模型：弱序一致性模型

- 弱序一致性模型 (Weak-ordering Consistency)

不保证任何对**不同的地址**的**读写**操作顺序

```
int data = 0;  
int flag = NOT_READY;
```

```
void proc_A(void) {  
    data = 666;  
    flag = READY;  
}
```



```
void proc_B(void) {  
    while (flag != READY) ;  
    handle(data);  
}
```

内存模型：弱序一致性模型

- 弱序一致性模型 (Weak-ordering Consistency)

不保证任何对**不同的地址**的**读写**操作顺序

```
int data = 0;
int flag = NOT_READY;
```

```
void proc_A(void) {
    flag = READY;
```

```
void proc_B(void) {
    while (flag != READY) ;
    handle(data);
```

```
} 时刻T1, proc_B读到错误的值
```

```
-----
    data = 666;
}
```

思考：TSO中有这个问题吗？为什么？

内存模型：弱序一致性模型

- 弱序一致性模型（Weak-ordering Consistency）

不保证任何对**不同的地址**的**读写**操作顺序

与TSO相比：

- 硬件逻辑更加简单
- 处理器复杂度下降
- 工艺/成本/功耗下降
- 并行程序性能受到影响（需要手动保证顺序）
- ARM硬件**复杂度、性能、成本、功耗、软件使用场景**权衡的结果

不同架构使用不同的内存模型



*2012年之前

不同架构使用不同的内存模型




如何在弱的内存模型中保证顺序

LockOne算法

通常的做法：添加**硬件内存屏障**（barrier/fence）


```
void proc_A(void) {  
    flag[0] = 1;  
    while(flag[1]);  
}
```

保证写-读顺序



```
void proc_B(void) {  
    flag[1] = 1;  
    while(flag[0]);  
}
```

保证写-读顺序




传输数据例子

任何访存操作**不会逾越**内存屏障


```
void proc_A(void) {  
    data = 666;  
    flag = READY;  
}
```

保证写-写顺序



```
void proc_B(void) {  
    while (flag != READY) ;  
    handle(data);  
}
```

保证读-读顺序



如何在弱的内存模型中保证顺序

LockOne算法

通常的做法：添加**硬件内存屏障**（ barrier/fence ）

```
void proc_A(void) {  
    flag[0] = 1;  
    barrier();  
    while(flag[1]);  
}
```

```
void proc_B(void) {  
    flag[1] = 1;  
    barrier();  
    while(flag[0]);  
}
```

传输数据例子

任何访存操作**不会逾越**内存屏障

```
void proc_A(void) {  
    data = 666;  
    barrier();  
    flag = READY;  
}
```

```
void proc_B(void) {  
    while (flag != READY) ;  
    barrier();  
    handle(data);  
}
```

系统软件开发视角下的内存模型

- 内存模型**不是透明的**
- 软件需要**手动**根据运行架构**保证访存操作顺序**
- **同步原语**（互斥锁、信号量等）拥有**保证访存顺序**的语义

临界区内的访存操作不会在锁获取之前**可见**

Acquire_lock(xxx); —————界限

LD/ST

所有的访存操作都应该留在这个界限之内

Release_lock(xxx); —————界限

临界区内的访存操作不会在释放锁之后**可见**

系统软件开发视角下的内存模型

- 内存模型**不是透明的**
- 软件需要**手动**根据运行架构**保证访存操作顺序**
- **同步原语**（互斥锁、信号量等）拥有**保证访存顺序**的语义
- 系统需要使用**保序手段**(如barrier)提供**正确**的同步原语，保证软件正确性
- 硬件内存屏障(如barrier)**开销很大**，需要**合适的地方用合适的方法**保证顺序
- 正常情况下，软件需要使用**同步原语**来同步

Sidebar: ARM中的保序选择

- 较TSO模型更弱，更容易出现违反程序顺序的执行流
- 通常的做法：添加**硬件内存屏障**（ barrier/fence ）
- 除此之外：构造**依赖关系**（ 数据依赖/控制依赖 ）
- 不同的方案适用不同的场景
- 硬件内存屏障**开销很大**
- 需要**合适的地方用合适的方法**保证顺序

| | Load to Load | Load to Store | Store to Load | Store to Store | Any to Any |
|-------------|--------------|---------------|---------------|----------------|------------|
| DMB/DSB all | ✓ | ✓ | ✓ | ✓ | ✓ |
| DMB/DSB ld | ✓ | ✓ | ✗ | ✗ | ✗ |
| DMB/DSB st | ✗ | ✗ | ✗ | ✓ | ✗ |
| LDAR | ✓ | ✓ | ✗ | ✗ | ✗ |
| STLR | ✗ | ✓ | ✗ | ✓ | ✗ |
| Data Dep | ✗ | ✓ | ✗ | ✗ | ✗ |
| ADDR Dep | ✓ | ✓ | ✗ | ✗ | ✗ |
| CTRL Dep | ✗ | ✓ | ✗ | ✗ | ✗ |
| CTRL + ISB | ✓ | ✓ | ✗ | ✗ | ✗ |

Order-preserving Options

Apple M1同时支持TSO和Weak

One major concern with Rosetta is performance. With the transition from PPC to x86, one factor slowing down Rosetta was the different byte ordering used by the two platforms, with PowerPC being a big-endian architecture, and x86 little-endian. While byte ordering is not a problem for the transition from x86 to ARM, another issue related to memory, namely the memory consistency model [total store ordering](#) (TSO), could hamper performance in this case. To prevent this from happening, [Apple added support for x86 memory ordering to the M1 CPU](#), as Robert Graham noted on Twitter.

“

4/ So Apple simply cheated. They added Intel's memory-ordering to their CPU. When running translated x86 code, they switch the mode of the CPU to conform to Intel's memory ordering.

— Rob^{ert} Graham, provocateur (@ErrataRob) [November 25, 2020](#)

<https://www.infoq.com/news/2020/11/rosetta-2-translation/>