IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

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

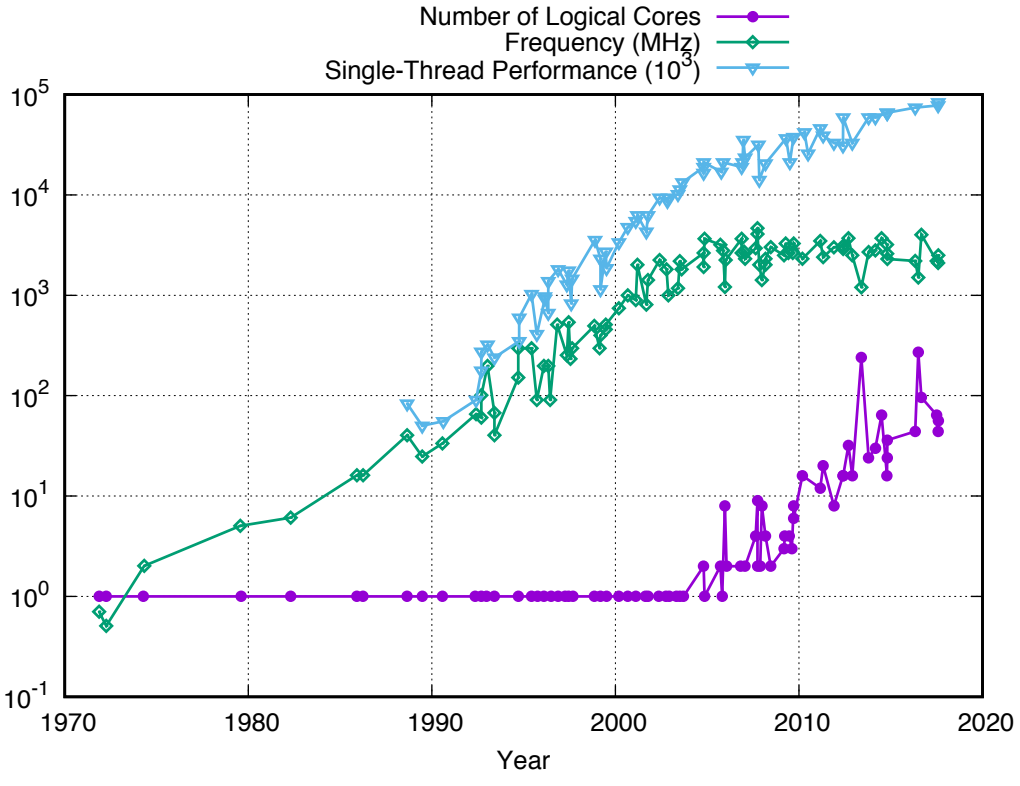
https://ipads.se.sjtu.edu.cn

版权声明

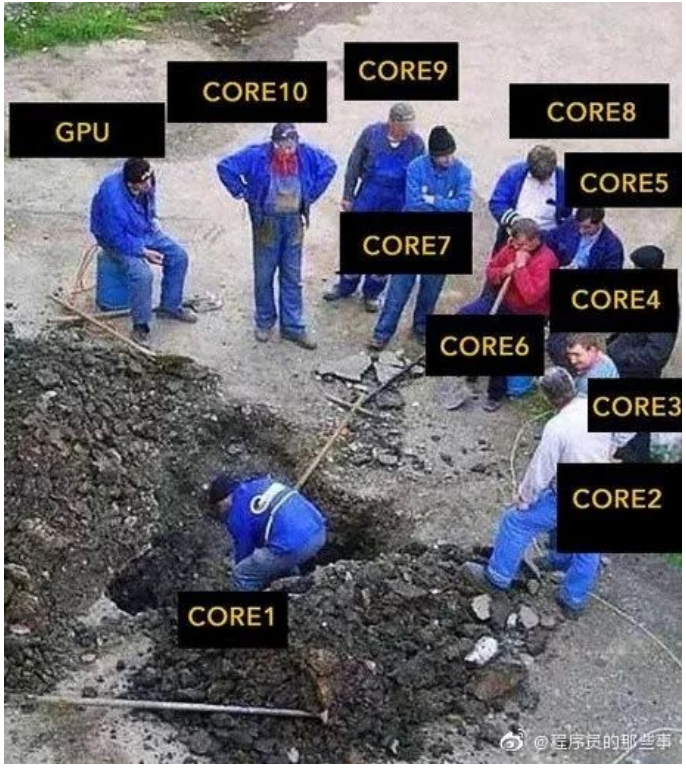
- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

回顾：多处理器与多核

- 单核性能提升遇到瓶颈
- 不能通过一味提升频率来获得更好的性能
- 通过增加核心数来提升软件的性能
- 桌面/移动平台均向多核迈进



回顾：多核不是免费的午餐



网图：多核的真相

假设现在需要建房子

工作量 = 1000人/年

工头找了10万人，需要多久？

面临的两个问题：

1. 工人人多手杂，不听指挥，导致施工事故。（**正确性**问题）
2. 工具有限，大部分工人无事可干。（**性能可扩展性**问题）

回顾：操作系统在多处理器多核环境下面临的问题

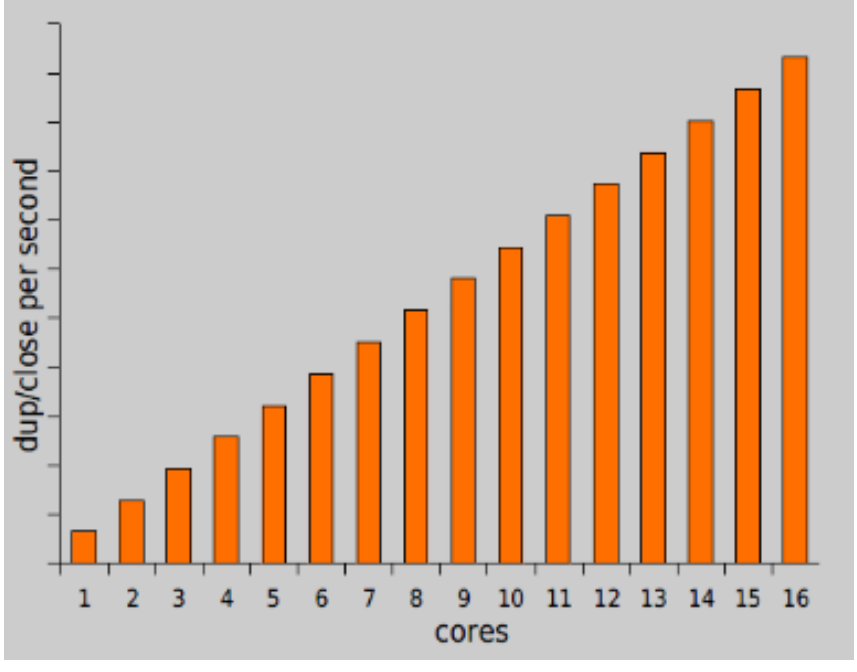
正确性保证

- 对共享资源的竞争导致错误
- 操作系统提供**同步原语**供开发者使用
- 使用同步原语带来的问题

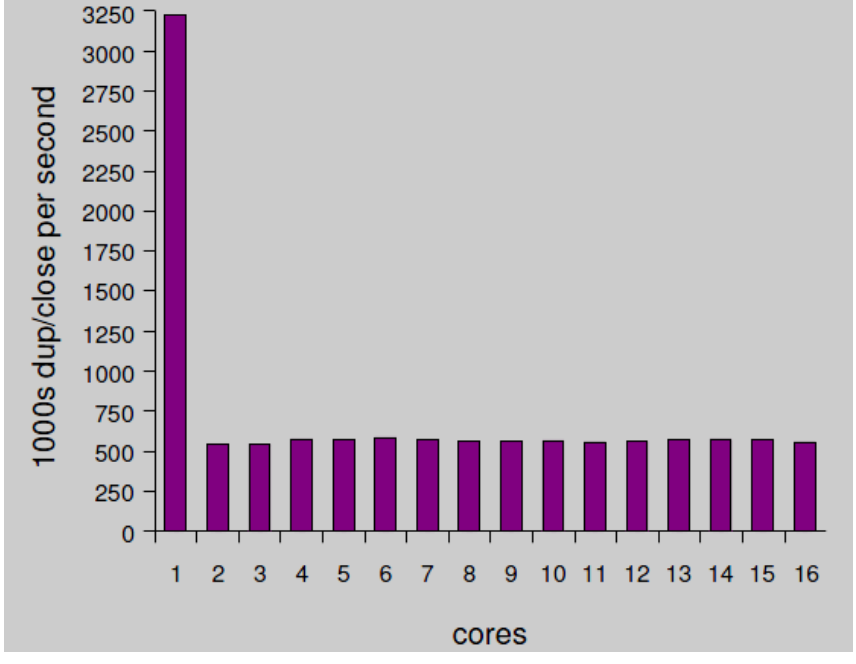
性能保证

- 多核多处理器硬件与特性
- 可扩展性问题导致性能断崖
- 系统软件设计如何利用特性

多核下应用的性能表现：理想 vs 现实



理想fd性能



实际fd性能

并行计算理论加速比（理想上限）

Amdahl's Law

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

同时执行的核心数

加速比

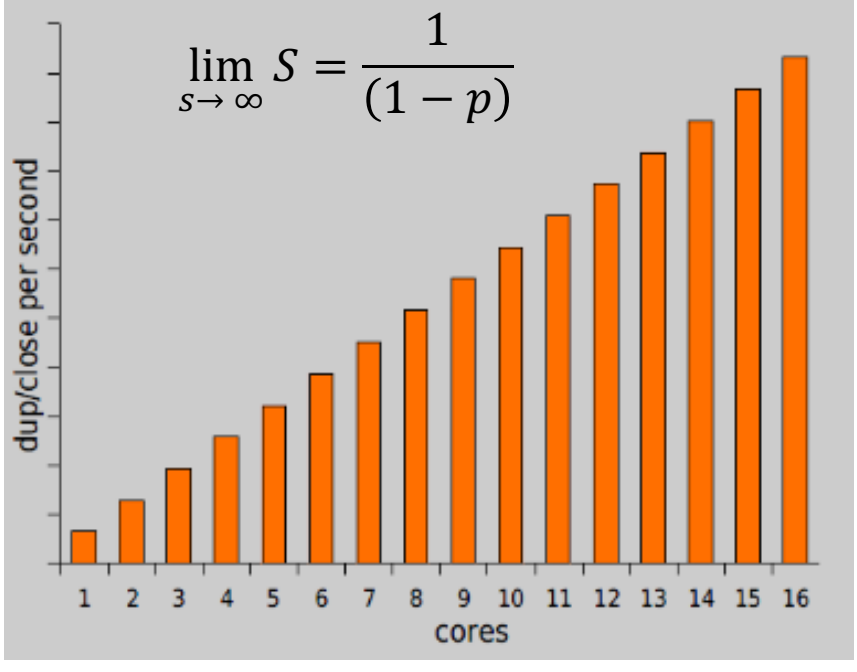
可以并行部分代码占比

当核心数增加时...

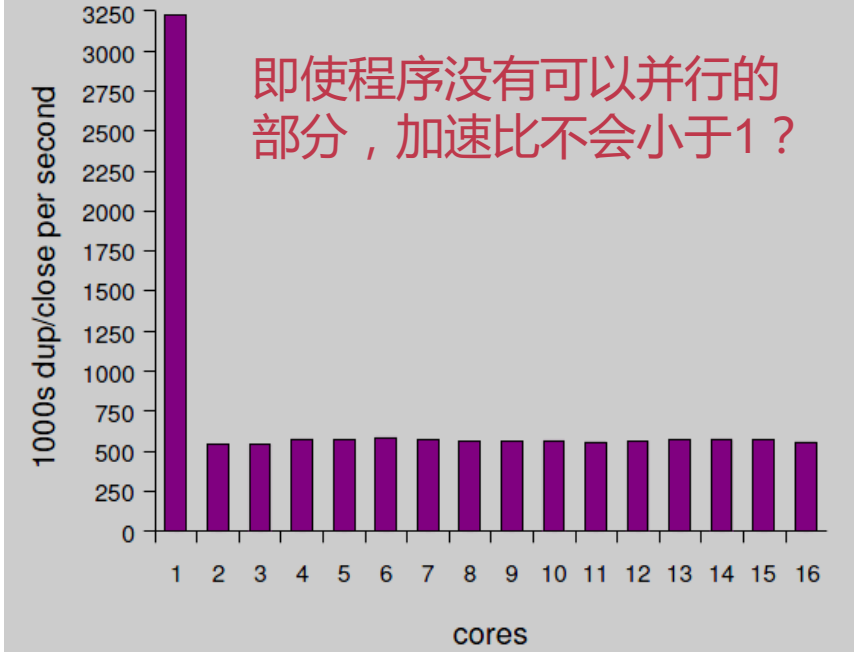
$$\lim_{s \to \infty} S = \frac{1}{(1-p)}$$

可以并行部分占比越多，这个程序理论上最大加速比越大

多核下应用的性能表现：理想 vs 现实



理想fd性能



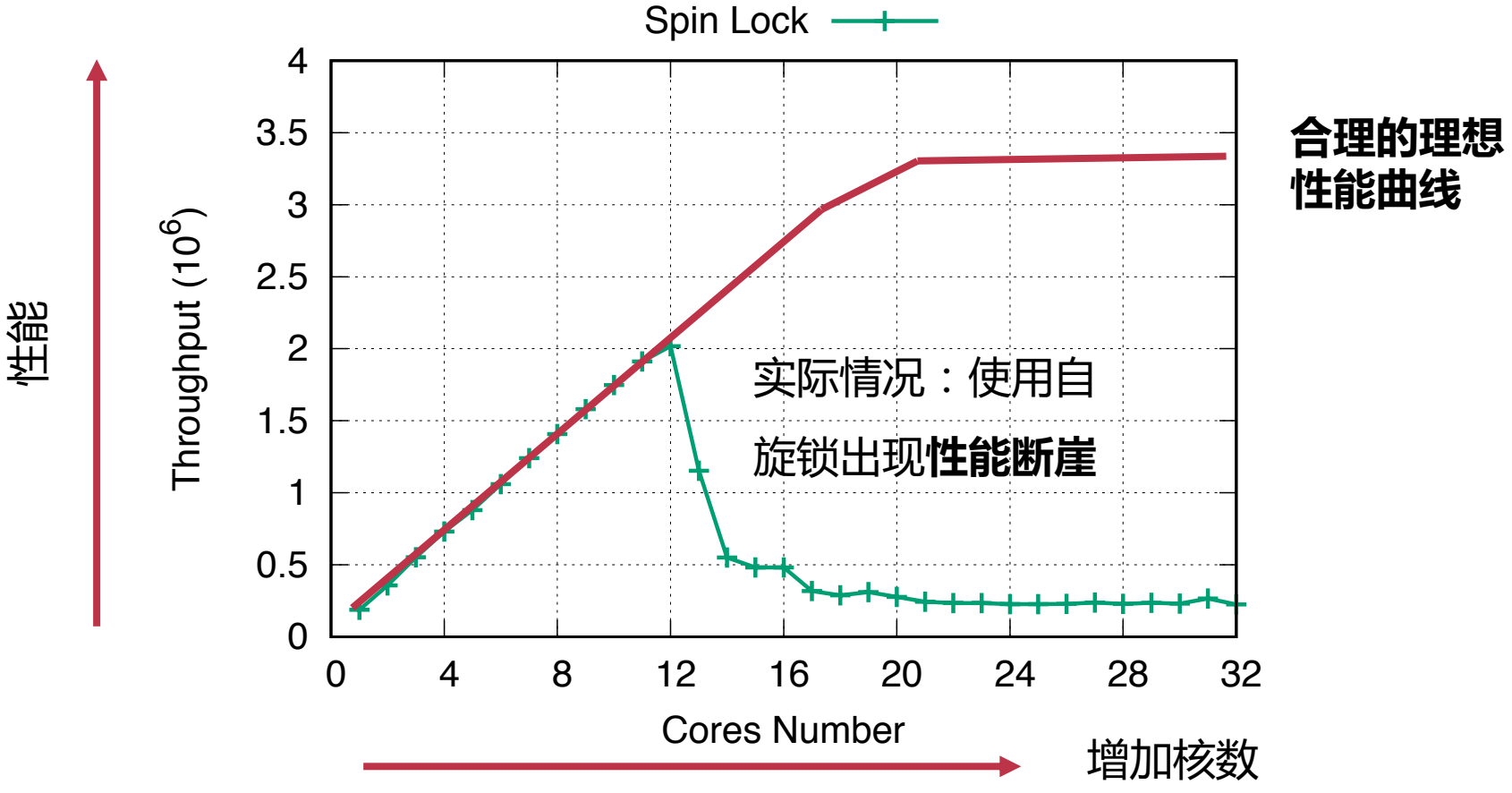
实际fd性能

互斥锁微基准测试

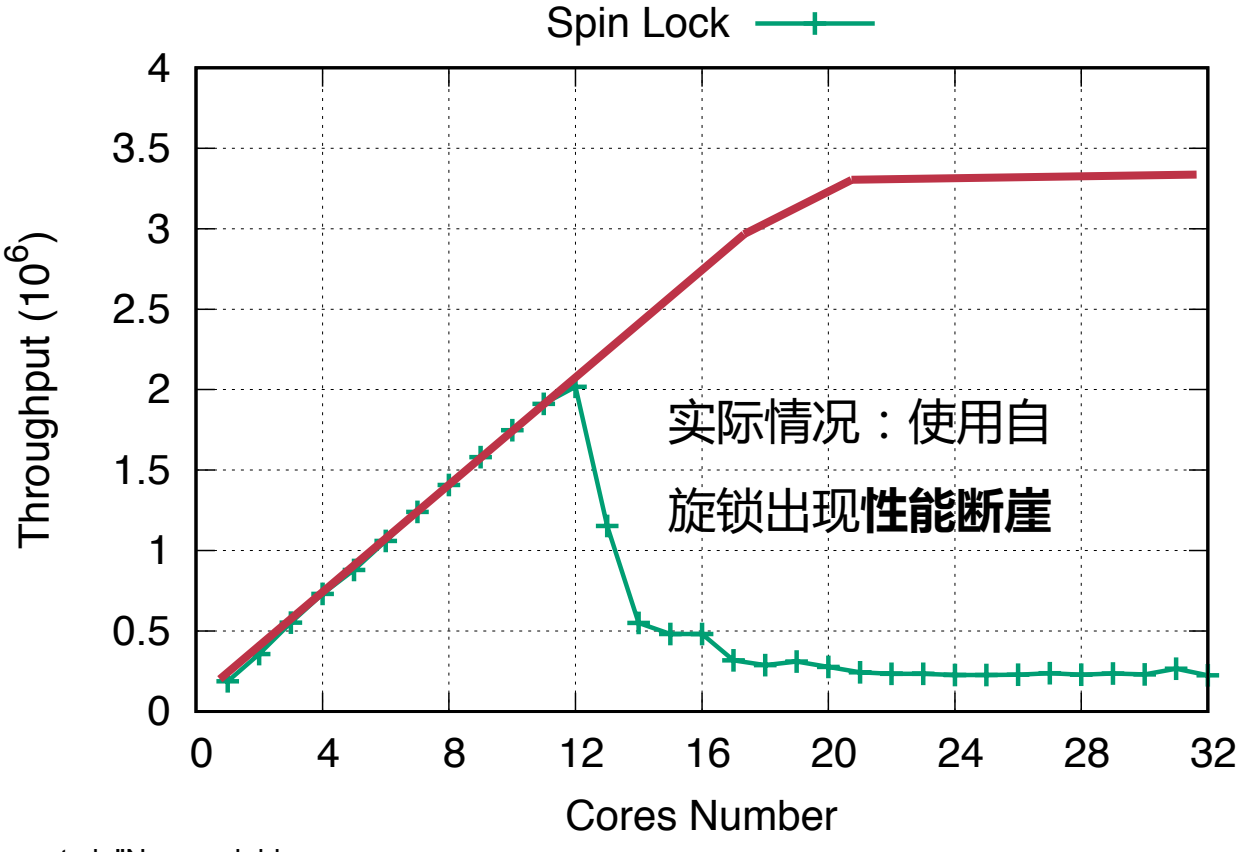
使用微基准测试来复现这个现象

```
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();
        }
}
```

可扩展性断崖


Spin Lock
性能
Throughput (10^6)
4
3.5
3
2.5
2
1.5
1
0.5
0
0
4
8
12
16
20
24
28
32
Cores Number
增加核数
实际情况：使用自
旋锁出现**性能断崖**
合理的理想
性能曲线

Non-scalable Locks are Dangerous!*



合理的理想性能曲线

为什么会出现这种情况？

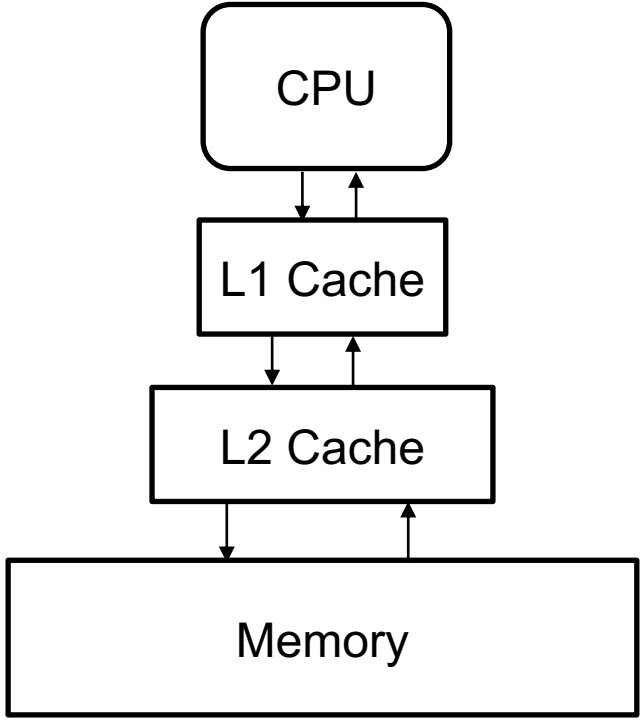
鲲鹏服务器上**互斥锁微基准测试**

* Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." *Proceedings of the Linux Symposium*. 2012.

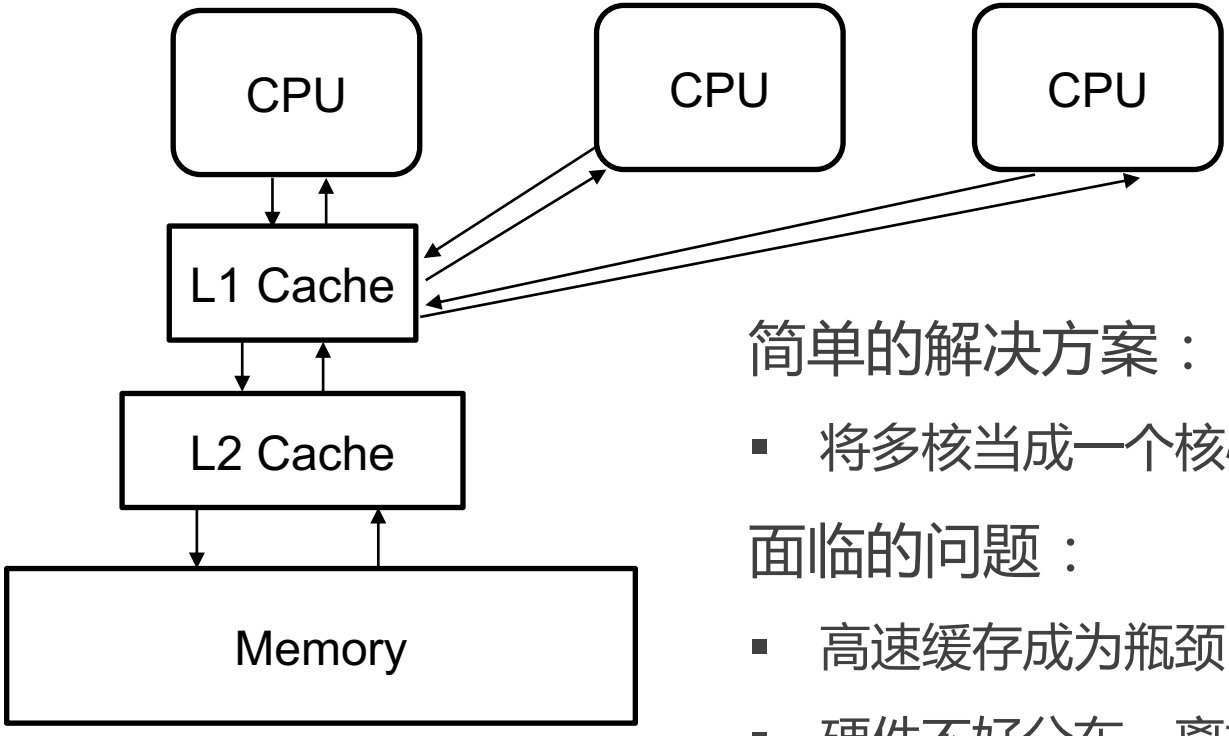
多核环境下的缓存

高速缓存（cache）回顾

- **多级缓存：**
  - 靠近CPU贵，速度快，容量小
  - 远离CPU便宜，速度慢，容量大
- **读操作：**
  - 逐层向下找
  - 没找到从内存中读取，放到缓存中
- **写操作：**
  - 直写/写回策略
  - 写入高速缓存，替换时写回



多处理器多核环境中的缓存结构



简单的解决方案：

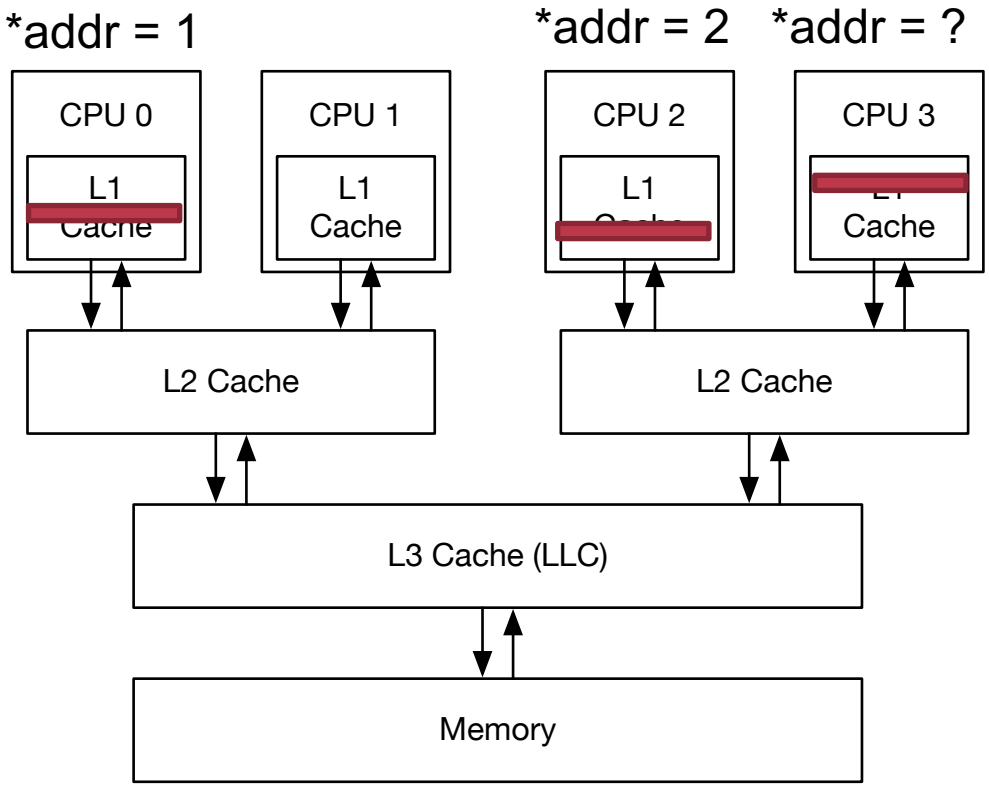
- 将多核当成一个核心，共享缓存设计

面临的问题：

- 高速缓存成为瓶颈，单点竞争
- 硬件不好分布，离核心远，速度减慢

多核环境中的缓存结构

- 多级缓存：
  - 每个核心有自己的**私有**高速缓存（L1 Cache）
  - 多个核心共享一个**二级**高速缓存（L2 Cache）
  - 所有核心共享一个**最末级**高速缓存（LLC）
- 非一致缓存访问（NUCA）
- 数据一致性问题



一个典型多核系统高速缓存架构*

*可以有其他选择，大部分多核系统采用该架构

缓存一致性

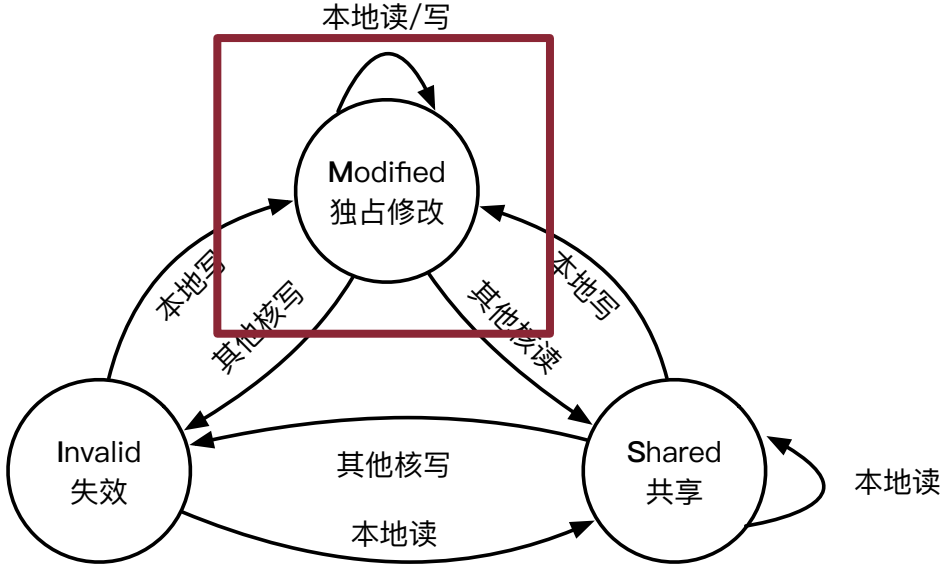
- 保证不同核心对同一地址的值达成共识
- 多种缓存一致性**协议**：窥探式/**目录式缓存一致性**协议

具体怎么做？

- 缓存行处于不同状态（MSI状态）
- 不同状态之间迁移
- 所有地读/写缓存行操作遵循协议流程

缓存一致性：MSI状态迁移

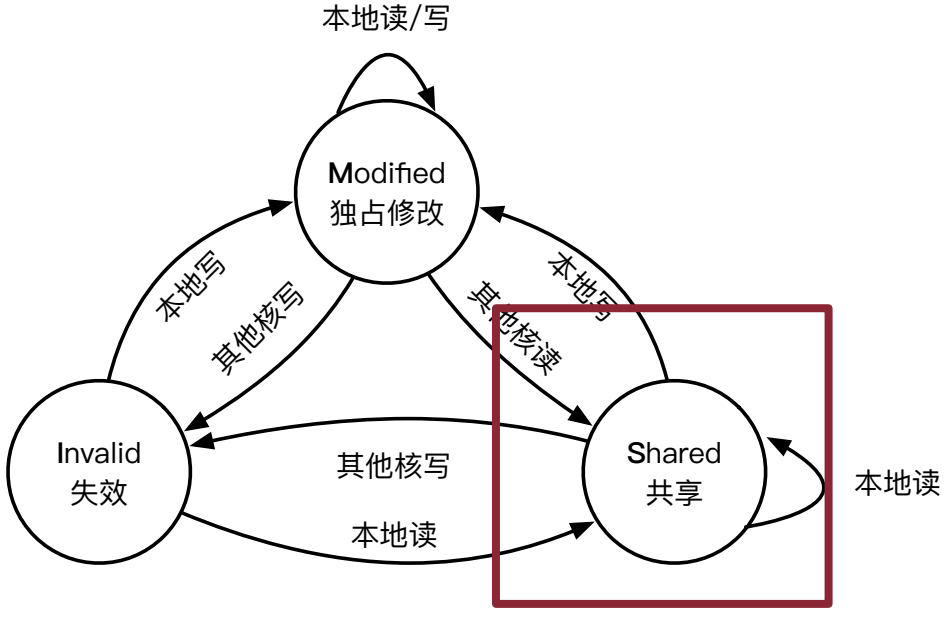
- 独占修改（**M**odified）
  - 该核心独占拥有缓存行
  - 本地可**读**可**写**
  - 其他核**读**需要迁移到**共享**
  - 其他核**写**需要迁移到**失效**



每核心每缓存行状态

缓存一致性：MSI状态迁移

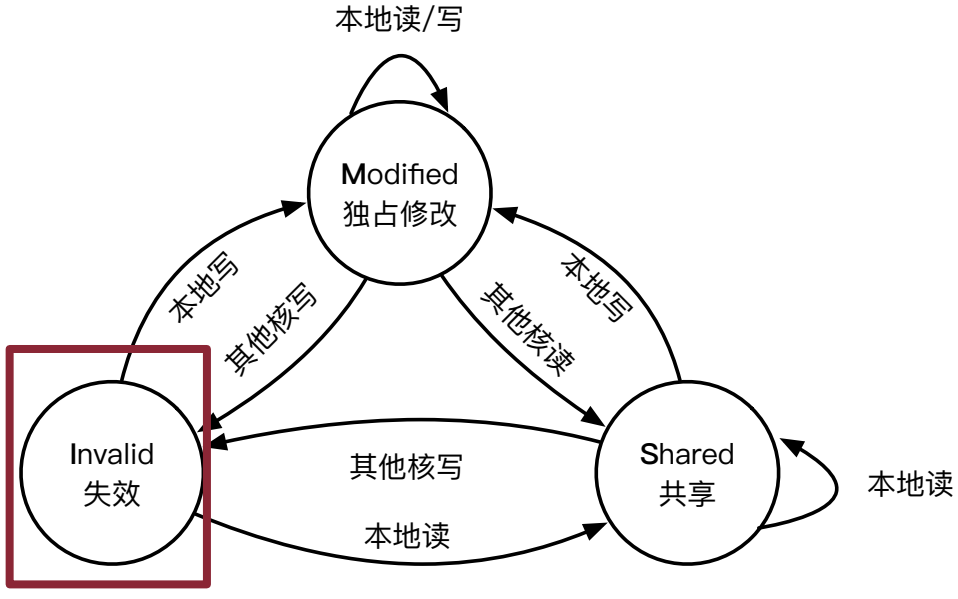
- 共享（**S**hared）
  - 可能多个核同时有缓存行的拷贝
  - 本地可**读**
  - 本地**写**需要迁移到**独占修改**，并使其他核该缓存行**失效**
  - 其他核**写**需要迁移到**失效**



每核心每缓存行状态

缓存一致性：MSI状态迁移

- 失效（Invalid）
  - 本地缓存行失效
  - **本地不能读/写**缓存行
  - 本地**读**需要迁移到**共享**，并使其他核该缓存行迁移到**共享**
  - 本地**写**需要迁移到**独占修改**，并使其他核心该缓存行**失效**

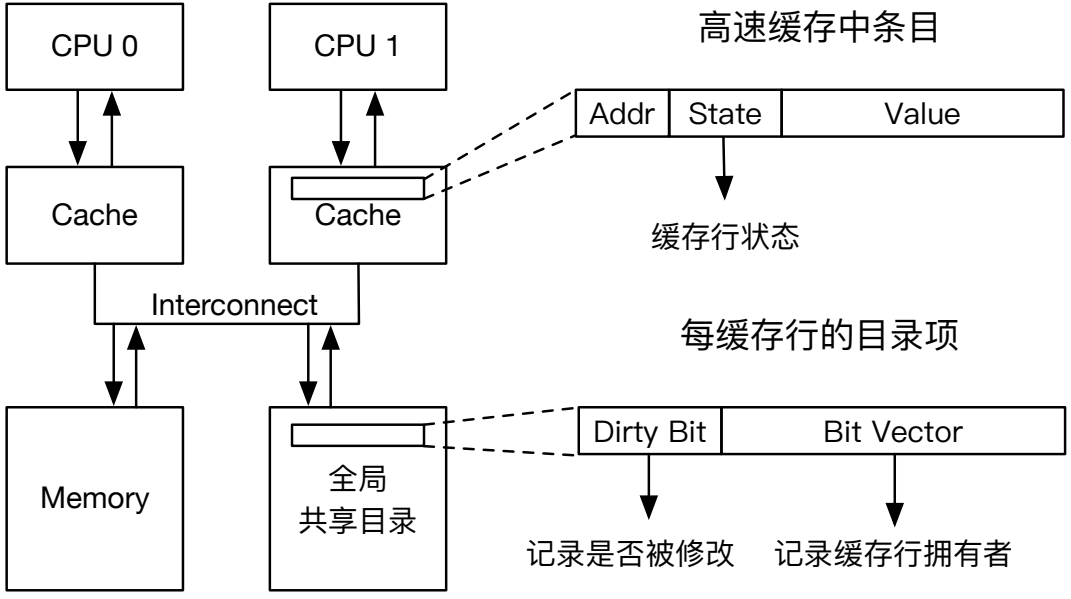


每核心每缓存行状态

缓存一致性：全局目录项

如何通知其他核心需要迁移缓存行状态？

全局目录项：记录缓存行在不同核上的状态，通过总线通讯



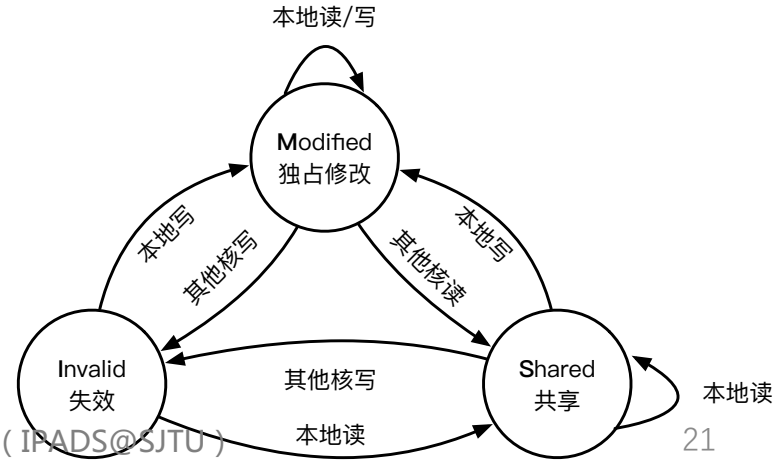
目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
x	S	666	S	666	S	666

全局共享目录项

	Dirty	Bit Vector		
x	0	1	1	1



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

执行ST X,233

CPU 0

	状态	内容
X	S	666

CPU 1

状态	内容
S	666

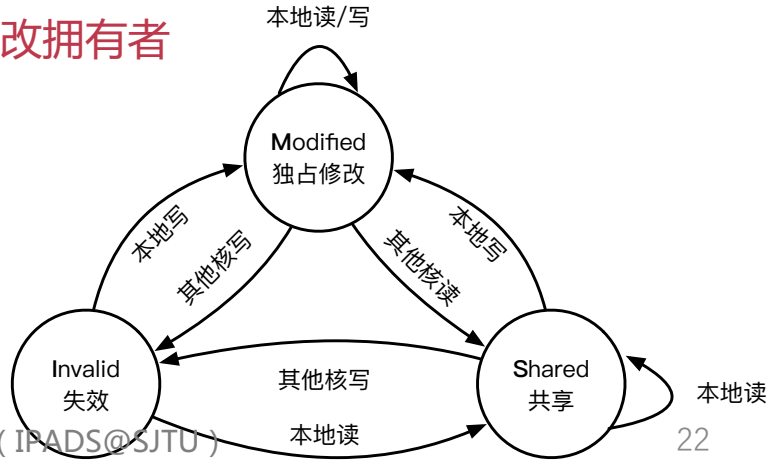
CPU 2

状态	内容
S	666

查看目录项，修改Dirty位，更改拥有者

全局共享目录项

	Dirty	Bit Vector		
X	1	1	0	0



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

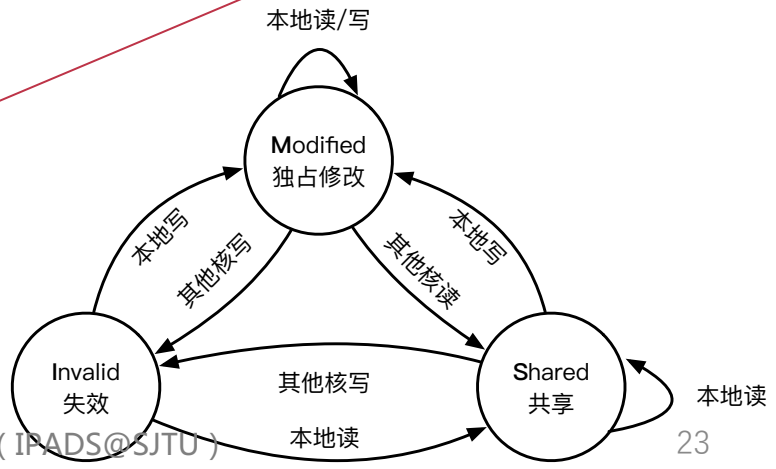
执行ST X,233

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
X	S	666	I	666	I	666

更新CPU1，CPU2目录项，使其失效

全局共享目录项

	Dirty	Bit Vector		
X	1	1	0	0



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

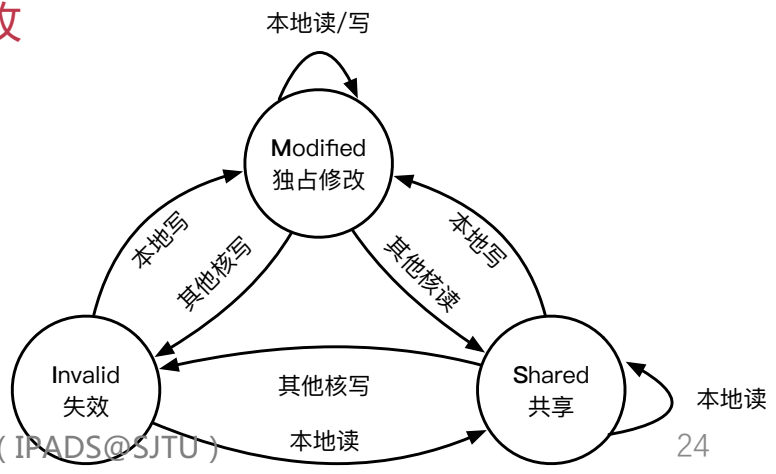
执行ST X,233

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
X	M	233	I	666	I	666

回复CPU0通知其可以独占修改

全局共享目录项

	Dirty	Bit Vector		
X	1	1	0	0



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

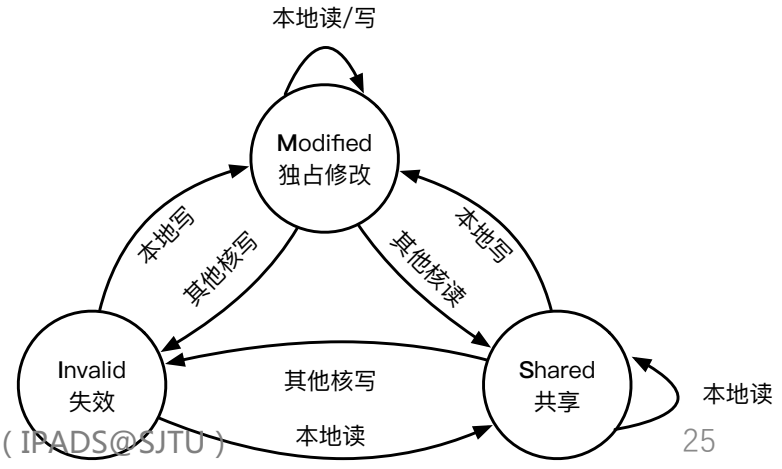
执行ST X,888

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
x	M	233	I	666	I	666

查看目录项，修改Dirty位，更改拥有者

全局共享目录项

	Dirty	Bit Vector		
x	1	0	1	0



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

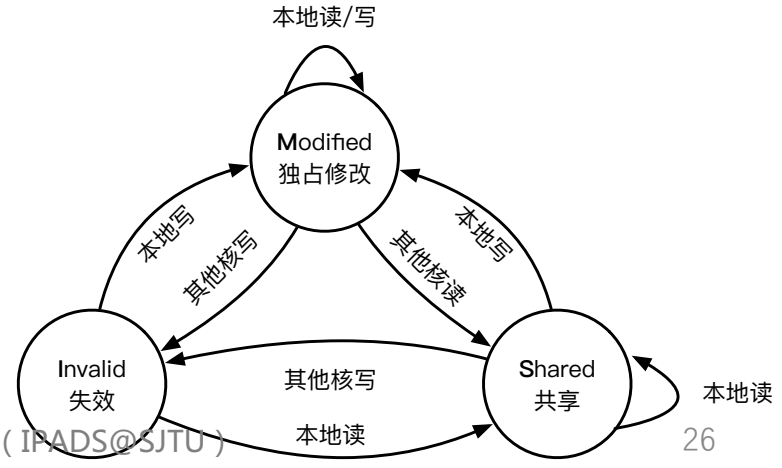
执行ST X,888

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
X	I	233	I	666	I	666

更新CPU0目录项，使其失效

全局共享目录项

	Dirty	Bit Vector		
X	1	0	1	0



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

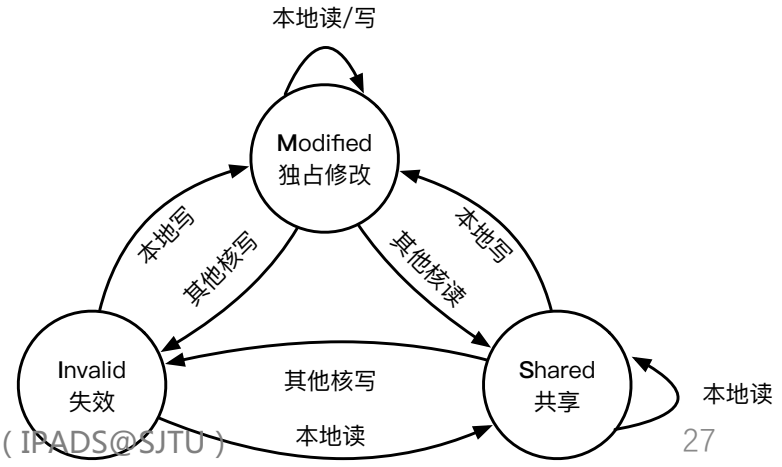
执行ST X,888

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
X	I	233	M	888	I	666

回复CPU1可以独占修改

全局共享目录项

	Dirty	Bit Vector		
X	1	0	1	0



思考：233是不是就丢失了？正确吗？

目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

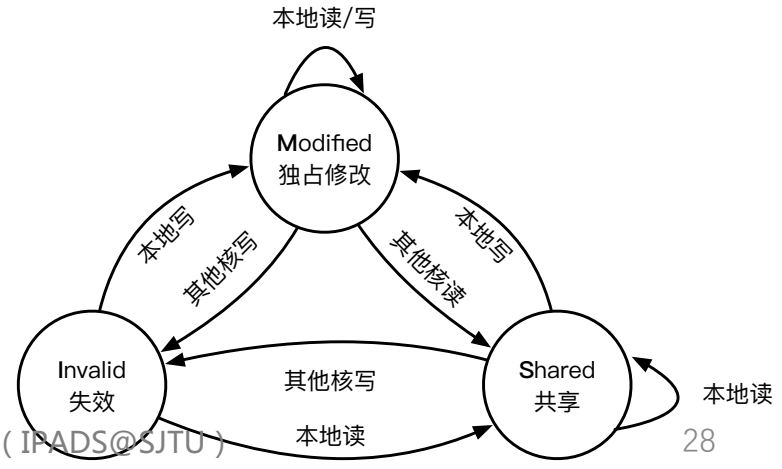
执行LD x

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
x	I	233	M	888	I	666

发现失效，去目录找谁拥有

全局共享目录项

	Dirty	Bit Vector		
x	1	0	1	0



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

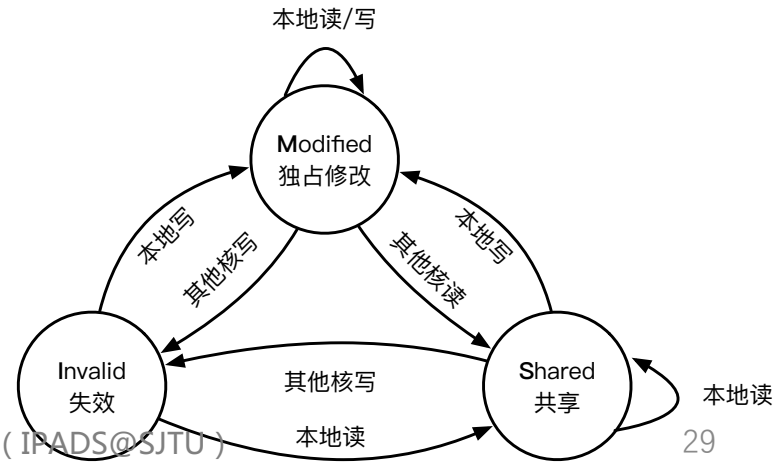
执行LD x

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
x	I	233	S	888	I	666

更新目录，并让拥有者给cpu0发送最新的值，迁移状态

全局共享目录项

	Dirty	Bit Vector		
x	0	1	1	0



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

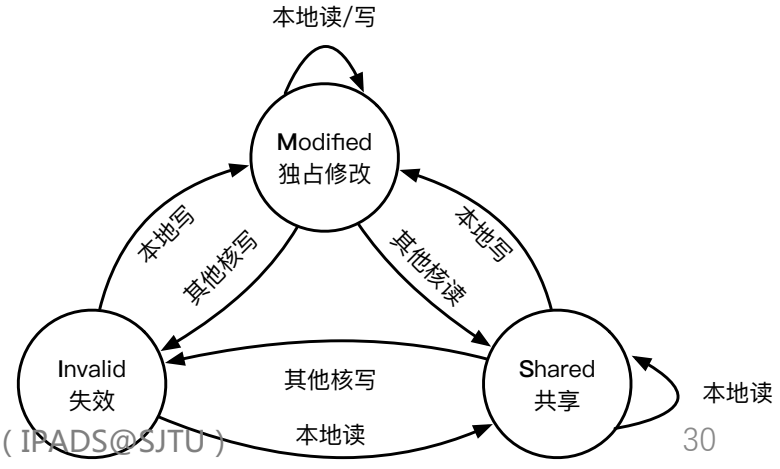
执行LD x

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
x	S	888	S	888	I	666

转发最新的值

全局共享目录项

	Dirty	Bit Vector		
x	0	1	1	0



目录式缓存一致性：一个例子

关注变量x所在缓存行，3个CPU，一个全局共享目录

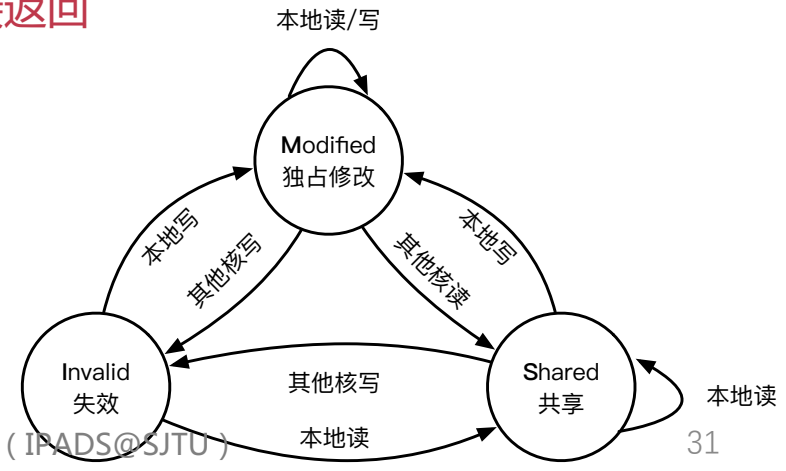
执行LD x

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
x	S	888	S	888	I	666

为共享状态，直接返回

全局共享目录项

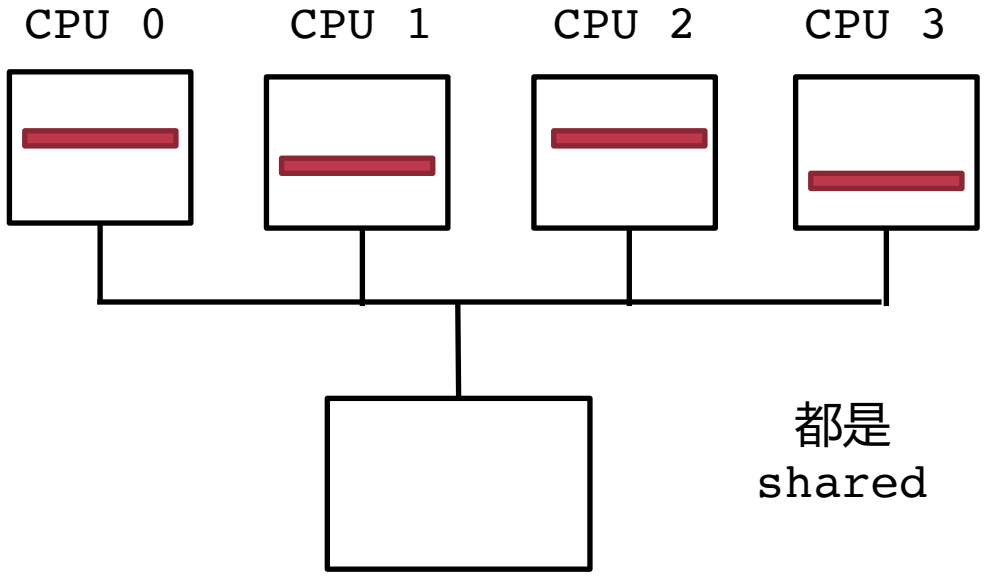
	Dirty	Bit Vector		
x	0	1	1	0



回到可扩展性断崖

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

void unlock(int *lock) {
    *lock = 0;
}
```

自旋锁实现



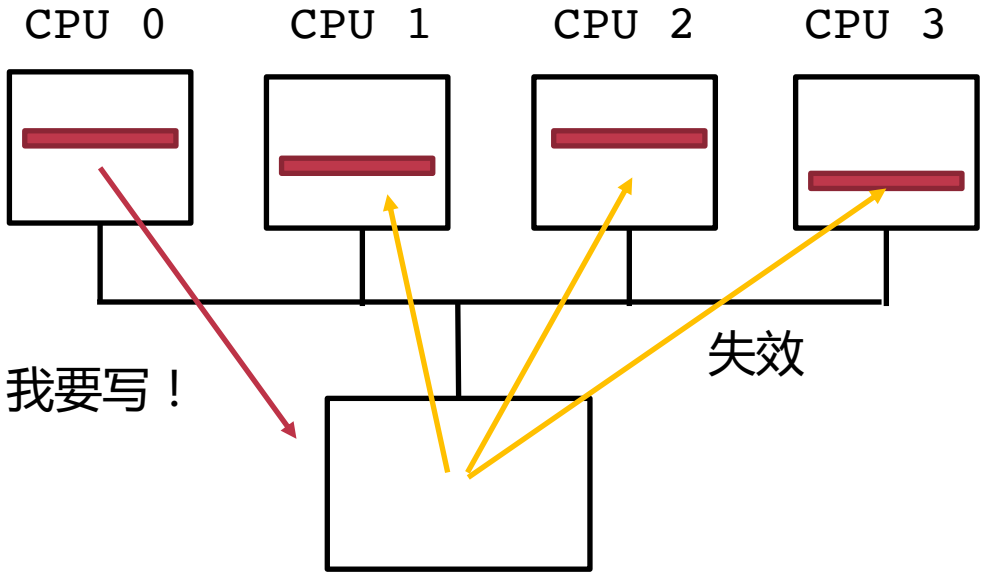
全局目录项

Lock所在缓存行状态

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



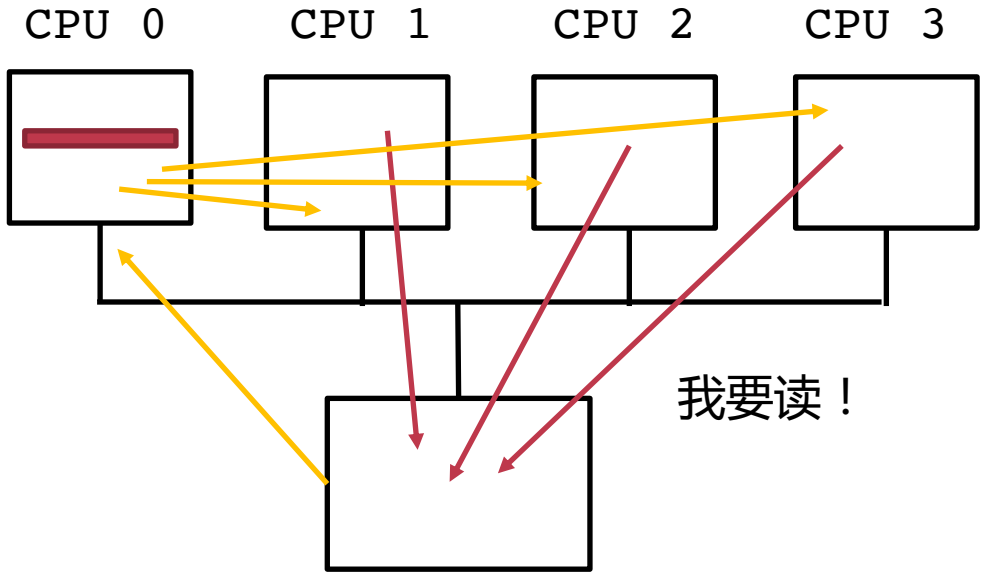
全局目录项

Lock所在缓存行状态

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



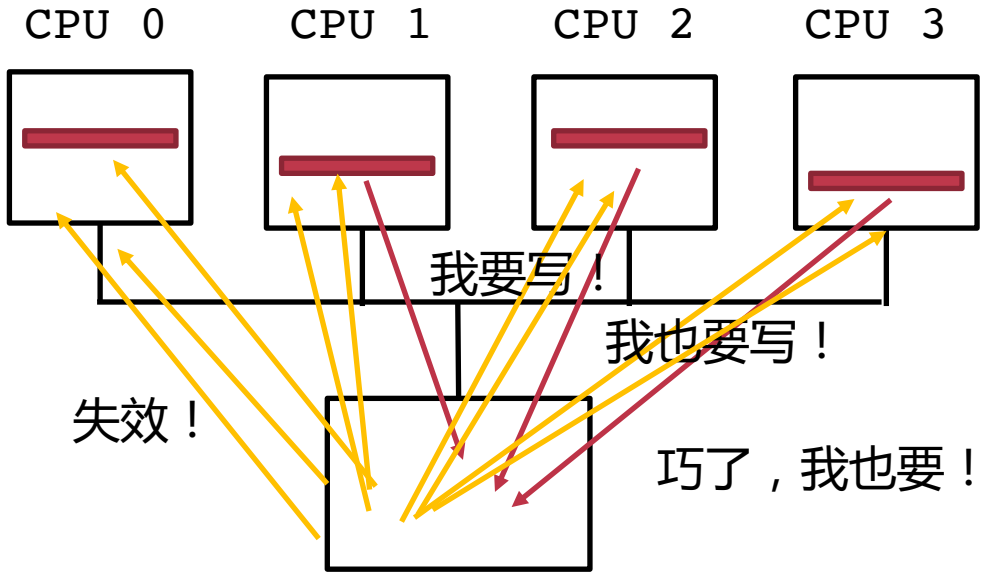
全局目录项

Lock所在缓存行状态

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

可扩展性断崖背后的原因

Amdahl's Law

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

同时执行的核心数

加速比

可以并行部分代码占比

当核心数增加时...

$$\lim_{s \to \infty} S = \frac{1}{(1-p)}$$

对**单一缓存行**的竞争导致**严重的性能**开销：公式中的**P**急剧下降，加速比下降

如何解决可扩展性问题

Simple fix：避免对单一缓存行的高度竞争 – Back-off 策略

```
void back_off(int time) {
    for (volatile int i=0; i<time; i++)
          cpu_relax();
}

void lock(int *lock) {
    while(atomic_CAS(lock, 0, 1) != 0)
         back_off(DEFAULT_TIME);
}
```

使用Back-off策略

思考：这样写能解决问题吗？

会有什么样的问题？

如何解决可扩展性问题

Simple fix：避免对单一缓存行的高度竞争 – Back-off 策略

```
void back_off(int time) {
    for (volatile int i=0; i<time; i++)
          cpu_relax();
}

void lock(int *lock) {
    while(atomic_CAS(lock, 0, 1) != 0)
         back_off(DEFAULT_TIME);
}
```

使用Back-off策略

思考：这样写能解决问题吗？

会有什么样的问题？

等待**相同**时间，**同时**停止等待，

同时开始下一轮竞争！

- 随机时间
- 指数后退

Back-off 是否完全解决可扩展性问题

Linus' Response

So I claim:

真实硬件/场景很难触发

- it's *really* hard to trigger in real loads on common hardware.

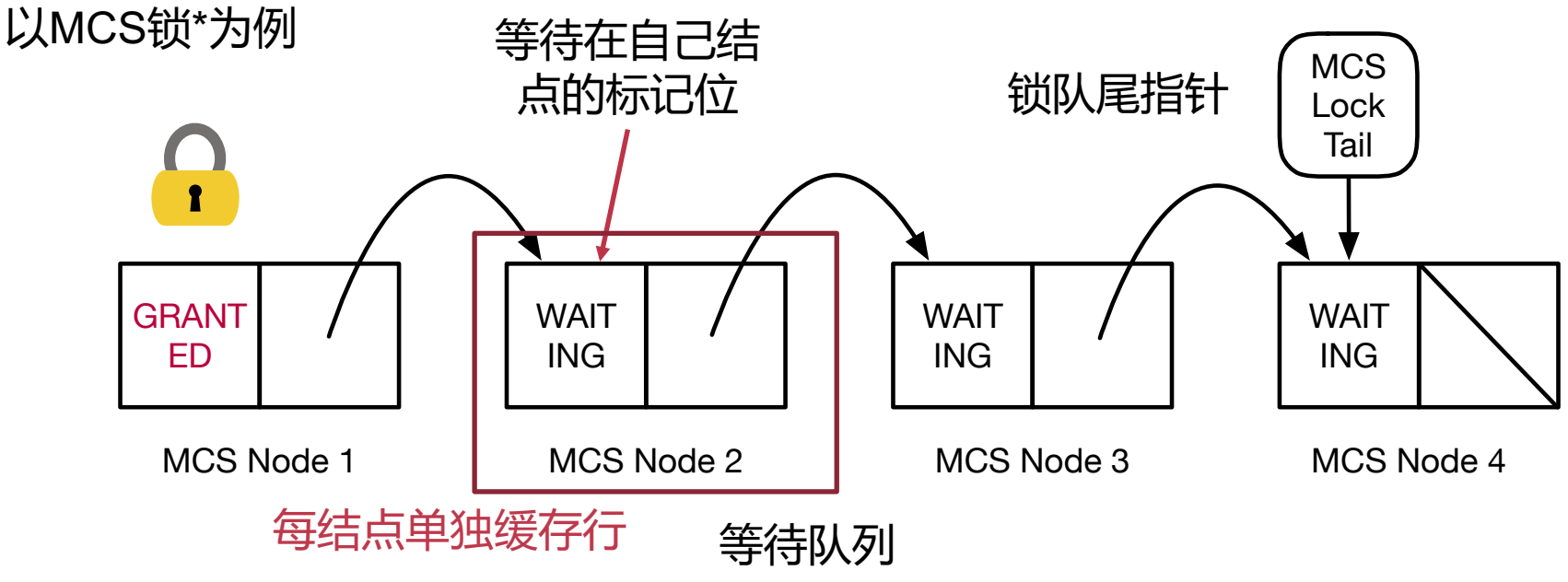
- if it does trigger in any half-way reasonably common setup (hardware/software), we most likely should work really hard at fixing the underlying problem, not the symptoms. 治标不治本

- we absolutely should *not* pessimize the common case for this

*http://linux-kernel.2935.n7.nabble.com/PATCH-v5-0-5-x86-smp-make-ticket-spinlock-proportional-backoff-w-auto-tuning-td596698i20.html

如何解决可扩展性问题：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



*Mellor-Crummey J M, Scott M L. Algorithms for scalable synchronization on shared-memory multiprocessors[J]. TOCS, 1991, 9(1): 21-65.

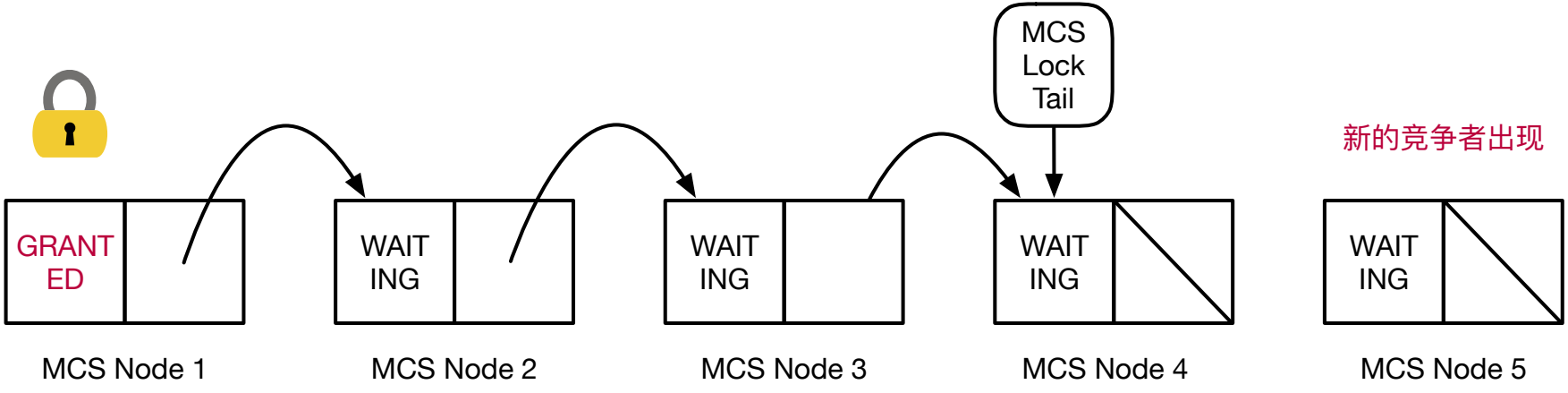
MCS实现示例

```
void *XCHG(void **addr, void *new_value) {
        void *tmp = *addr;
        *addr = new_value;
        return tmp;
}
```

```
void unlock(struct mcs_lock *lock) {
        struct mcs_node *me = &my_node;

        barrier();
        if (!me->next) {
                /* Try to free the lock */
                if (atomic_CAS(&lock->tail, me, 0) ==
                 ↪  me)
                        return;
                while (!me->next)
                        ;
        }
        me->next->flag = GRANTED;
}
```

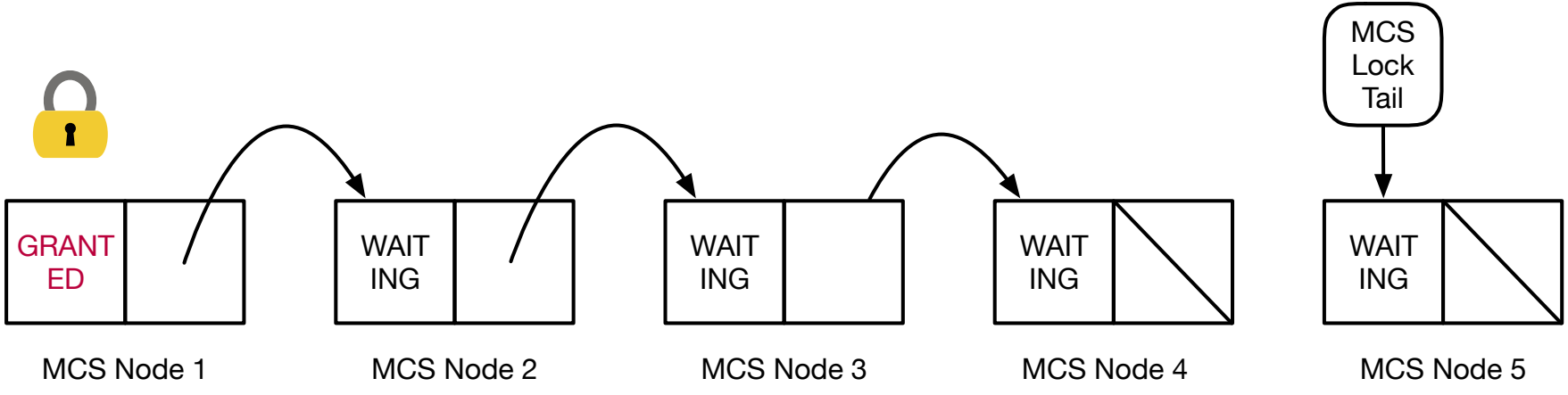
```
struct mcs_node {
        volatile struct mcs_node *next;
        volatile int flag;
} __attribute__((aligned(CACHELINE_SZ)));

struct mcs_lock {
       struct mcs_node *tail;
}

__thread struct mcs_node my_node;

void lock(struct mcs_lock *lock) {
        struct mcs_node *me = &my_node;
        struct mcs_node *tail = 0;

        me->next = NULL;
        me->flag = WAITING;

        tail = atomic_XCHG(&lock->tail, 0, me);

        if (tail) {
                barrier();
                tail->next = me;
                while (me->flag != GRANTED)
                        ; /* Busy waiting */
        } else {
                me->flag = GRANTED;
        }
        barrier();
}
```

MCS锁：新的竞争者加入等待队列



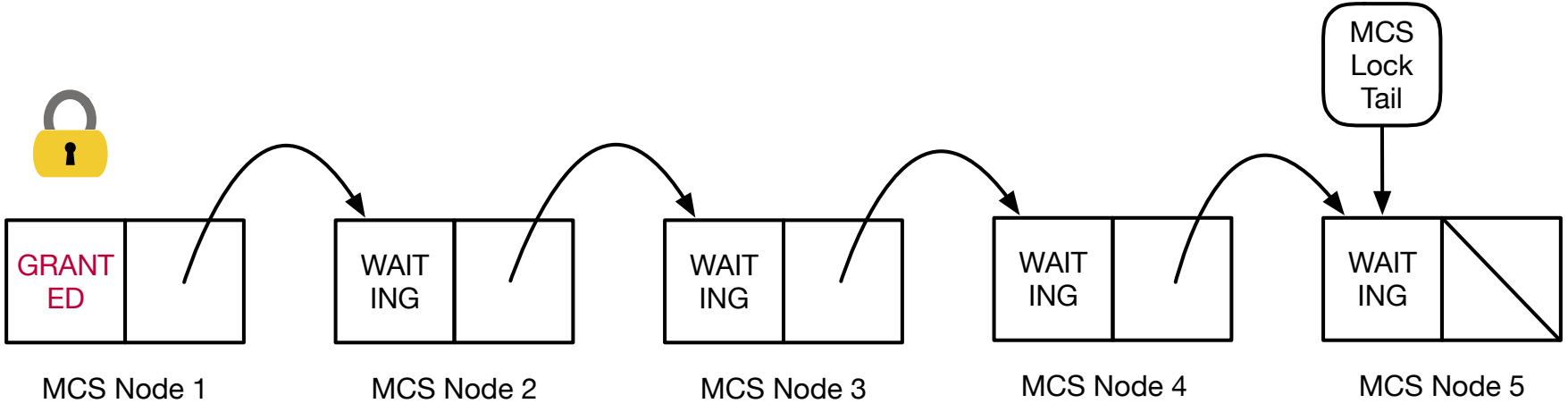
先填写自己结点的内容

MCS锁：新的竞争者加入等待队列



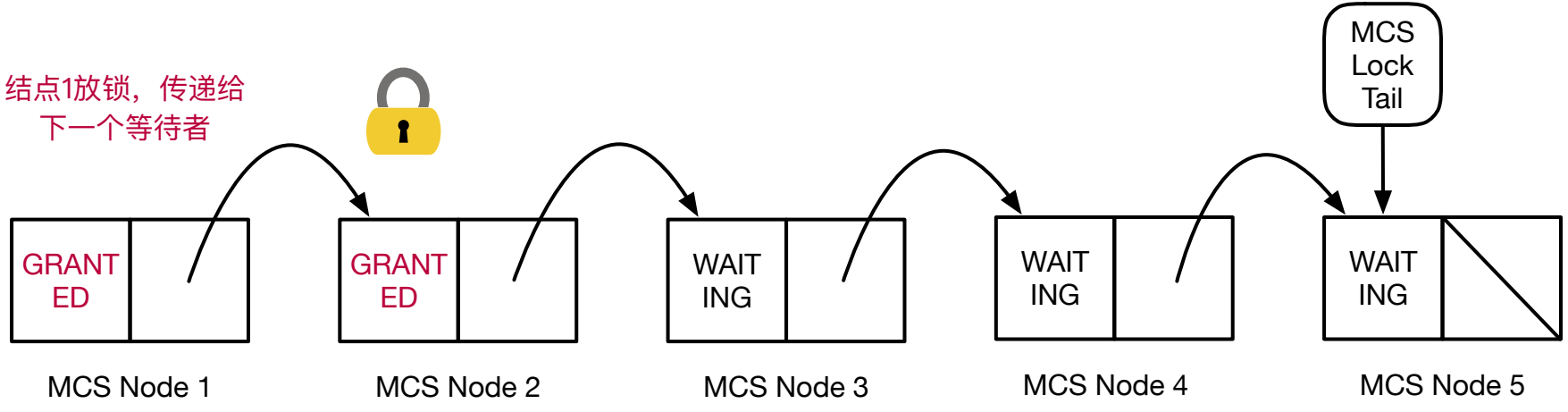
通过**原子操作**更新MCS锁的尾指针

MCS锁：新的竞争者加入等待队列

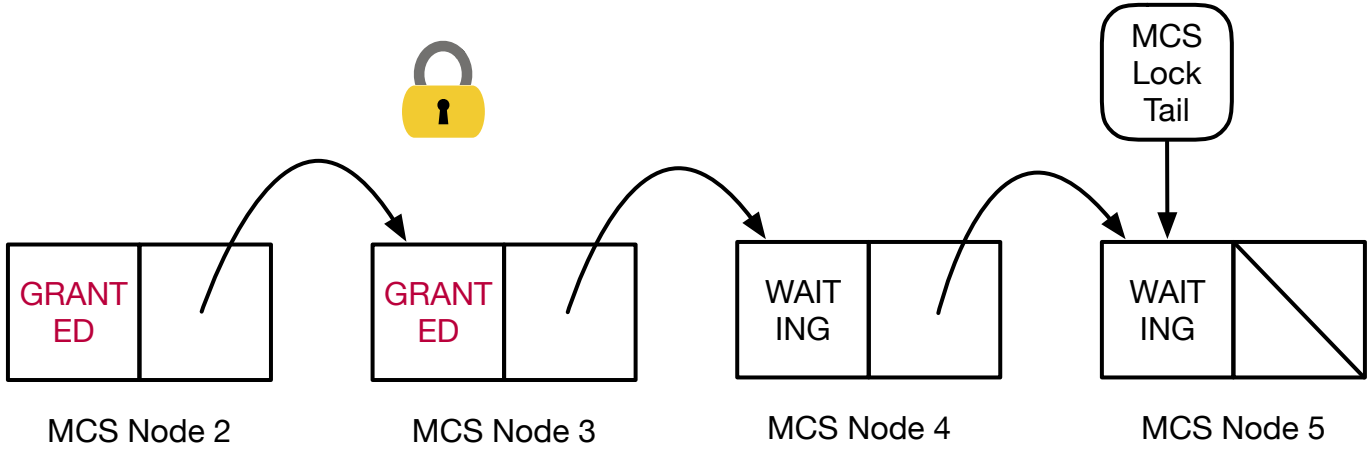


最后链接入等待队列

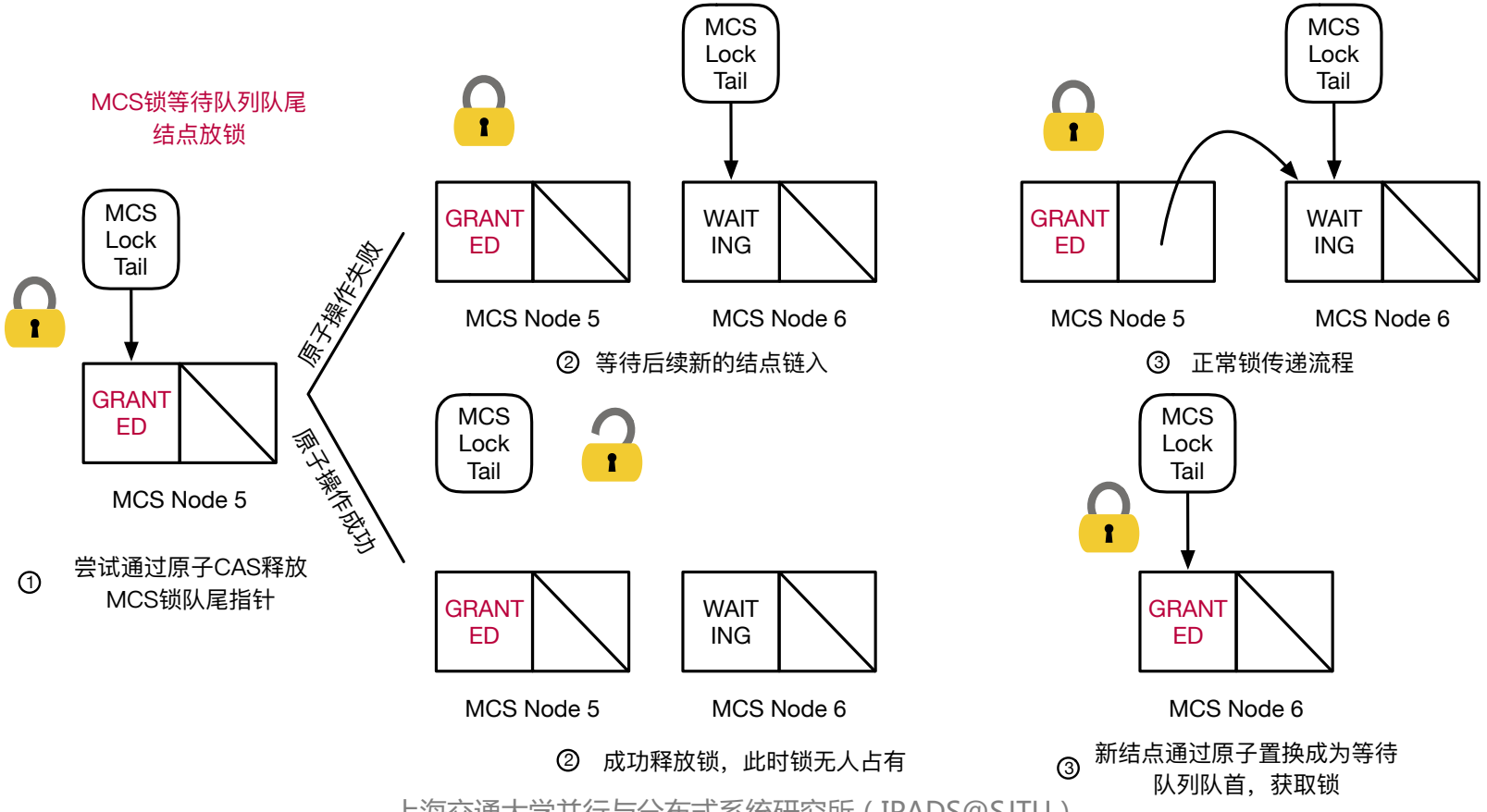
MCS锁：锁持有者的传递



MCS锁：锁持有者的传递

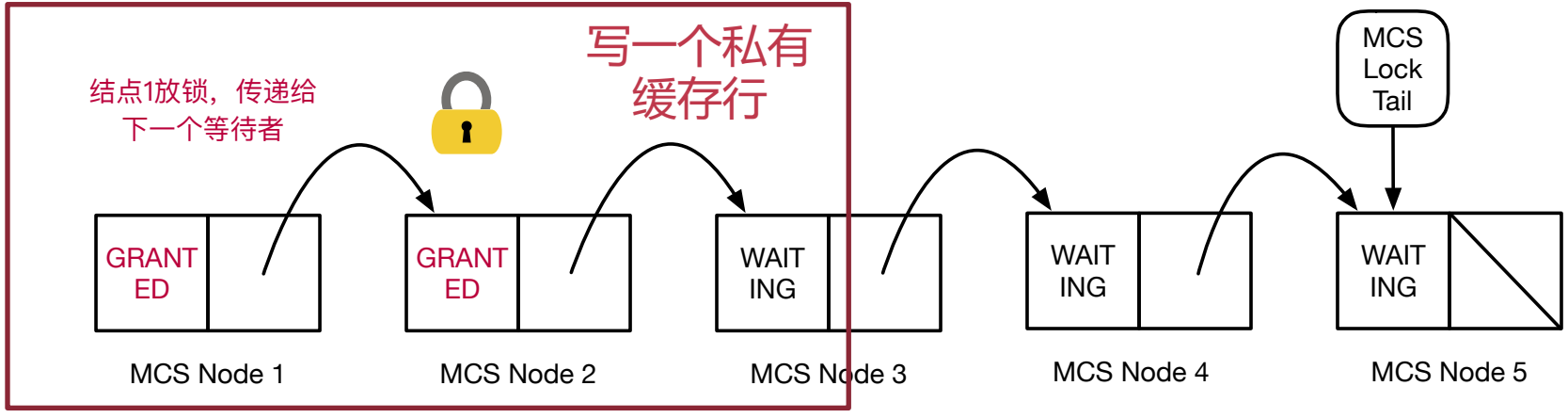


MCS锁：放锁流程

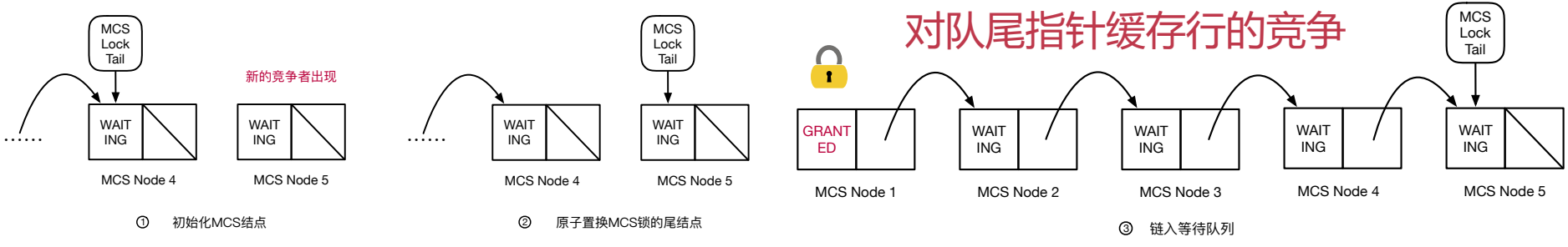

MCS锁等待队列队尾
结点放锁
MCS
Lock
Tail
GRANT
ED
MCS Node 5
原子操作失败
原子操作成功
① 尝试通过原子CAS释放
MCS锁队尾指针
MCS
Lock
Tail
GRANT
ED
WAIT
ING
MCS Node 5
MCS Node 6
② 等待后续新的结点链入
MCS
Lock
Tail
GRANT
ED
WAIT
ING
MCS Node 5
MCS Node 6
② 成功释放锁，此时锁无人占有
MCS
Lock
Tail
GRANT
ED
WAIT
ING
MCS Node 5
MCS Node 6
③ 正常锁传递流程
MCS
Lock
Tail
GRANT
ED
MCS Node 6
③ 新结点通过原子置换成为等待
队列队首，获取锁

MCS锁：性能分析

不再会高频竞争全局缓存行



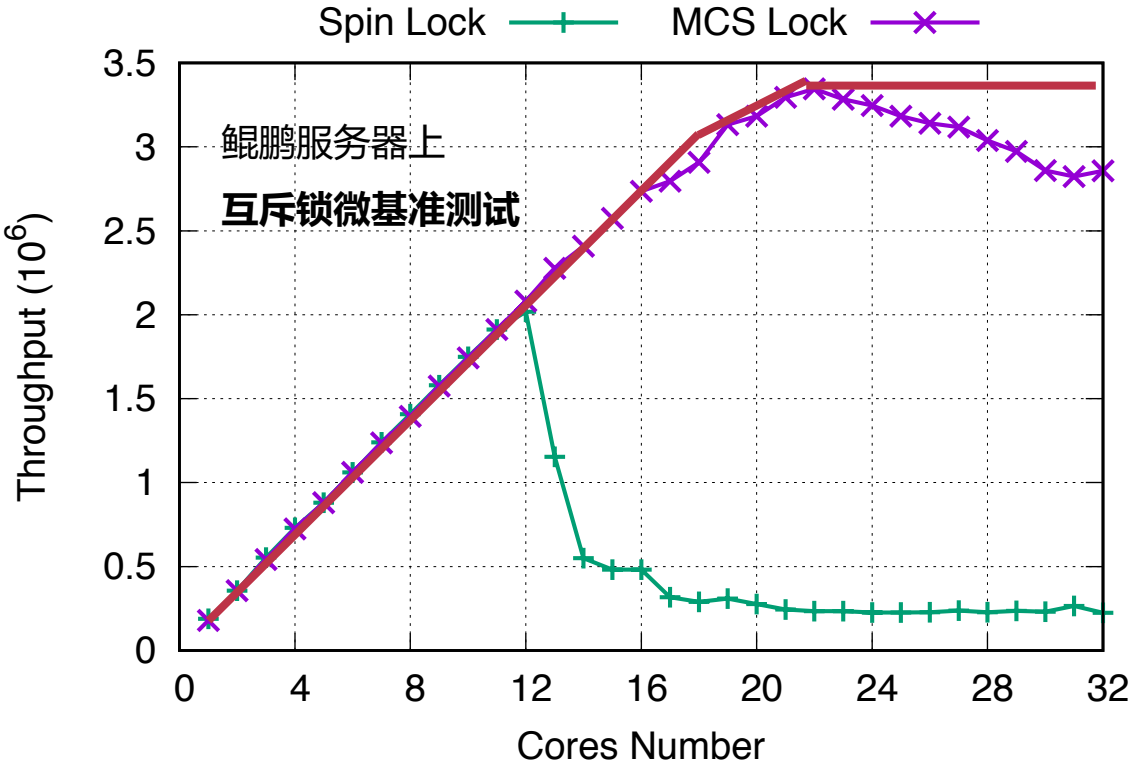
高竞争程度时的**关键路径**



高竞争程度时**关键路径之外**

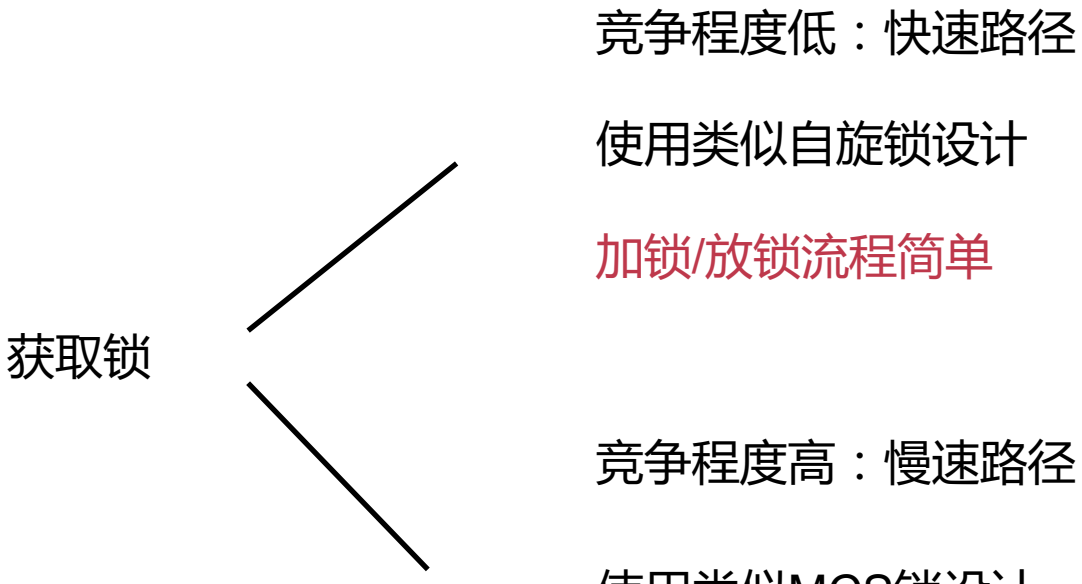
MCS锁：性能分析

核心思路：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!

Linux Kernel中的可扩展锁：QSpinlock*

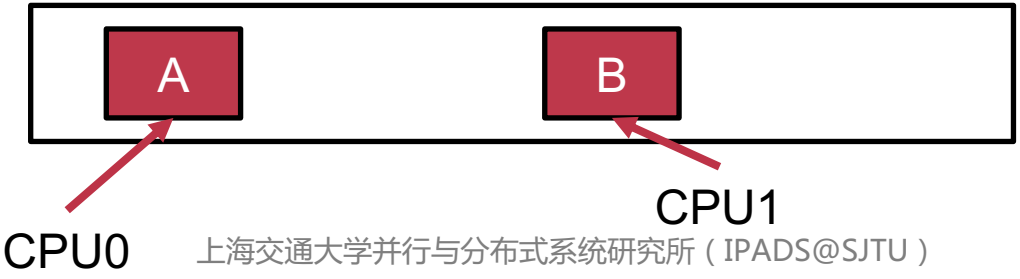


*qspinlock: Introducing a 4-byte queue spinlock implementation
https://lwn.net/Articles/561775/

系统软件开发者视角下的缓存一致性

- 多核硬件中面对私有高速缓存**硬件提供**的**正确性**设计
- 对软件开发者**透明**
- 系统软件开发者视角：
  1. 多个核心对于**同一缓存行**的高频竞争将会面临严重的性能开销
  2. **虚假共享（False Sharing）**在多核情况下是致命的

某一缓存行



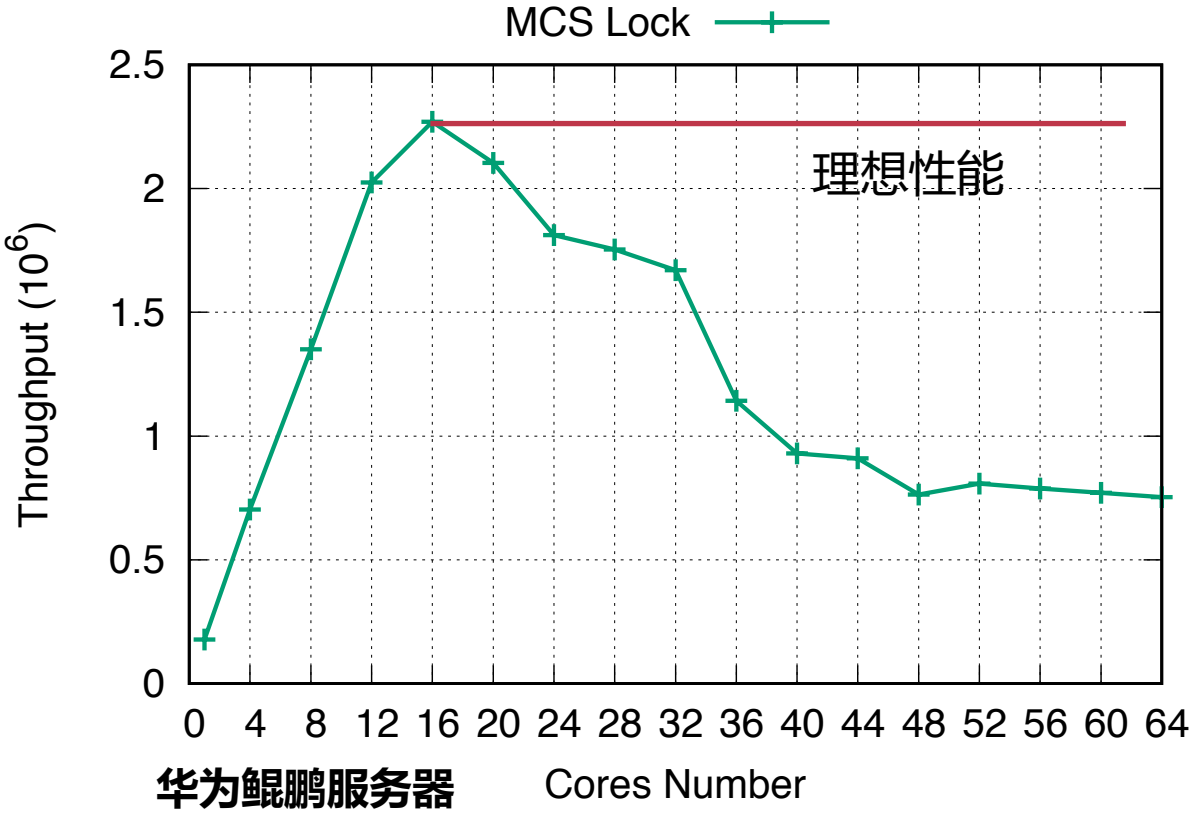
非一致内存访问，AKA，NUMA

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锁可扩展性？

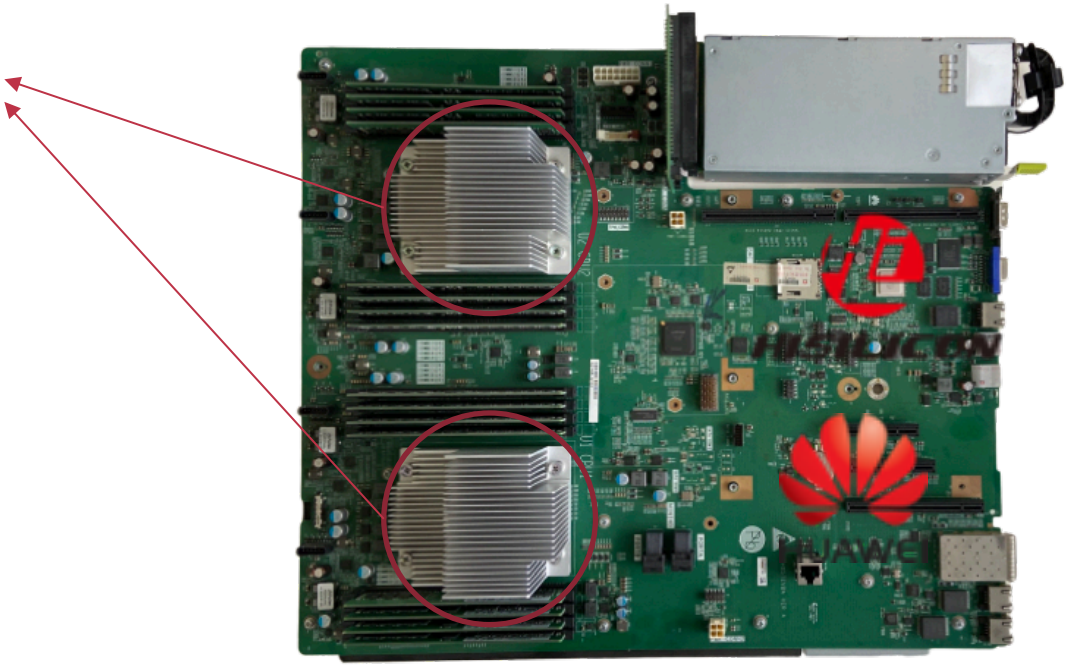


与之前测试有两点不同：

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



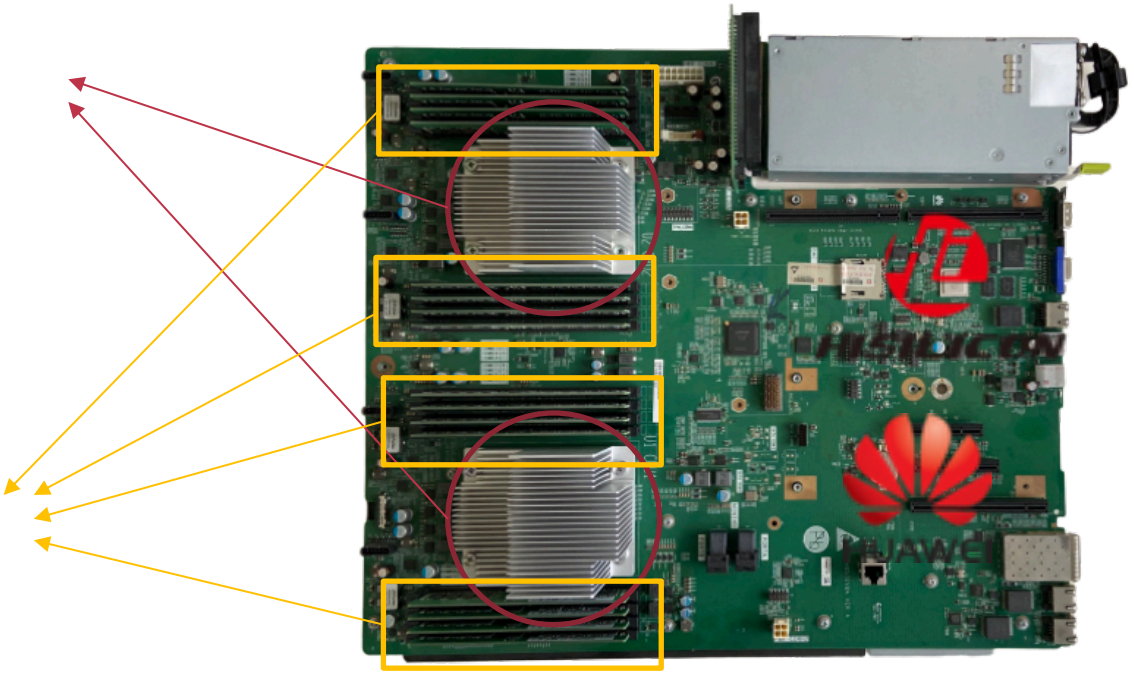
华为鲲鹏服务器（ARM架构）

常见于多处理器（多插槽）机器

鲲鹏服务器

2*处理器

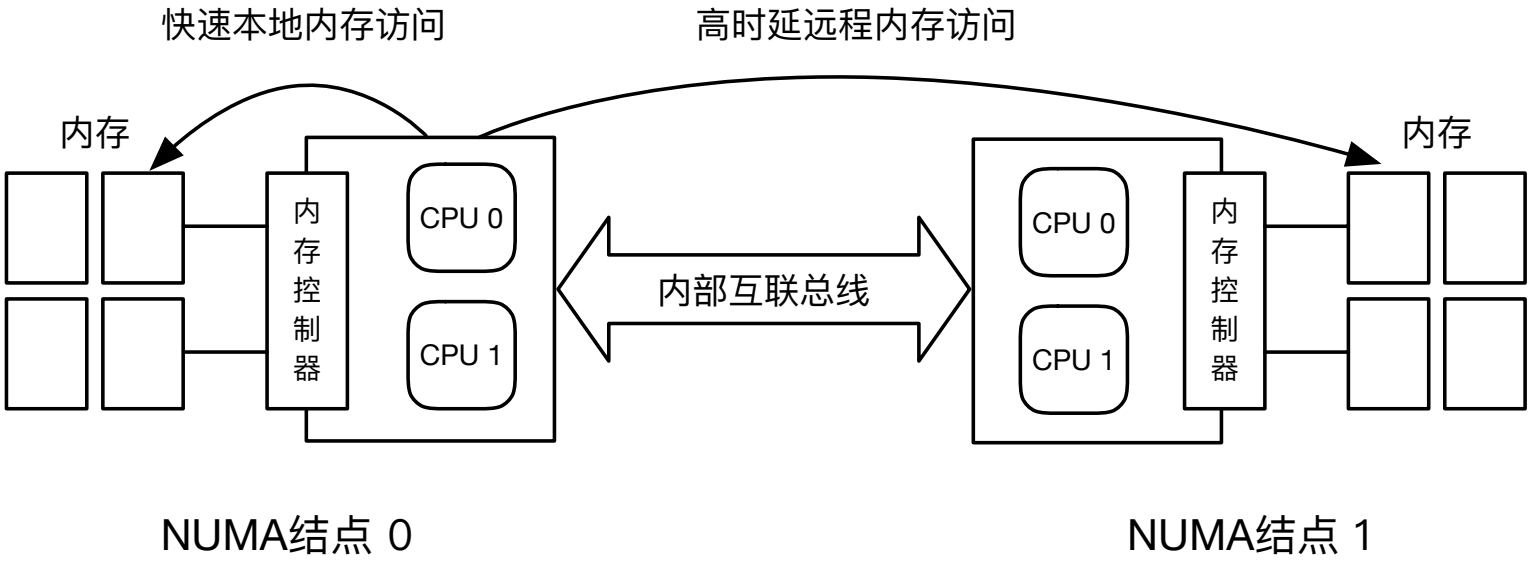
内存条



华为鲲鹏服务器（ARM架构）

常见于多处理器（多插槽）机器

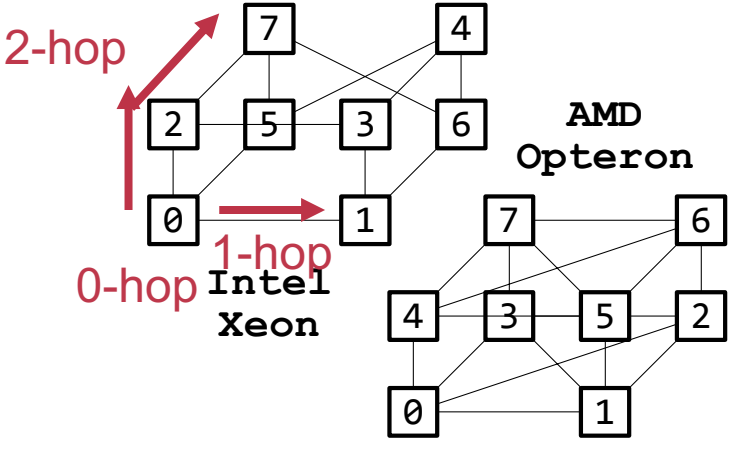
非一致内存访问



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) {
```

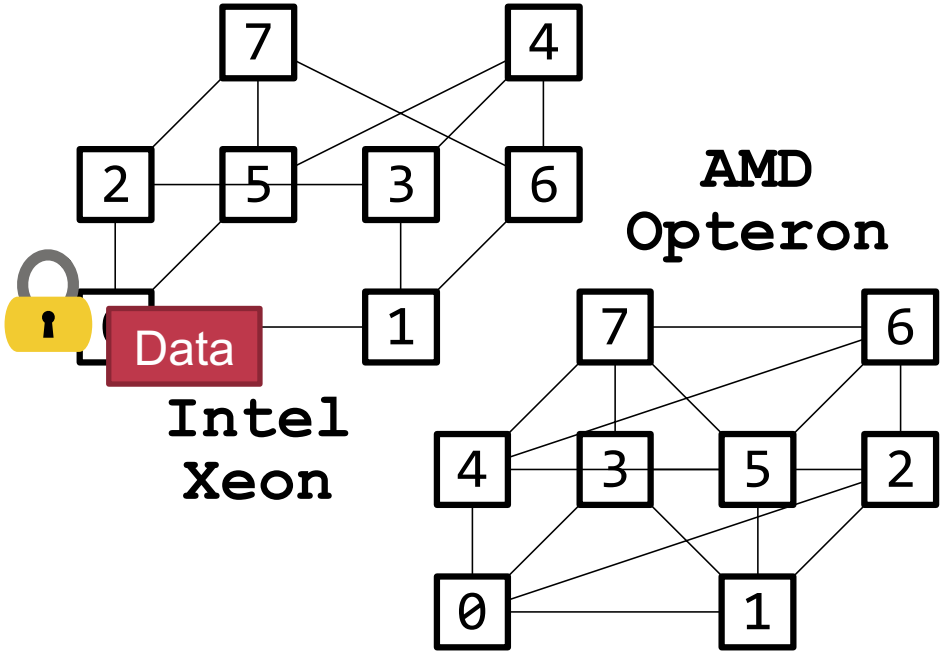
申请进入临界区

临界区部分

通知退出临界区

其他代码

```
}
```



NUMA环境中新的挑战

Challenge: 锁**不知道**临界区中需要访问的内容！

除了锁的元数据，
主要是临界区中
访问的共享数据

```
while(TRUE) {
```

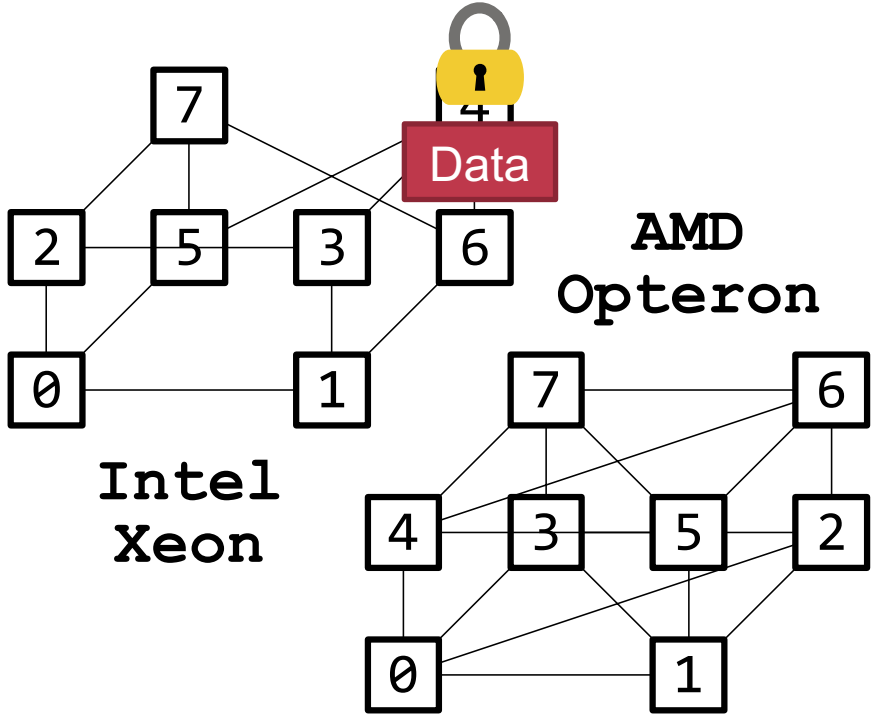
申请进入临界区

临界区部分

通知退出临界区

其他代码

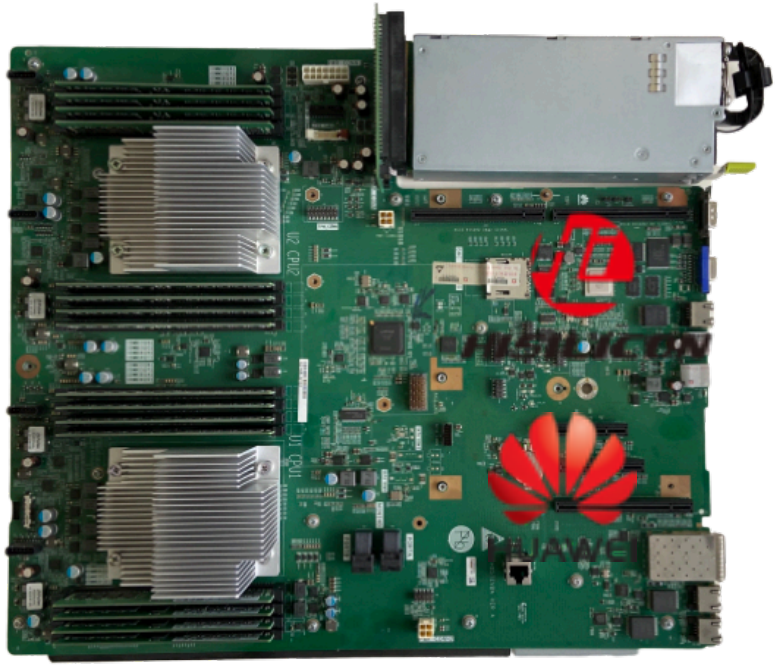
```
}
```



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**

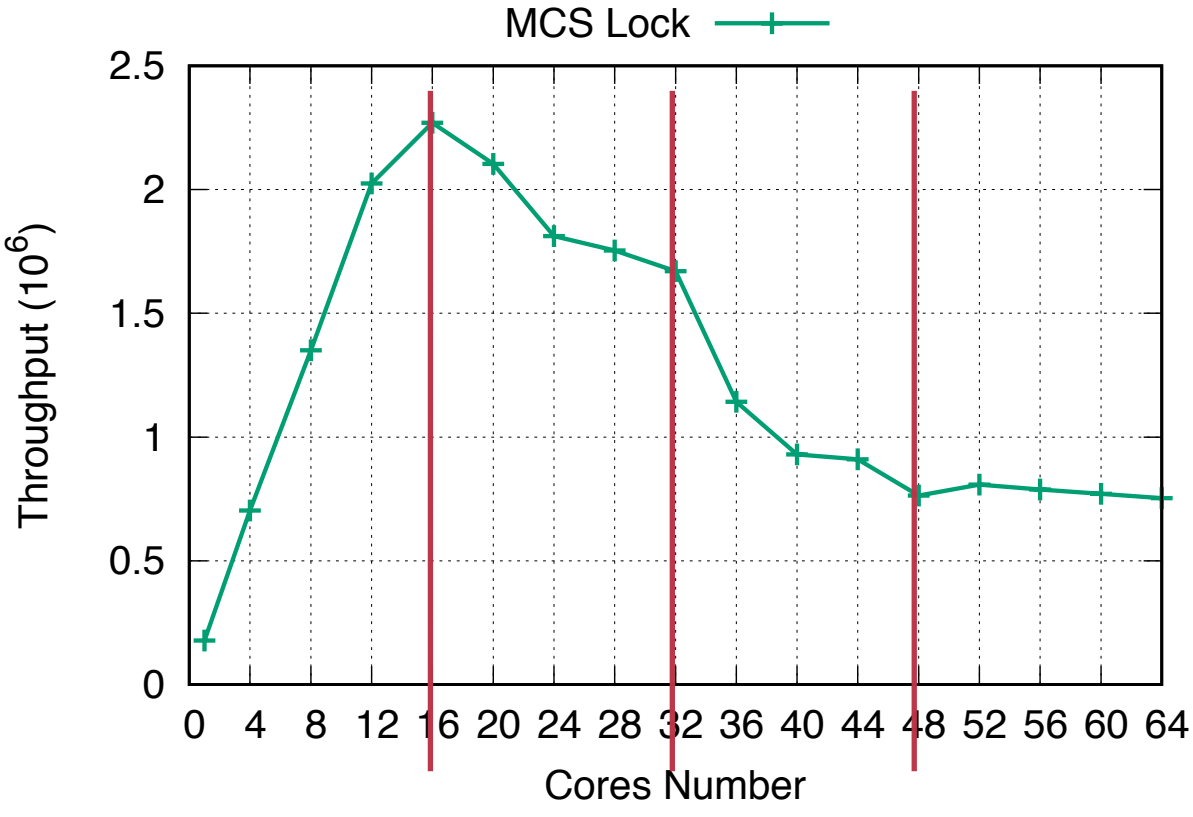
鲲鹏服务器的NUMA结构

```
$ 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？

临界区访问共享**缓存行数量增加到10**



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

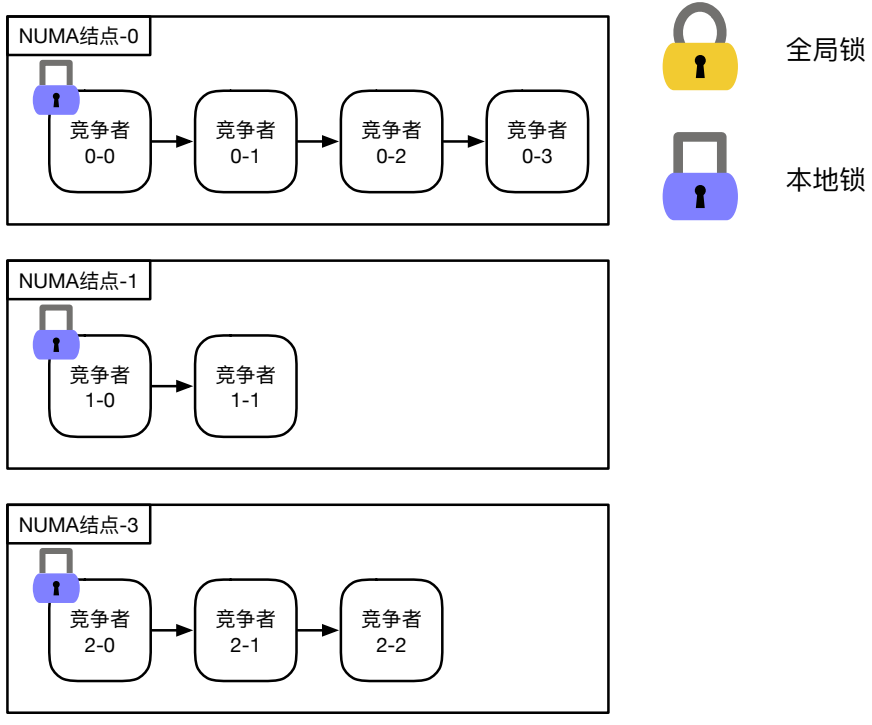
成功获取全局锁

释放时将其传递给

本地等待队列的下一位

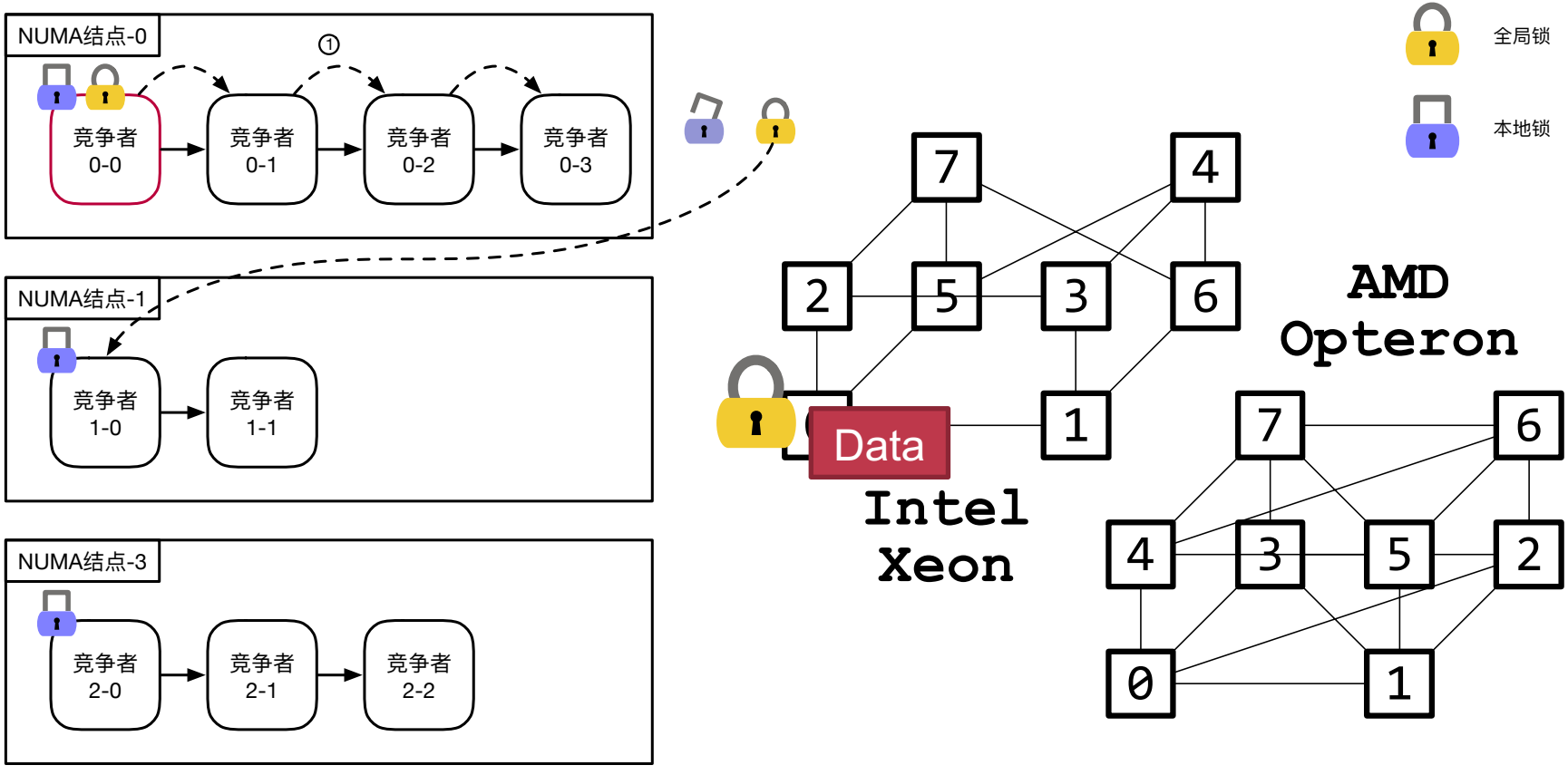
全局锁在一段时间内

只在一个结点内部传递

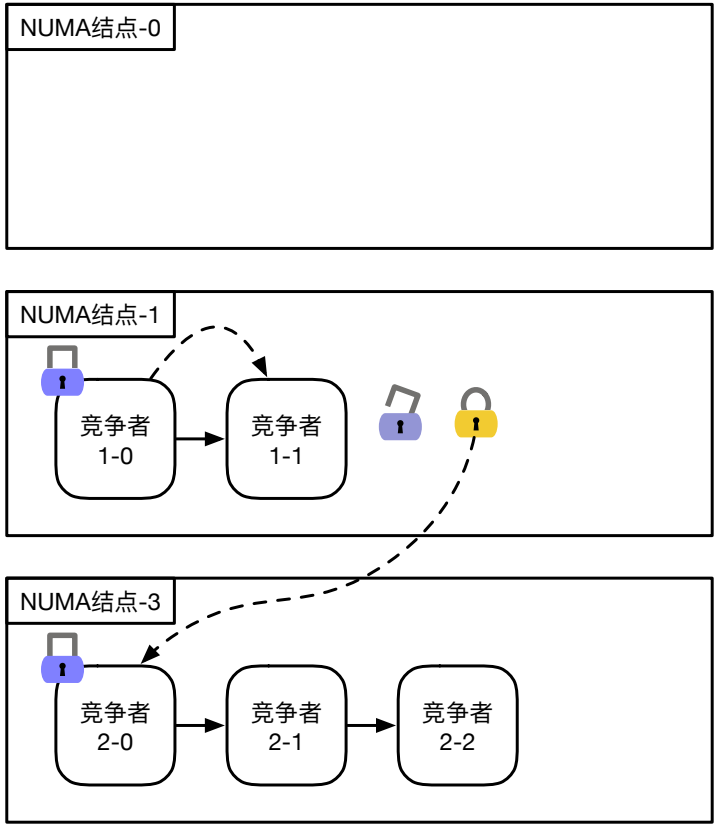


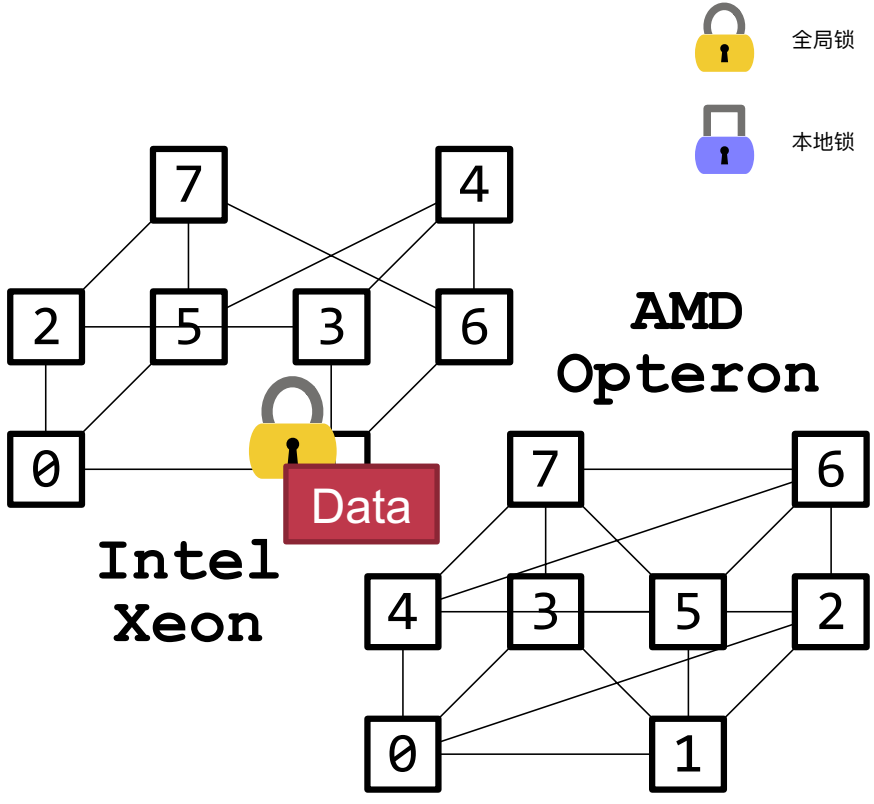
*Dice, David, Virendra J. Marathe, and Nir Shavit. "Lock cohorting: a general technique for designing NUMA locks."

NUMA-aware设计：以cohort锁*为例

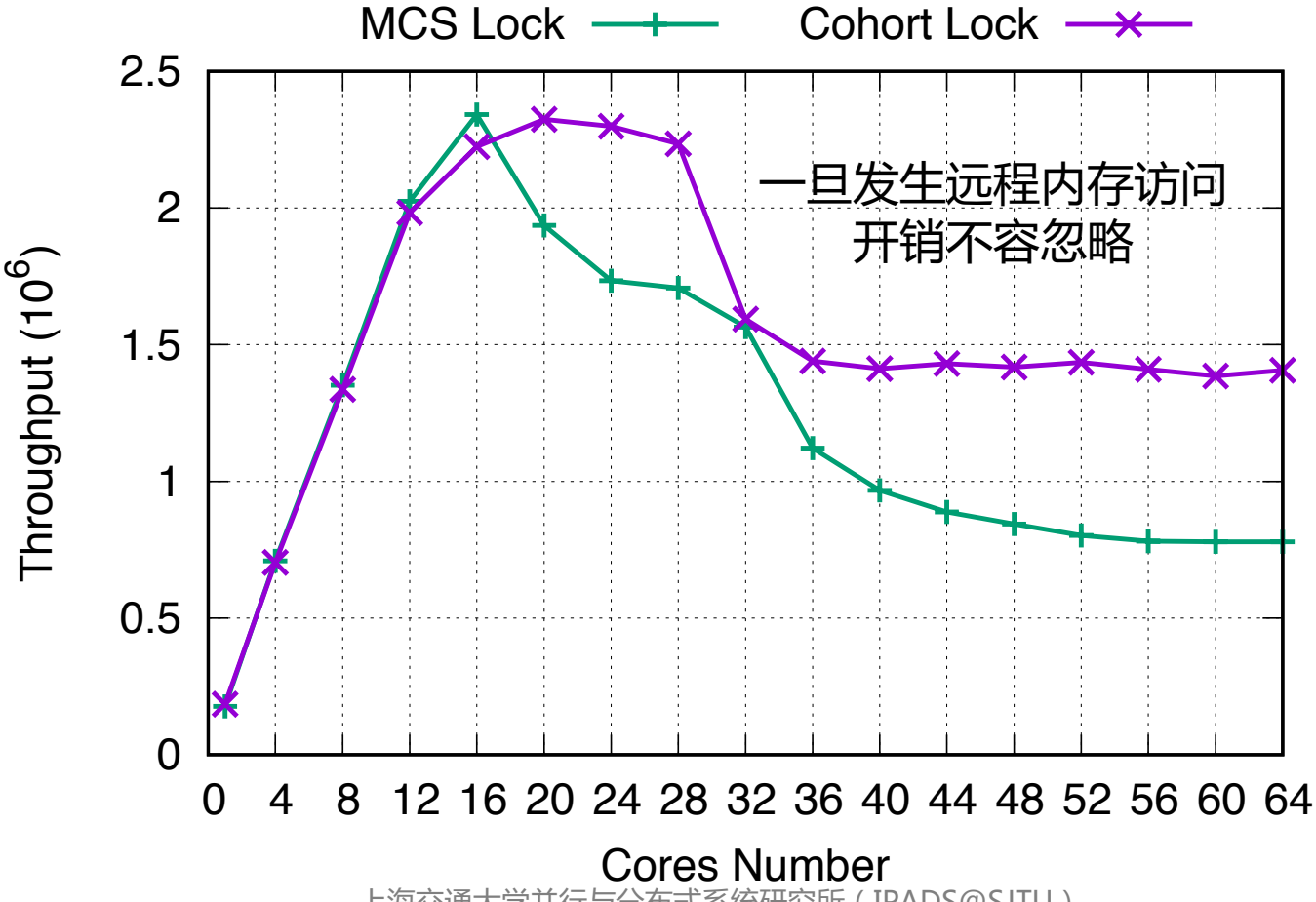


NUMA-aware设计：以cohort锁*为例





NUMA-aware设计：以cohort锁*为例



系统软件开发者视角下的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？

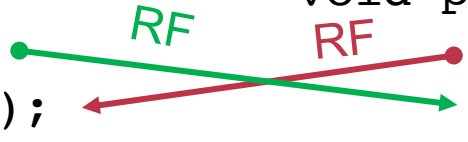
LockOne在现实硬件中能够保证互斥访问吗？

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]);
```



RF: 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]);
```


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

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

```
    void proc_A(void) {          void proc_B(void) {
T1          while(flag[1]);
T2                                          while(flag[0]);
T3          flag[0] = 1;
T4                                          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

proc_A 与 proc_B 都读到对方flag为0，**同时进入临界区**

```
    void proc_A(void) {          void proc_B(void) {
T1          while(flag[1]);
-------------------------------------------------------
T2                                        while(flag[0]);
-------------------------------------------------------
T3          flag[0] = 1;
-------------------------------------------------------
T4                                        flag[1] = 1;
```

LockOne在现实硬件中能够保证互斥访问吗？

会发生这个现象

ARM

x86/64

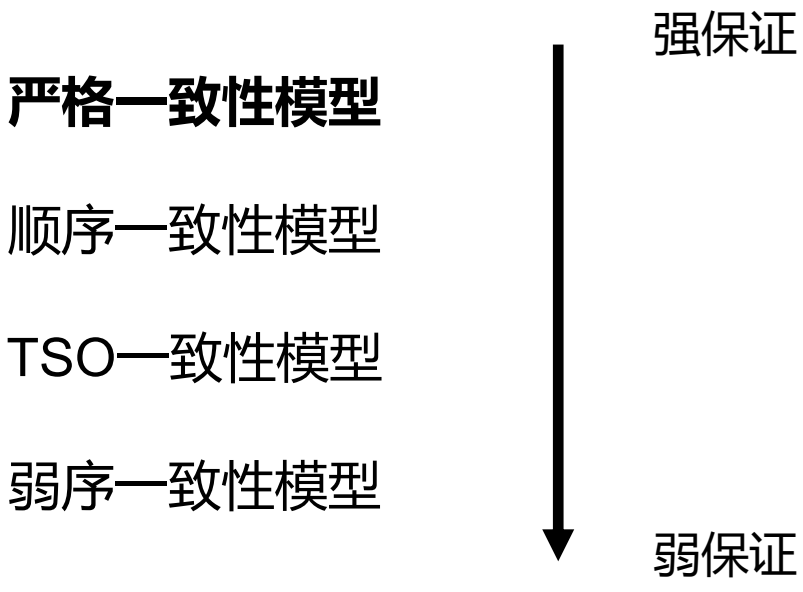
PowerPC

SPARC TSO

不会发生这个现象

dual 386 (circa 1989)

几种内存模型



内存模型：严格一致性模型

- 严格一致性模型（Strict Consistency）

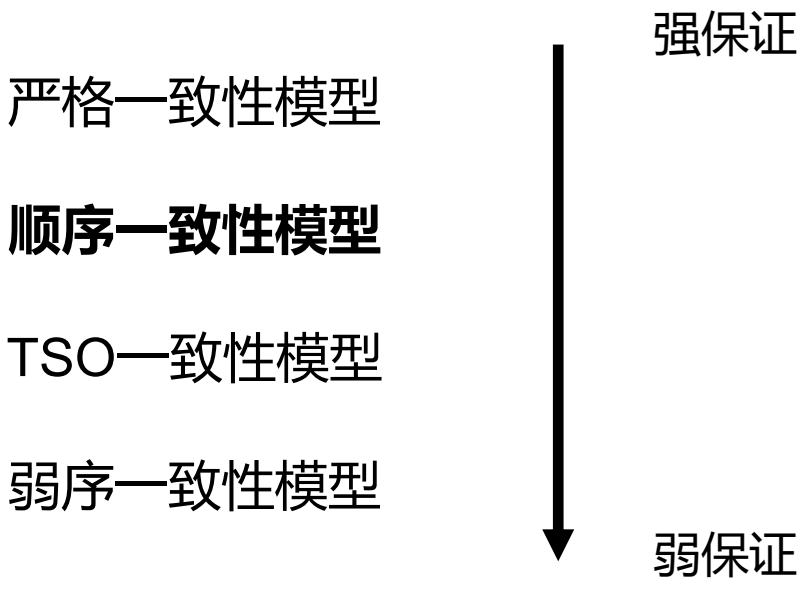
  对一个地址的任意的**读操作**都能读到这个地址**最近一次写**的数据

  访存操作顺序与**全局时钟**的顺序完全一致

```
      void proc_A(void) {          void proc_B(void) {
T1          flag[0] = 1;
T2                                          flag[1] = 1;
T3          A = flag[1];
T4    }                                     B = flag[0];
                                          }
```

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）

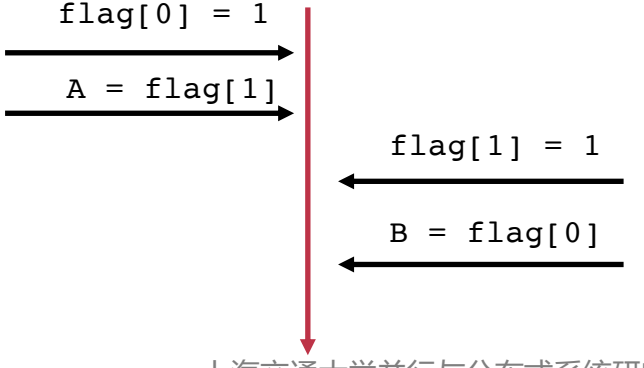
  不要求操作按照真实发生的时间顺序（全局时钟）全局可见

  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：

全局顺序



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）

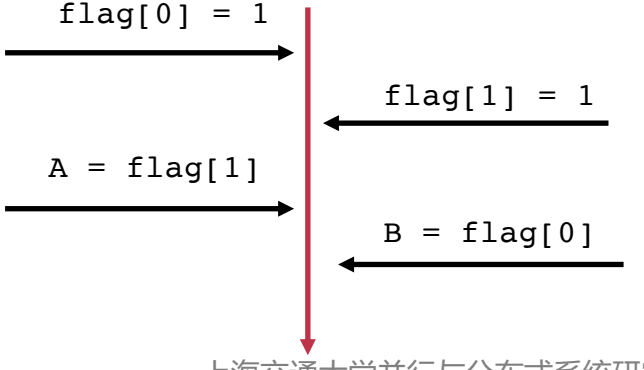
  不要求操作按照真实发生的时间顺序（全局时钟）全局可见

  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：

全局顺序



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）

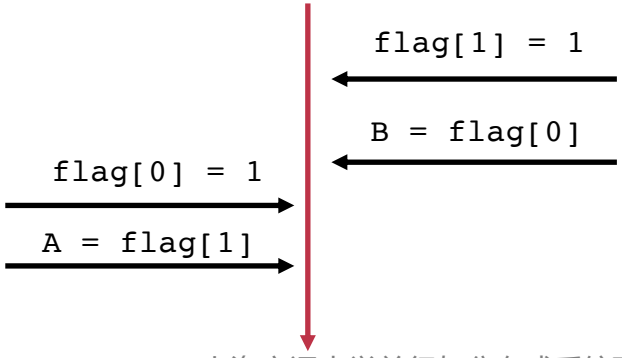
  不要求操作按照真实发生的时间顺序（全局时钟）全局可见

  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：

全局顺序



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

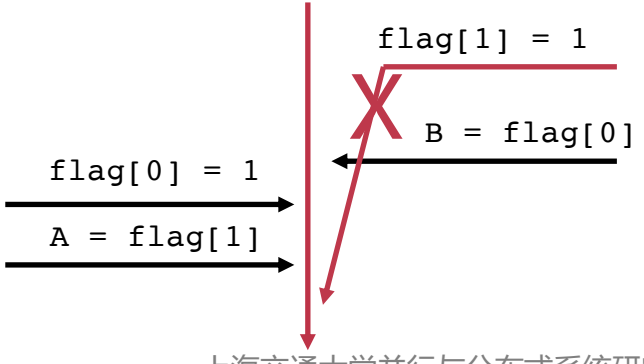
- 顺序一致性模型（Sequential Consistency）

  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：



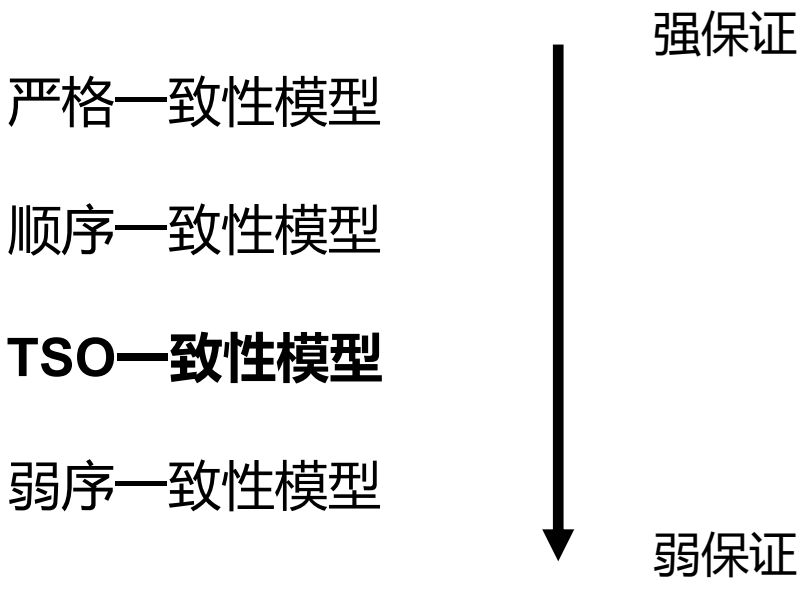
全局序列中，proc_B的程序顺序被打破了

```
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

```
     void proc_A(void) {                    void proc_B(void) {
T1         flag[0] = 1;  写
T2                                                  flag[1] = 1; 写
T3         A = flag[1];  读
T4   }                                              B = flag[0]; 读
                                                    }
```

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

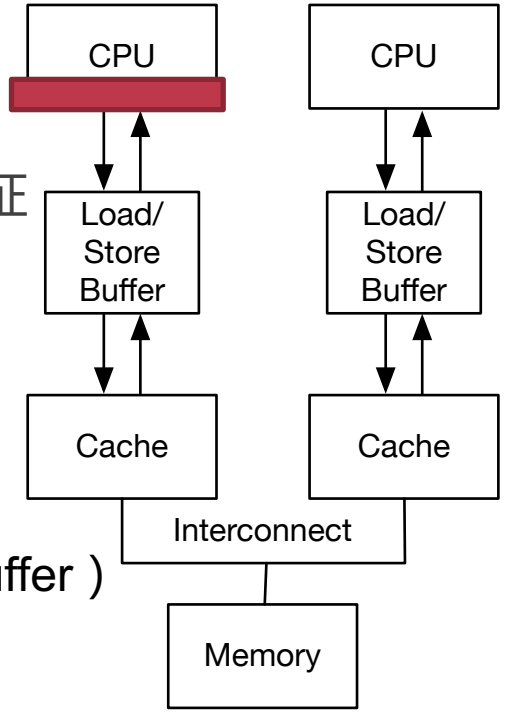
  只有**写-读**的顺序**不能**够得到保证

初始值：`flag[0] = 0 flag[1] = 0`

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

写操作**需要等待缓存一致性**



- 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

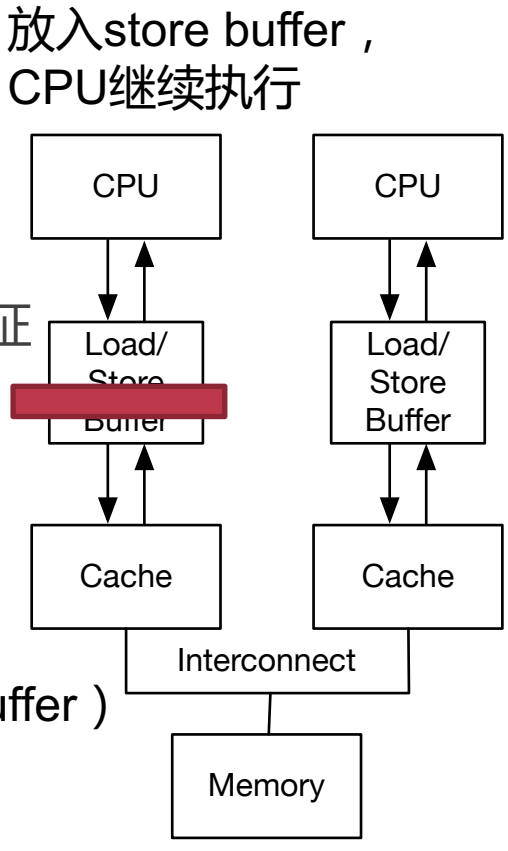
内存模型：TSO一致性模型

- 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

内存模型：TSO一致性模型

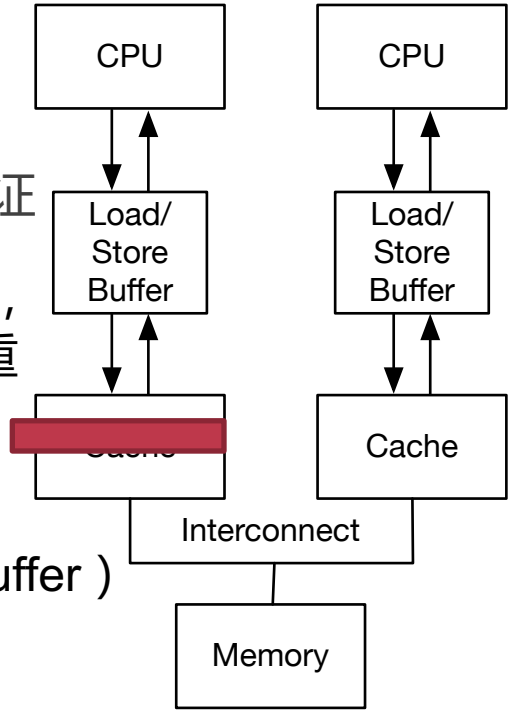
- TSO 一致性模型（Total Store Ordering）

  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

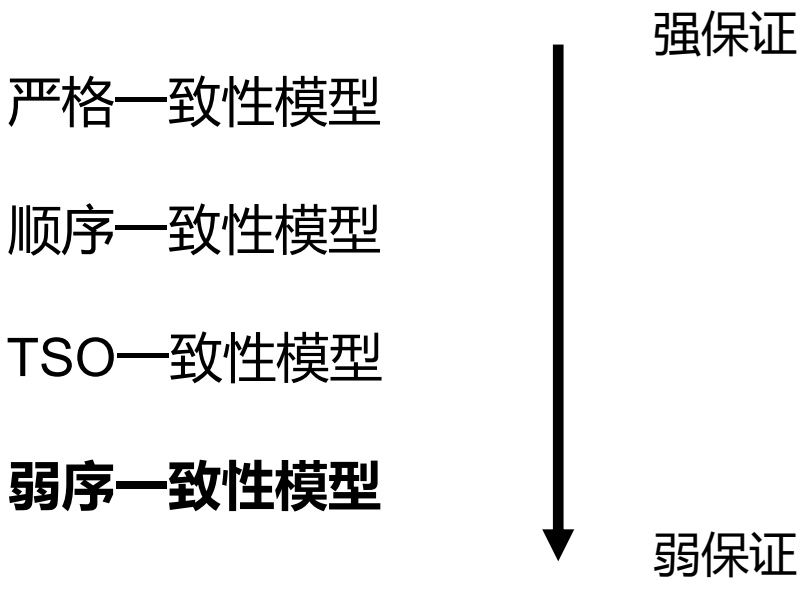
缓存一致性满足，
真实写到cache重

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

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;

        data = 666;
}
```

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

时刻T1，proc_B读到错误的值

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

移动场景，成本控制

主要考虑CPU复杂度与功耗表现

“高性能”

CPU更复杂，能效更高

*2012年之前

Credit: https://preshing.com/20120930/weak-vs-strong-memory-models/

不同架构使用不同的内存模型

ARM维持**弱内存模型**

但是也在向高性能高

能效服务器发展

基本也没了

主机阵营去x86

（除switch用的ARM）

Apple全换intel

***2020**年

Credit: https://preshing.com/20120930/weak-vs-strong-memory-models/

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

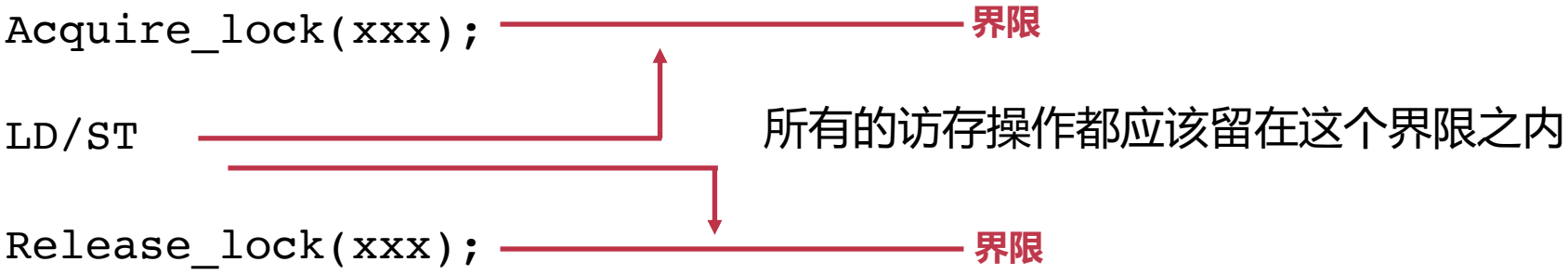
```
void proc_A(void) {
      data = 666;
      barrier();
      flag = READY;
}
```

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

系统软件开发者视角下的内存模型

- 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**



临界区内的访存操作不会在释放锁之后**可见**

系统软件开发者视角下的内存模型

- 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