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

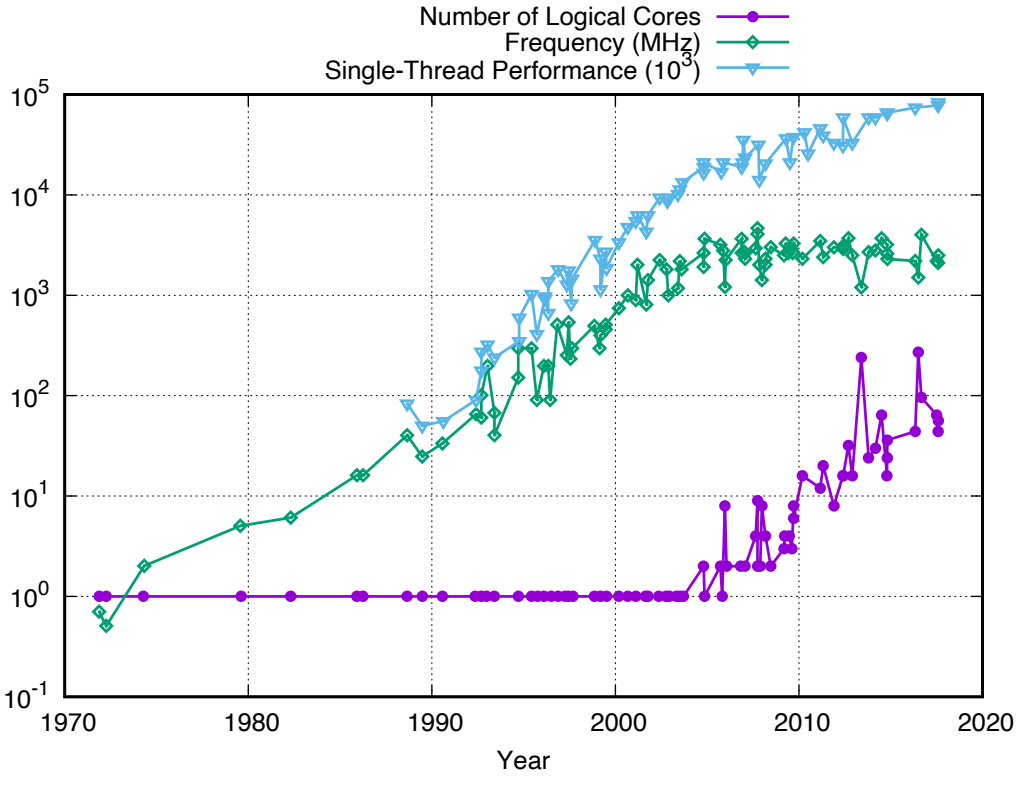
https://ipads.se.sjtu.edu.cn

版权声明

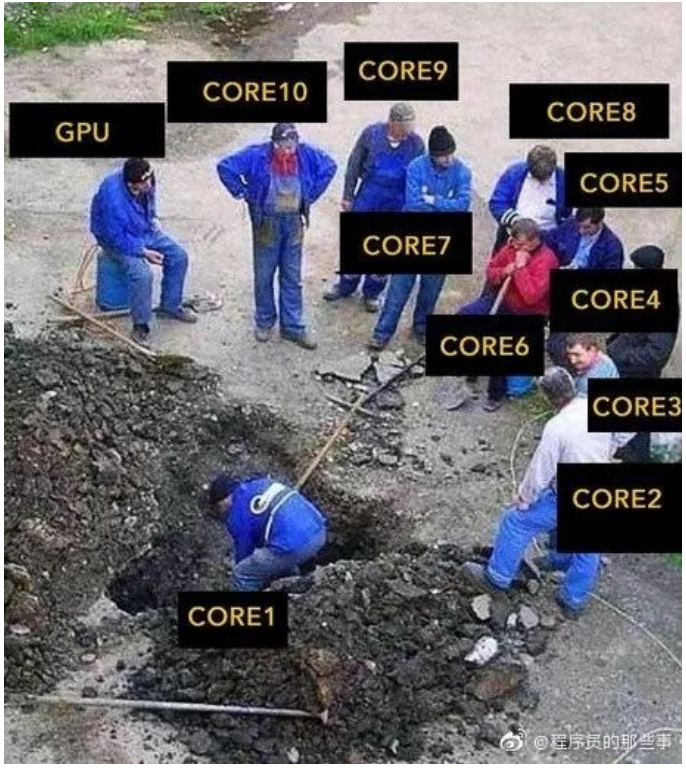
- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

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

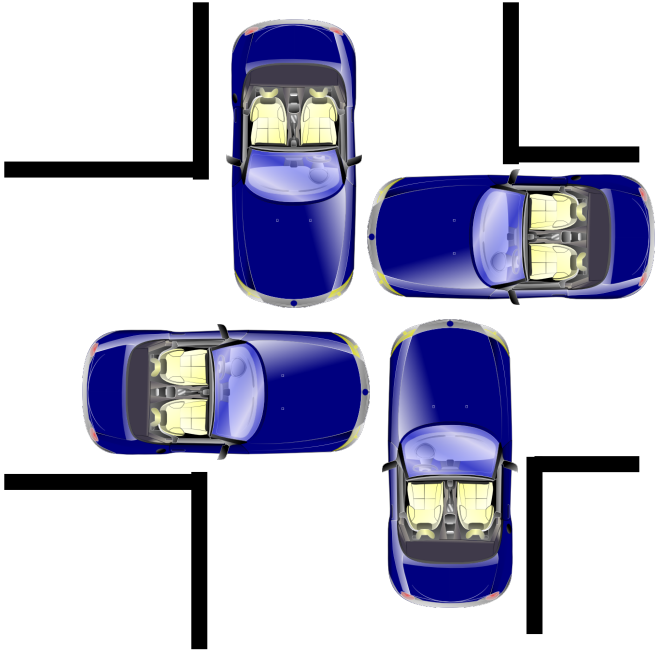
工头找了10万人，需要多久？

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）

同步带来的问题：死锁

死锁



十字路口的“困境”

```
void proc_A(void) {
        lock(A);
        /* Time T1 */
        lock(B);
        /* Critical Section */
        unlock(B);
        unlock(A);
}

void proc_B(void) {
        lock(B);
        /* Time T1 */
        lock(A);
        /* Critical Section */
        unlock(A);
        unlock(B);
}
```

T1时刻的死锁

死锁产生的原因

- 互斥访问
- 持有并等待
- 资源非抢占
- 循环等待

```
void proc_A(void) {
        lock(A);
        /* Time T1 */
        lock(B);
        /* Critical Section */
        unlock(B);
        unlock(A);
}

void proc_B(void) {
        lock(B);
        /* Time T1 */
        lock(A);
        /* Critical Section */
        unlock(A);
        unlock(B);
}
```

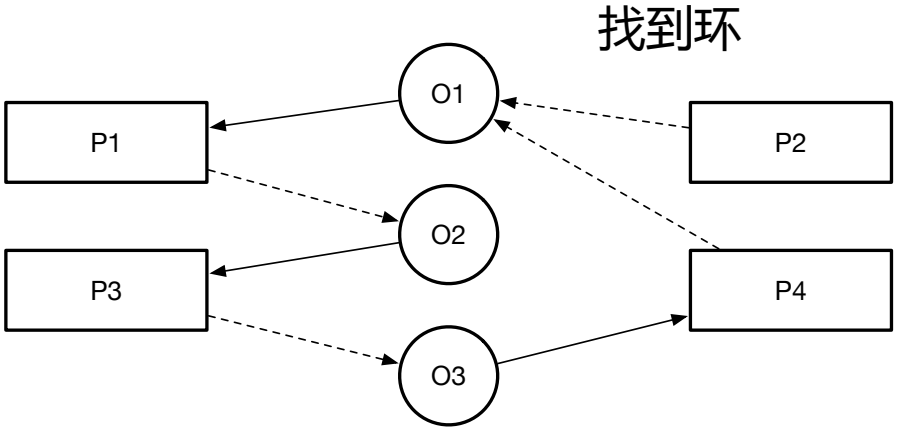
T1时刻的死锁

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

检测死锁与恢复



资源分配图

资源分配表

进程号	资源号
P1	O1
P3	O2
P4	O3

进程等待表

进程号	资源号
P1	O2
P2	O1
P3	O3

如何恢复？打破循环等待！

- 直接kill所有循环中的进程
- Kill一个，看有没有环，有的话继续kill
- 全部回滚到之前的某一状态

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

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）

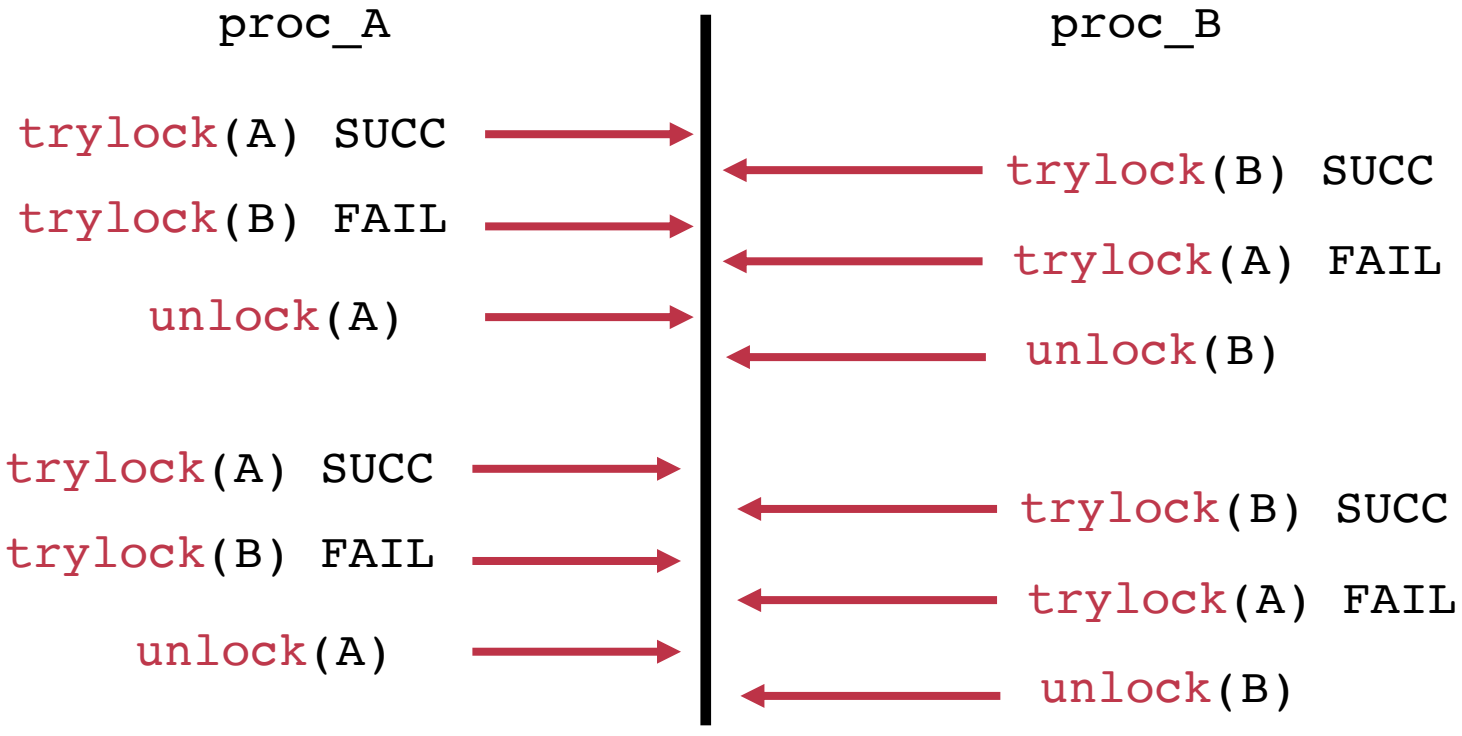
2. 不允许持有并等待：一次性申请所有资源

trylock非阻塞
立即返回成功或失败

```
while (true) {
        if(trylock(A) == SUCC)
                if(trylock(B) == SUCC) {
                        /* Critical Section */
                        unlock(B);
                        unlock(A);
                        break;
                } else
                        unlock(A);
}
```

无法获取B，那么释放A

避免死锁带来的活锁 Live Lock



如此往复...　死锁是**无法恢复**的，但是活锁**可能自己恢复**

另一种避免持有并等待：乐观并发控制

我们能否避免等待？（之前是避免持有时等待）

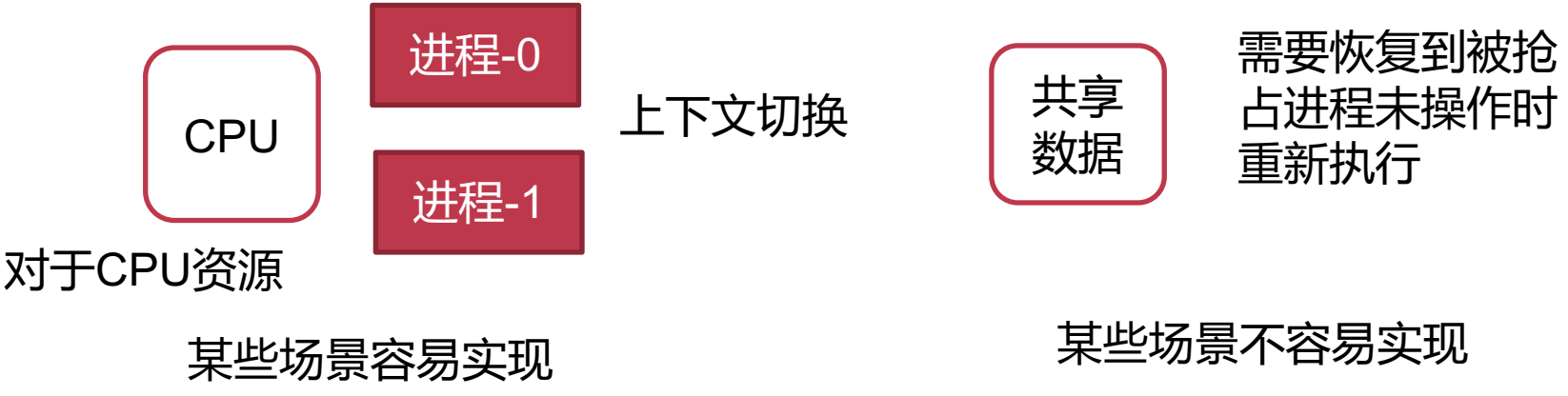
1. 读取：将数据读入缓存，获得一个**时间戳**。
2. 校验：执行完毕后，进行**提交**。这时校验所有其他进程，如果**读取的数据在读取之后又被其他事务修改**，则产生冲突，**中断回滚**。
3. 写入：通过校验阶段后，完成数据更新。

与之前介绍的LL/SC类似

思考：这也会出现Live Lock吗？

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



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
   - 所有资源进行编号
   - 所有进程递增获取

   任意时刻：获取最大资源号的进程可以继续执行，然后释放资源

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**

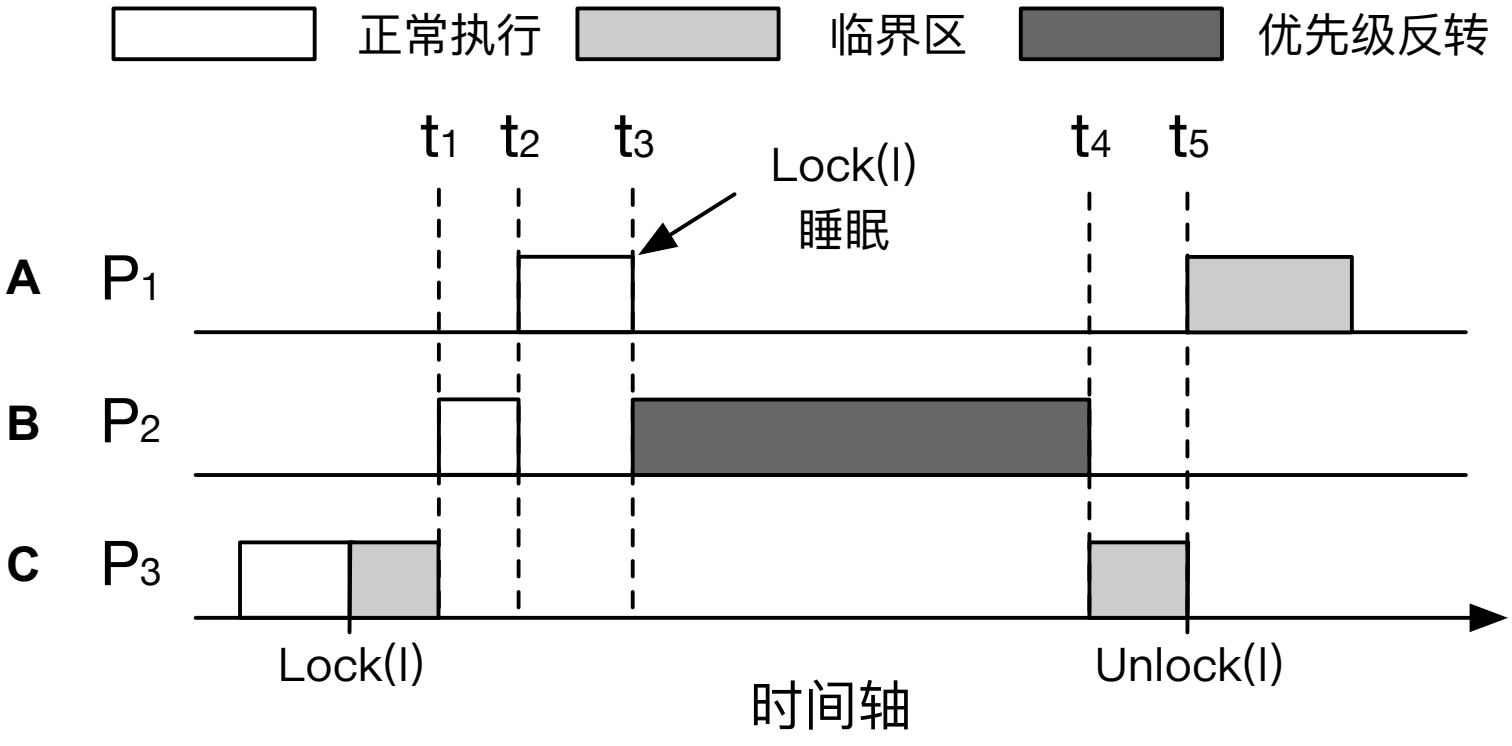
死锁避免：运行时检查是否会出现死锁

银行家算法

- 所有进程获取资源需要通过**管理者**同意
- 管理者**预演**会不会造成死锁
  - 如果会造成：阻塞进程，下次再给
  - 如果不会造成：给进程该资源

同步带来的问题：优先级反转

回顾：优先级反转


正常执行
临界区
优先级反转
t1
t2
t3
t4
t5
Lock(I)
睡眠
A
P1
B
P2
C
P3
Lock(I)
Unlock(I)
时间轴

优先级反转解决方案

思考：为什么会出现优先级反转？

操作系统：基于优先级调度

锁：对于竞争同一个资源的进程按照锁使用的策略进行“调度”

双重调度导致

如何解决？打通两重调度，给另一个调度hint

优先级反转解决方案

思考：为什么会出现优先级反转？

操作系统：基于优先级调度　　**根本原因：双重调度不协调**

锁：对于竞争同一个资源的进程按照锁使用的策略进行“调度”

如何解决？打通两重调度，给另一个调度hint

- 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

优先级反转解决方案

思考：为什么会出现优先级反转？

操作系统：基于优先级调度　　**根本原因：双重调度不协调**

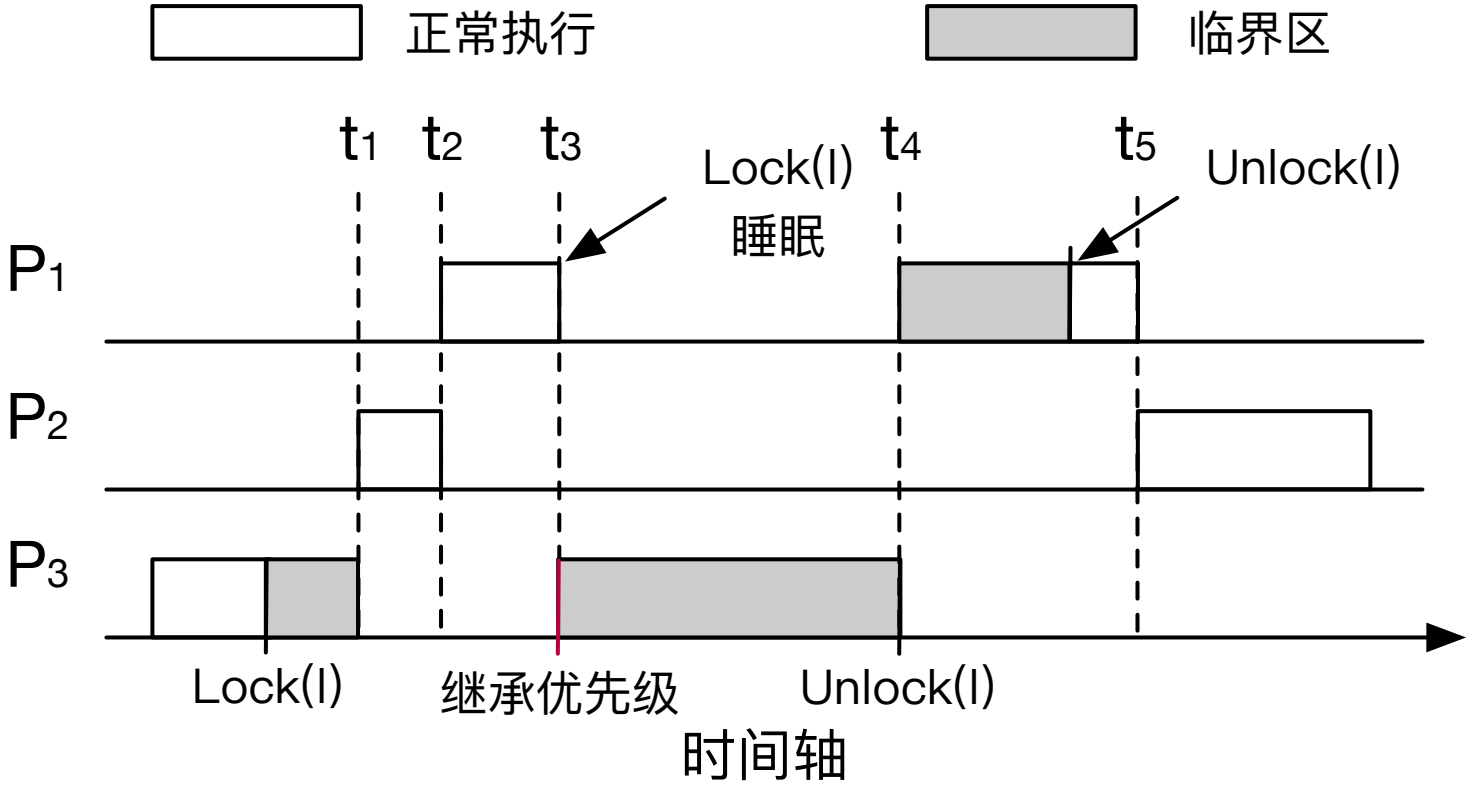
锁：对于竞争同一个资源的进程按照锁使用的策略进行“调度”

如何解决？打通两重调度，给另一个调度hint

- 不可打断临界区协议 (Non-preemptive Critical Sections, NCP)
- 优先级继承协议 (Priority Inheritance Protocol, PIP) （调度章节介绍过）

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

优先级继承协议


正常执行
临界区
t1
t2
t3
Lock(l)
睡眠
t4
t5
Unlock(l)
P1
P2
P3
Lock(l)
继承优先级
Unlock(l)
时间轴

优先级反转解决方案

思考：为什么会出现优先级反转？

操作系统：基于优先级调度　　**根本原因：双重调度不协调**

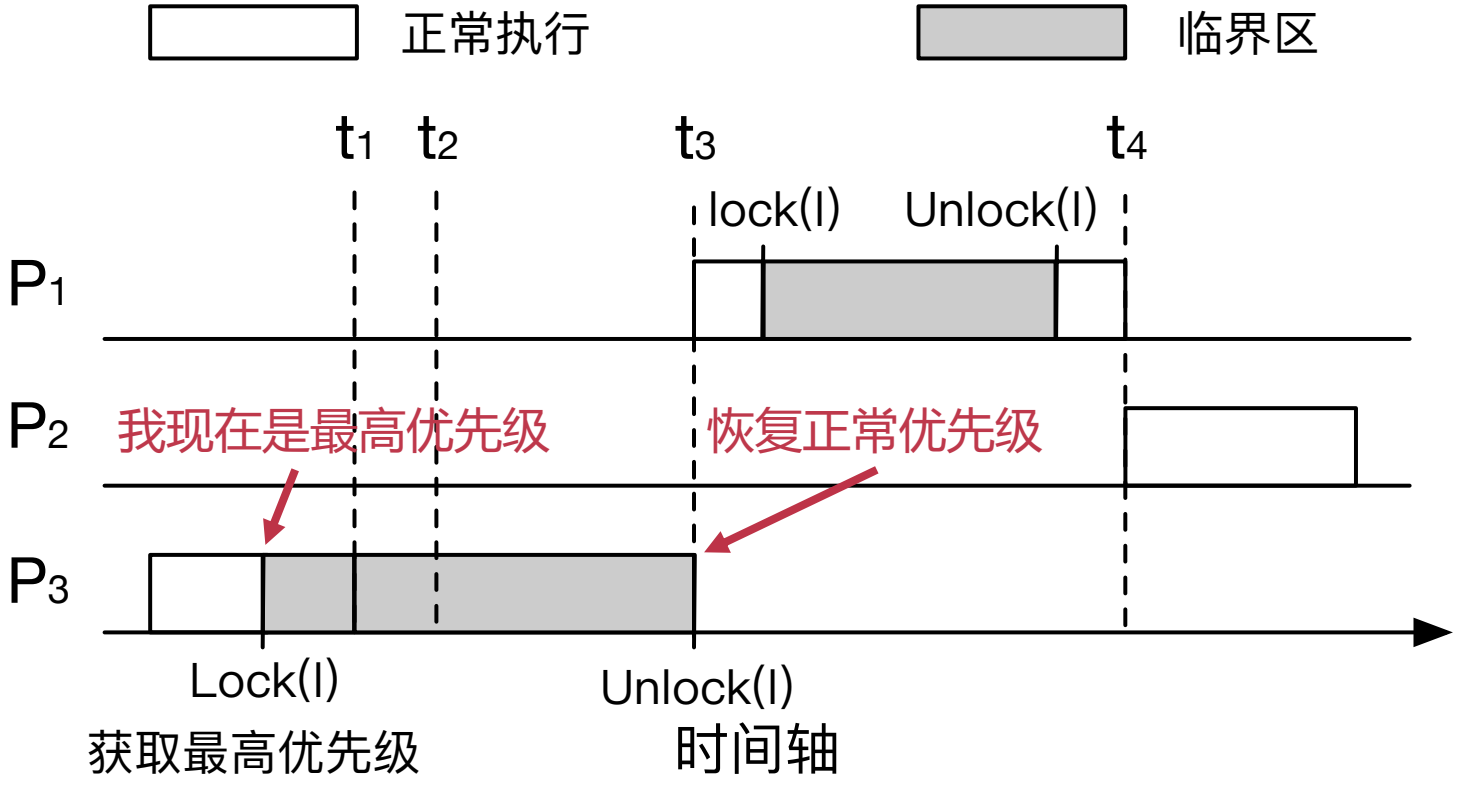
锁：对于竞争同一个资源的进程按照锁使用的策略进行“调度”

如何解决？打通两重调度，给另一个调度hint

- 不可打断临界区协议 (Non-preemptive Critical Sections, NCP)
- 优先级继承协议 (Priority Inheritance Protocol, PIP)
- 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

即时优先级置顶协议


正常执行
临界区
t1
t2
t3
t4
lock(l)
Unlock(l)
P1
P2
P3
我现在是最高优先级
恢复正常优先级
Lock(l)
Unlock(l)
获取最高优先级
时间轴

优先级反转解决方案

思考：为什么会出现优先级反转？

操作系统：基于优先级调度

根本原因：双重调度不协调

锁：对于竞争同一个资源的进程按照锁使用的策略进行“调度”

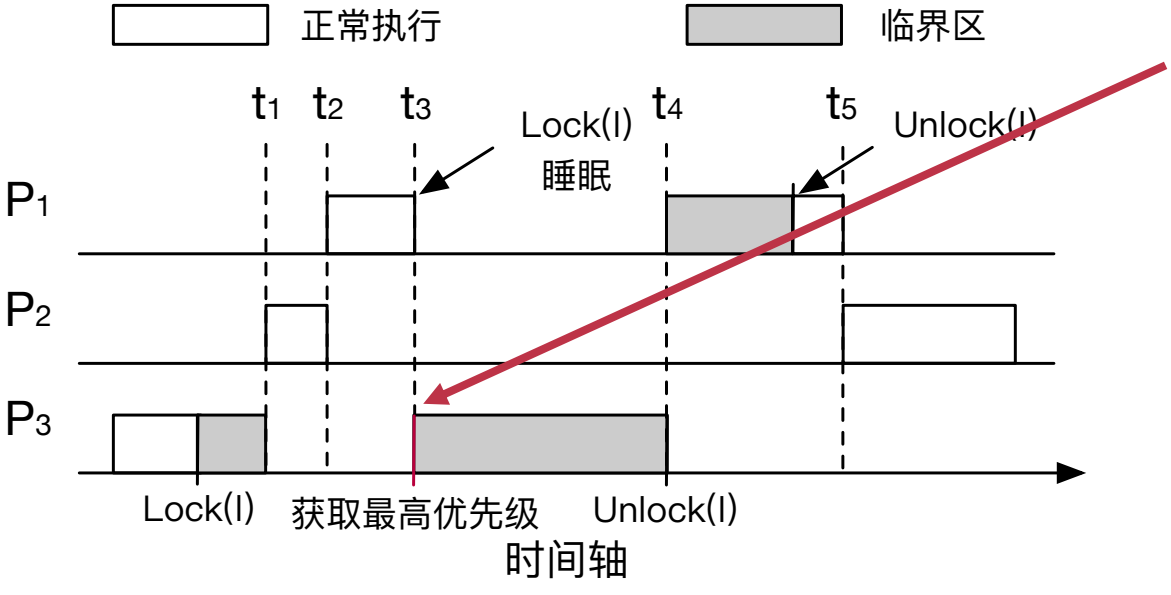
如何解决？打通两重调度，给另一个调度hint

1. 不可打断临界区协议 (Non-preemptive Critical Sections, NCP)
2. 优先级继承协议 (Priority Inheritance Protocol, PIP)
3. 即时优先级置顶协议 (Immediate Priority Ceiling Protocols, IPCP)
4. 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

原生优先级置顶协议

与优先级继承区别：直接给可能获取锁进程中最高优先级，避免未来再被打断，尽快执行完



如果存在更高优先级 P0
且需要获取同一个互斥锁

原生优先级置顶协议：

升为P0优先级

能更快执行完

优先级继承协议：

升为P1优先级

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

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

多核与同步

回顾：操作系统在多处理器多核环境下面临的问题

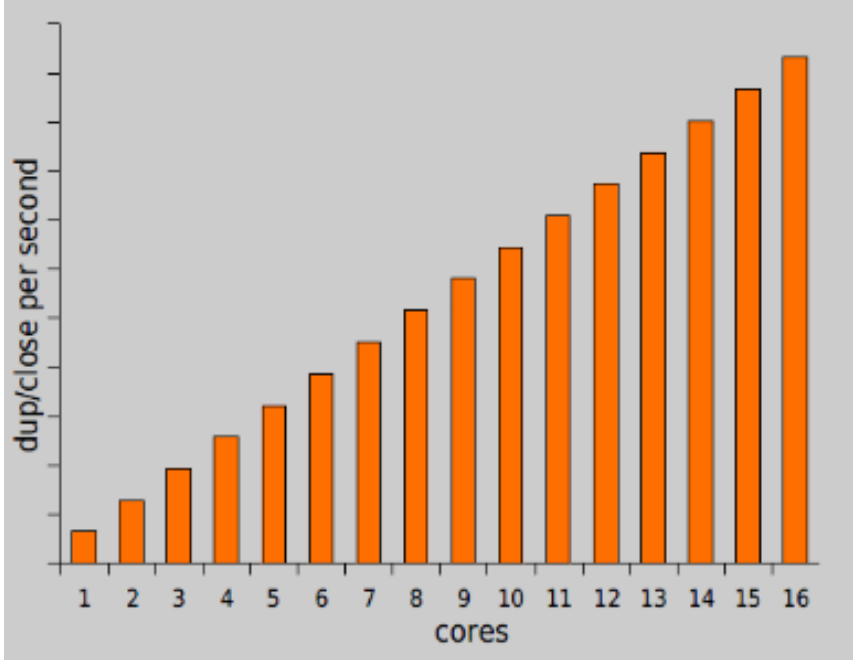
正确性保证

- 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

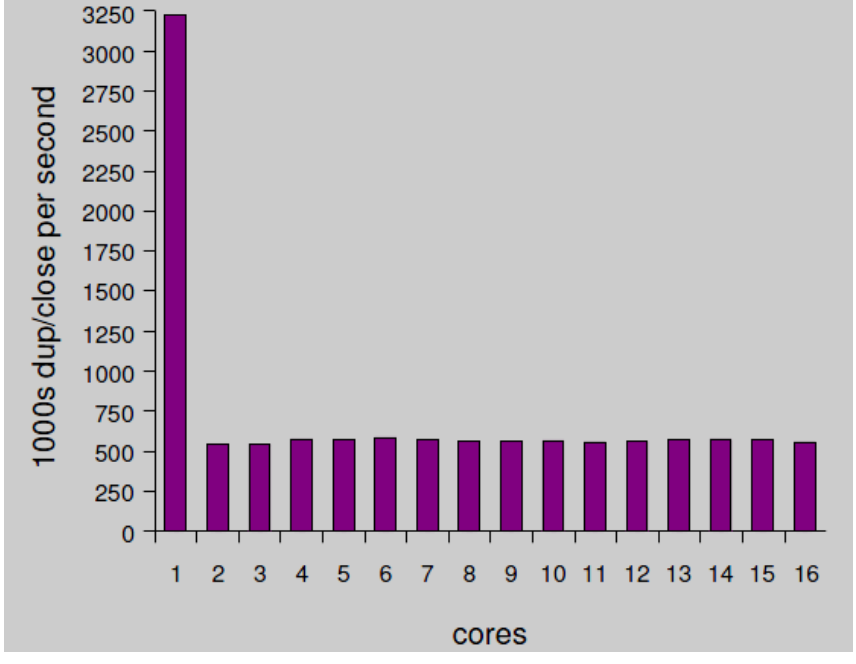
性能保证

- 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

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

加速比

可以并行部分代码占比

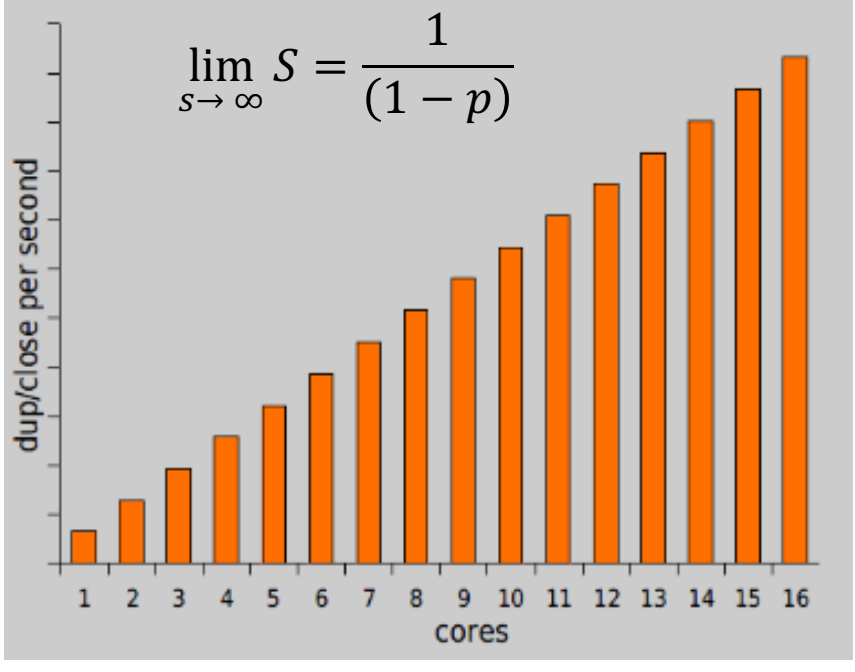
同时执行的核心数

当核心数增加时...

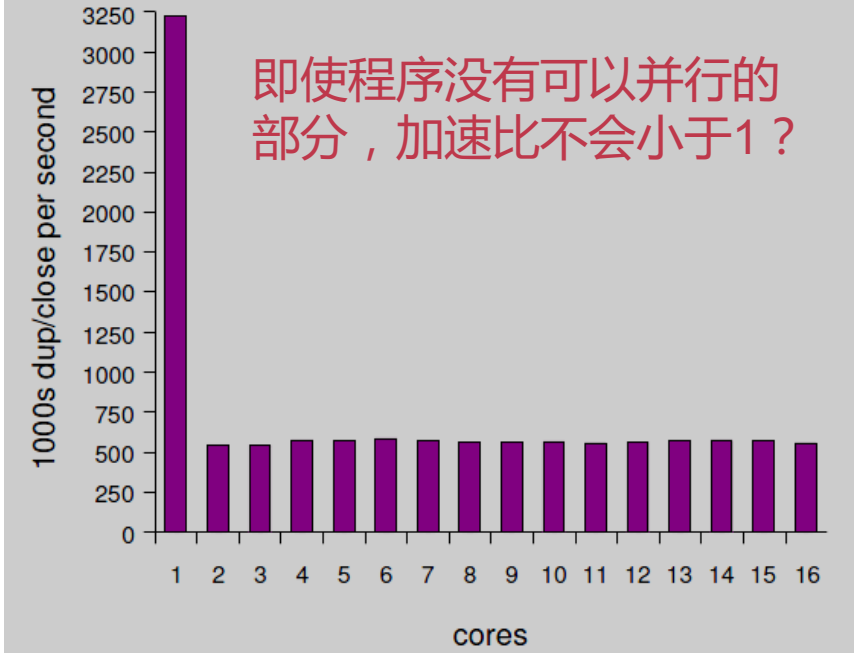
$$\lim_{s \to \infty} S = \frac{1}{(1-p)}$$

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



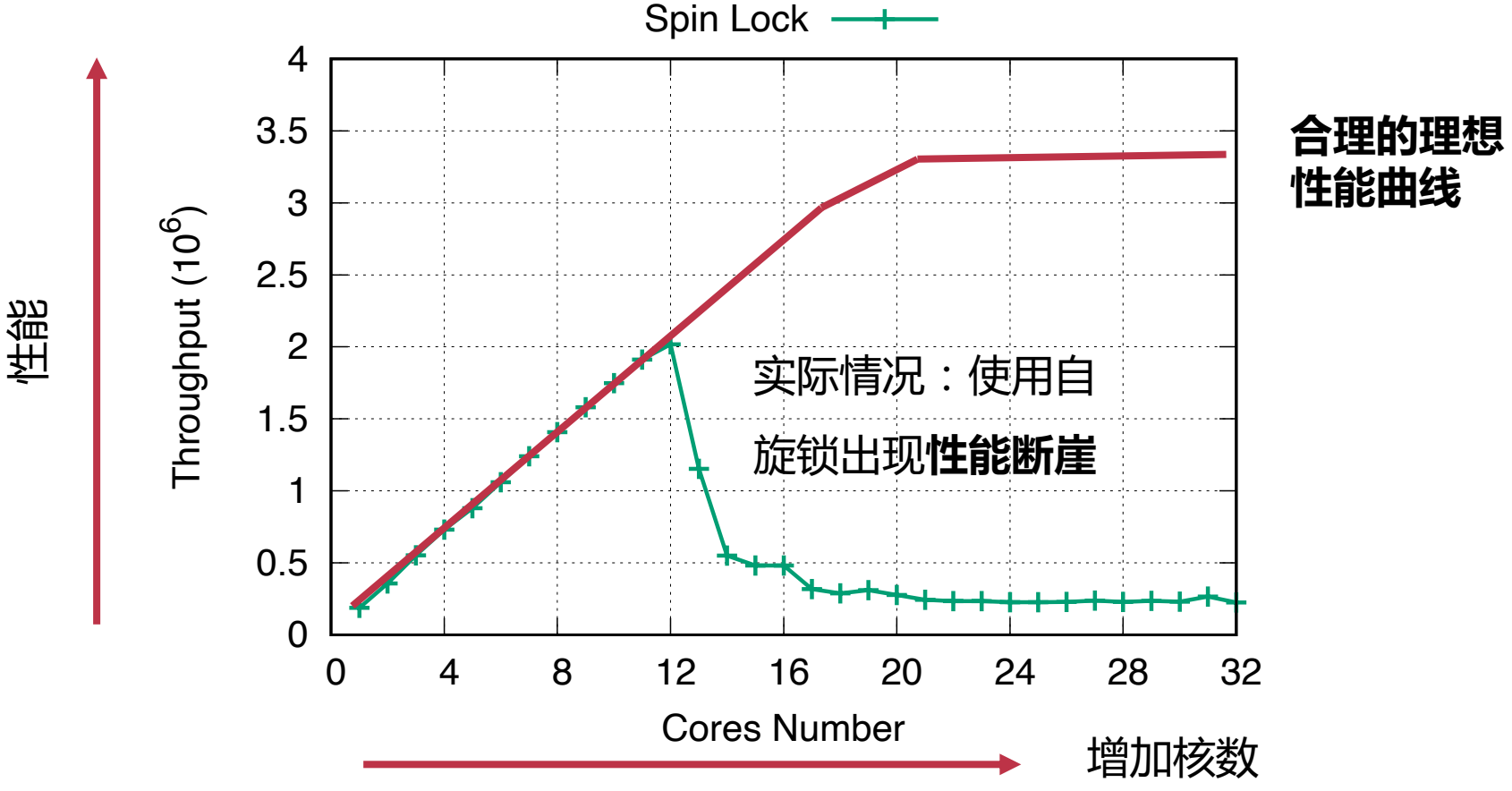
实际fd性能

互斥锁微基准测试

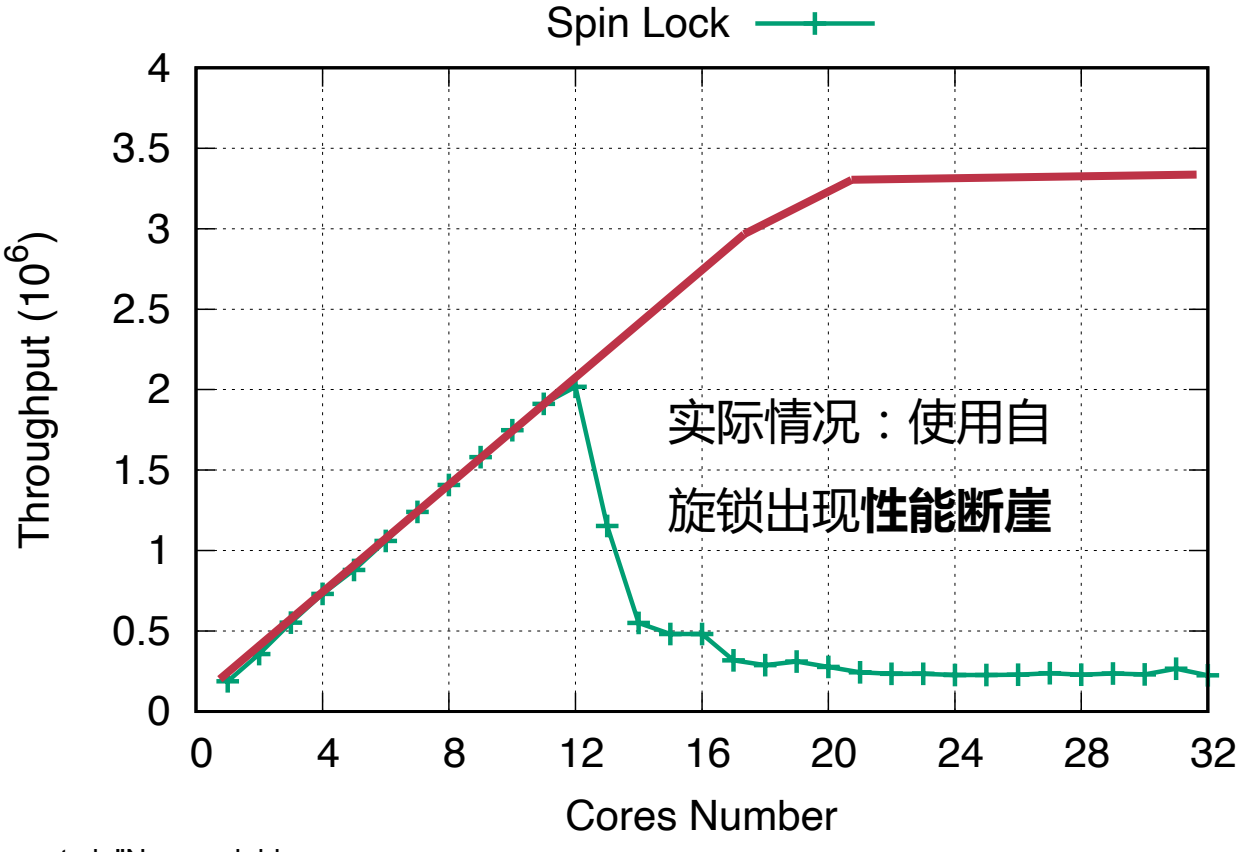
使用微基准测试来复现这个现象

```
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!*



32核服务器上**互斥锁微基准测试**

合理的理想性能曲线

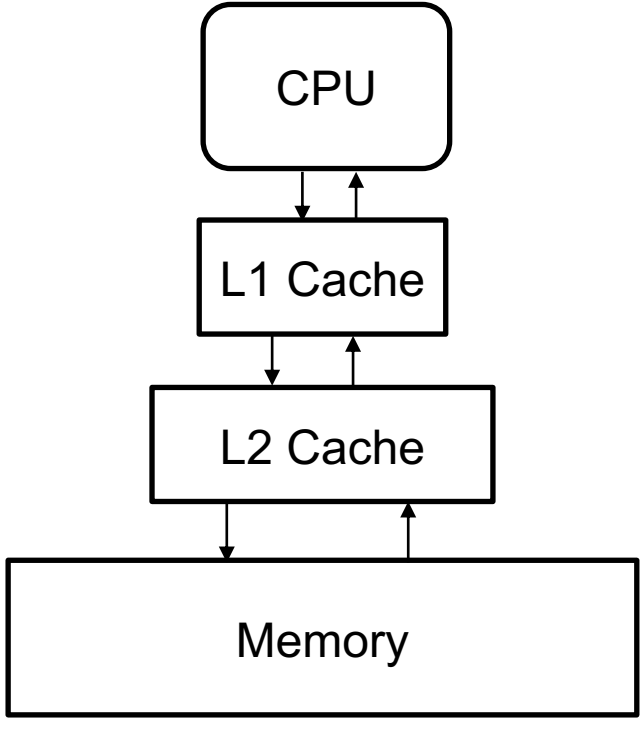
为什么会出现这种情况？

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

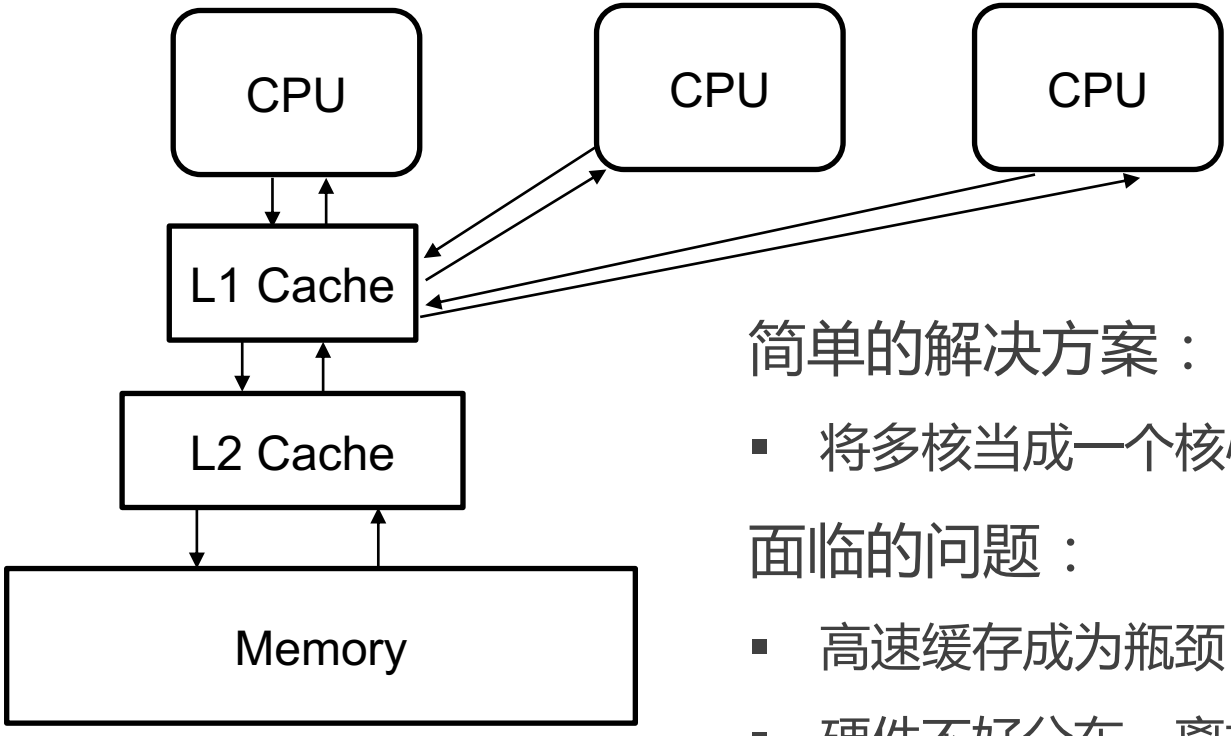
多核环境下的缓存

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：

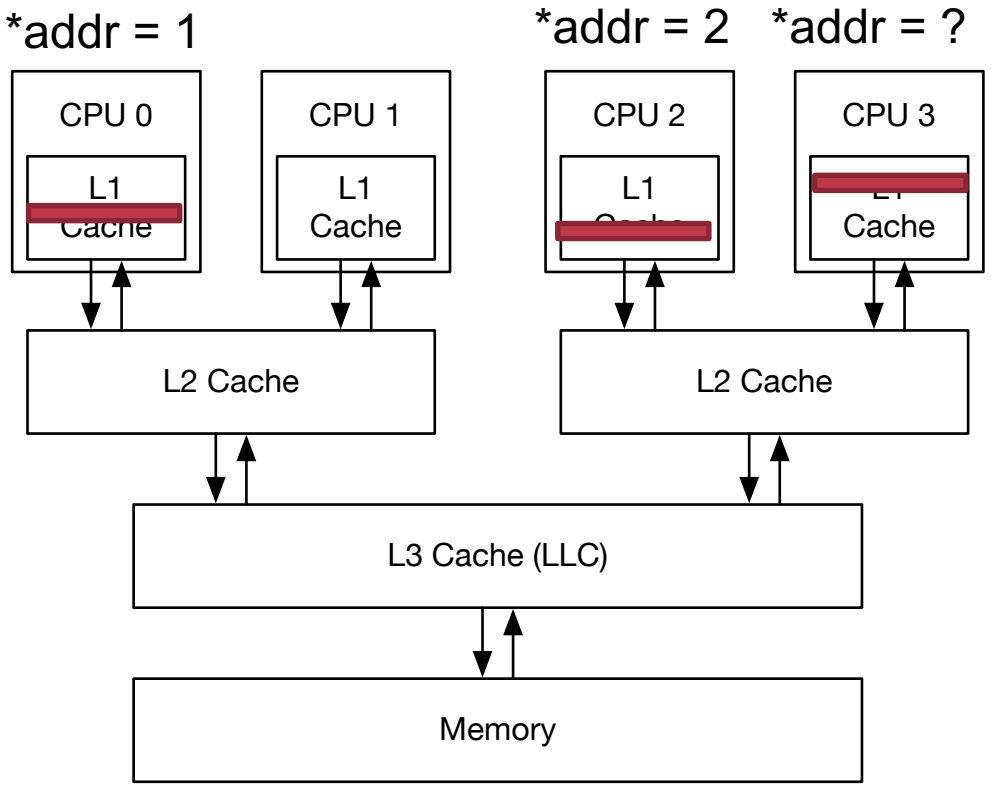
- 将多核当成一个核心，共享缓存设计

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

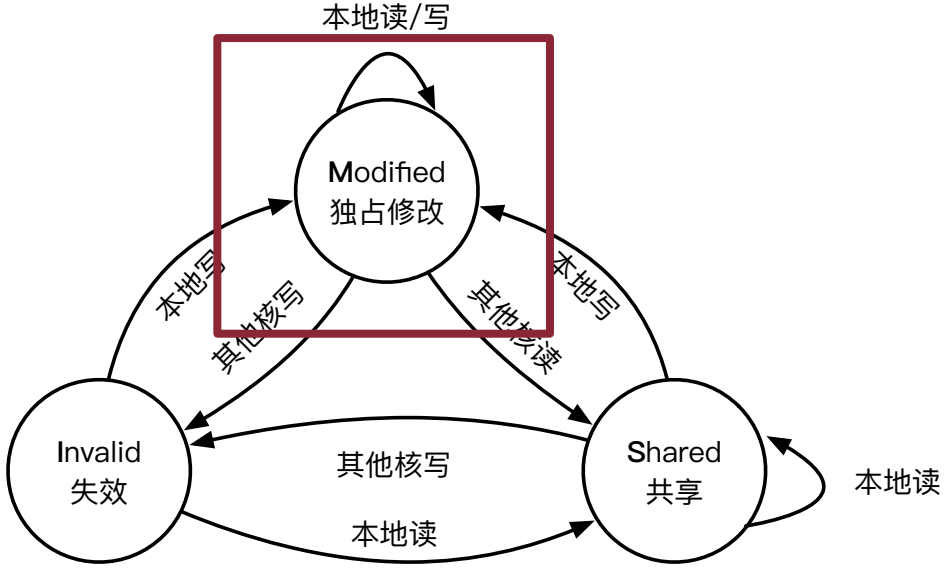
- 保证不同核心对同一地址的值达成共识
- 多种缓存一致性**协议**：窥探式/**目录式缓存一致性**协议

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

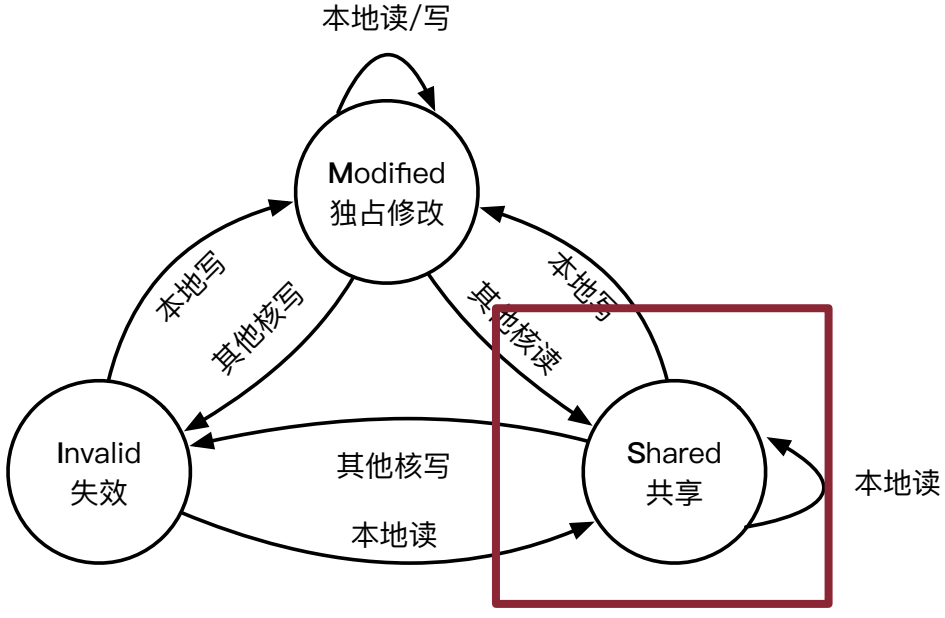
- 独占修改（**M**odified）
  - 该核心独占拥有缓存行
  - 本地可**读**可**写**
  - 其他核**读**需要迁移到**共享**
  - 其他核**写**需要迁移到**失效**



每核心每缓存行状态

缓存一致性：MSI状态迁移

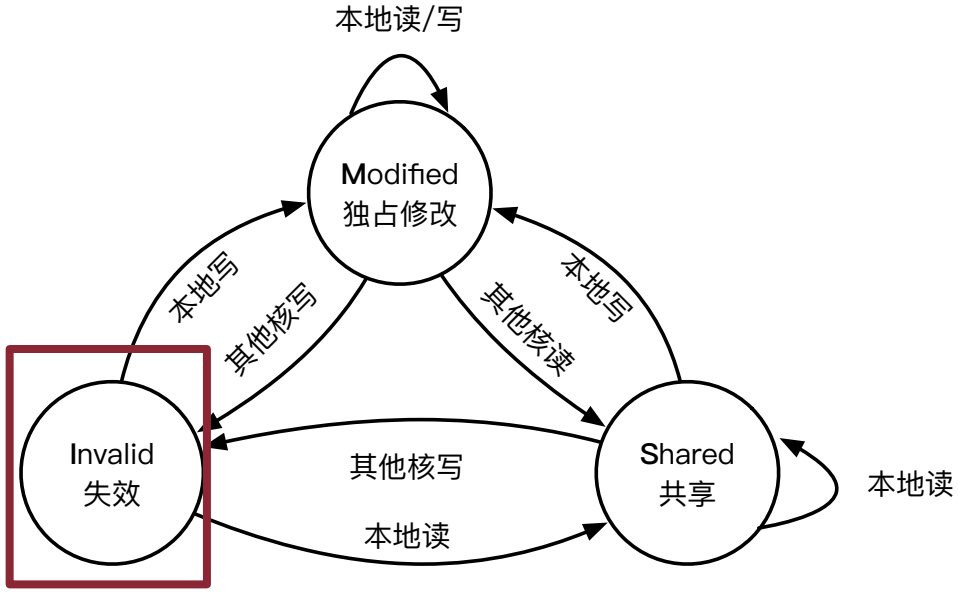
- 共享（**S**hared）
  - 可能多个核同时有缓存行的拷贝
  - 本地可**读**
  - 本地**写**需要迁移到**独占修改**，并使其他核该缓存行**失效**
  - 其他核**写**需要迁移到**失效**



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**

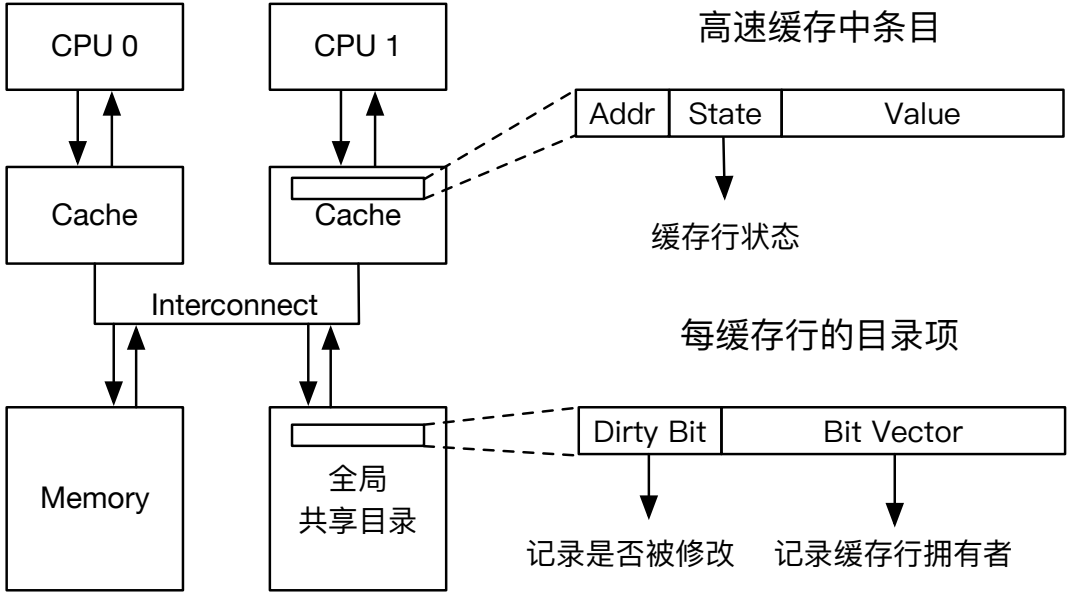


每核心每缓存行状态

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



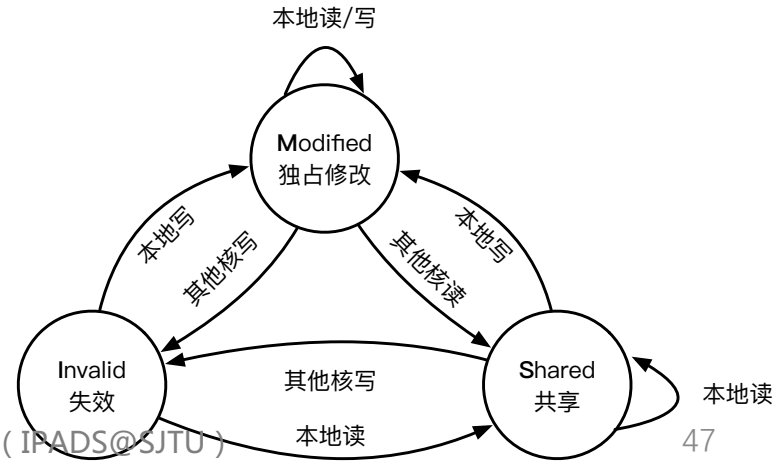
目录式缓存一致性：一个例子

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

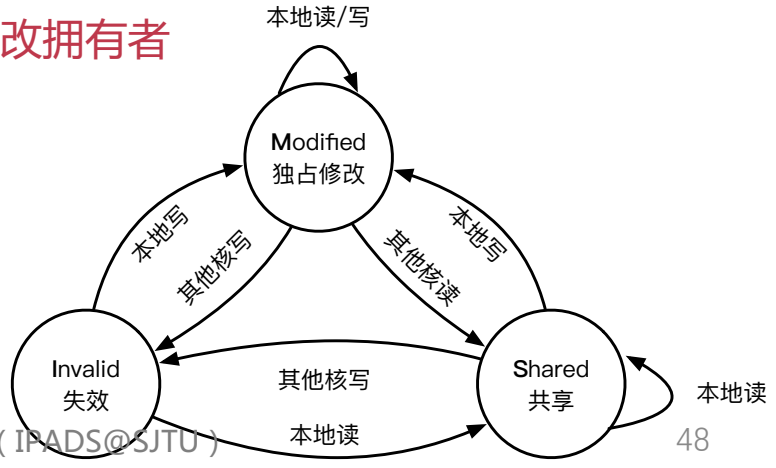
执行ST X,233

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

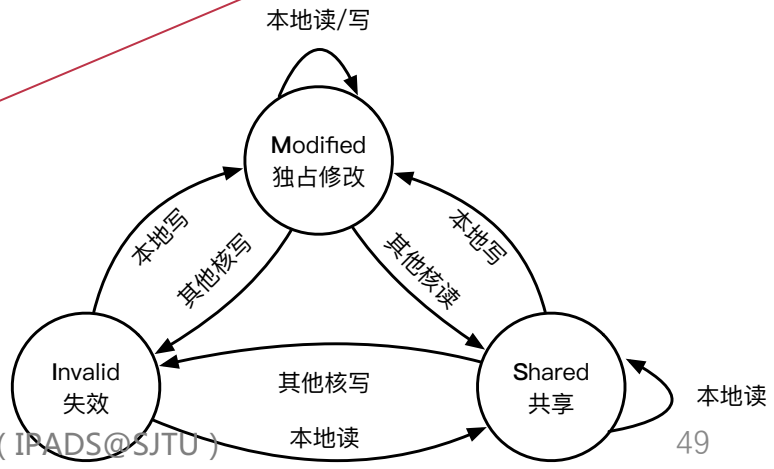
执行ST X,233

	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

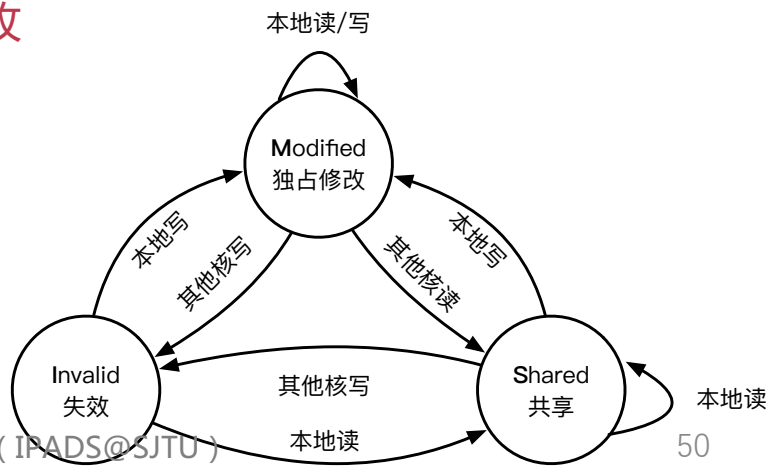
执行ST X,233

	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

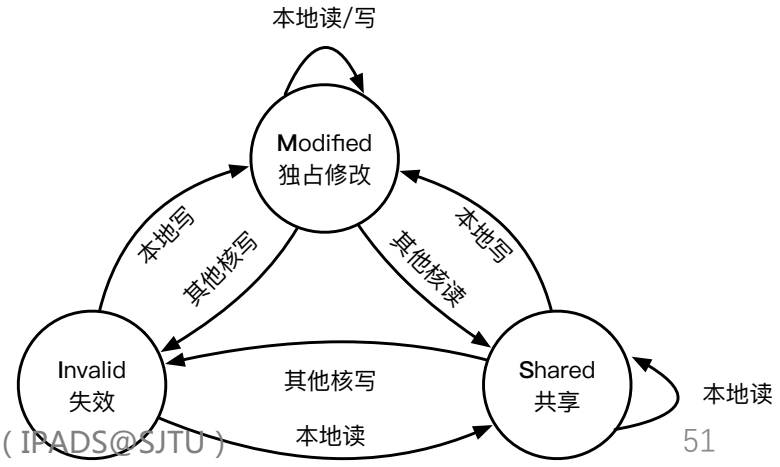
执行ST X,888

	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

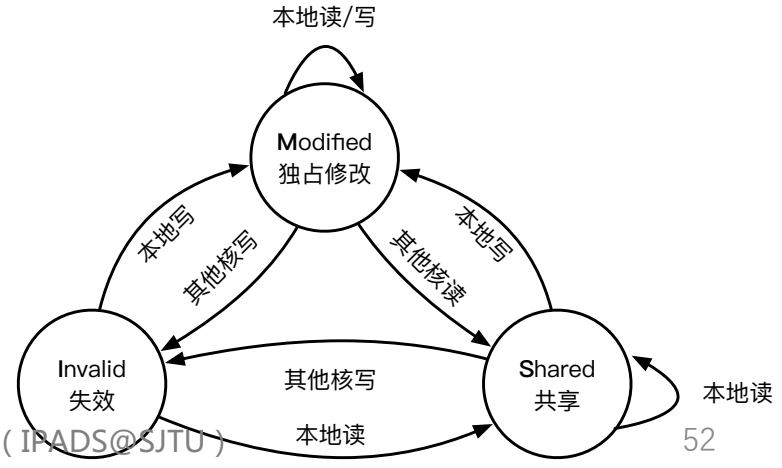
执行ST X,888

	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

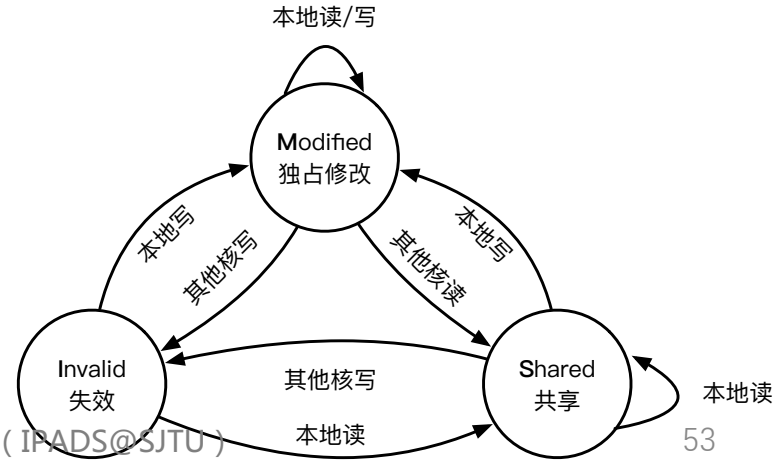
执行ST X,888

	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

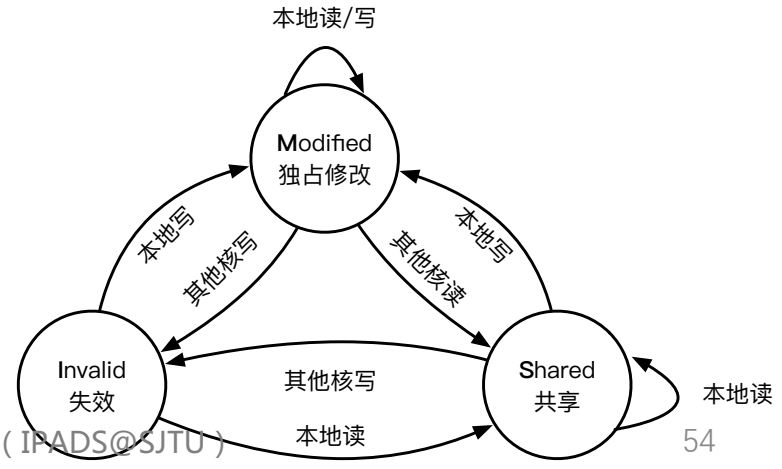
执行LD x

	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

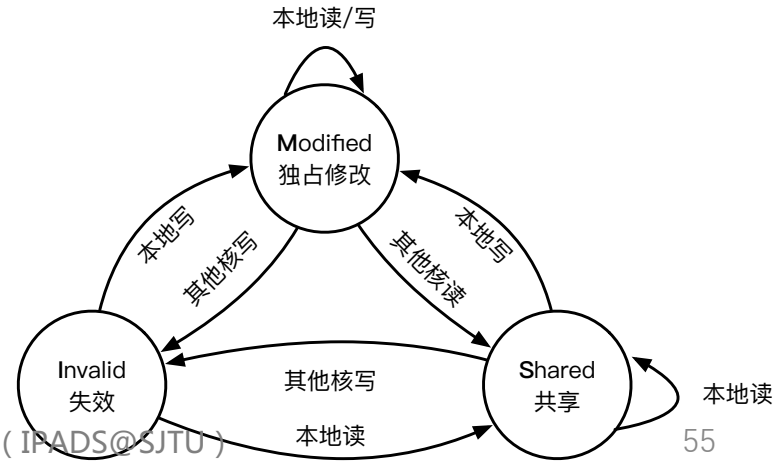
执行LD x

	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

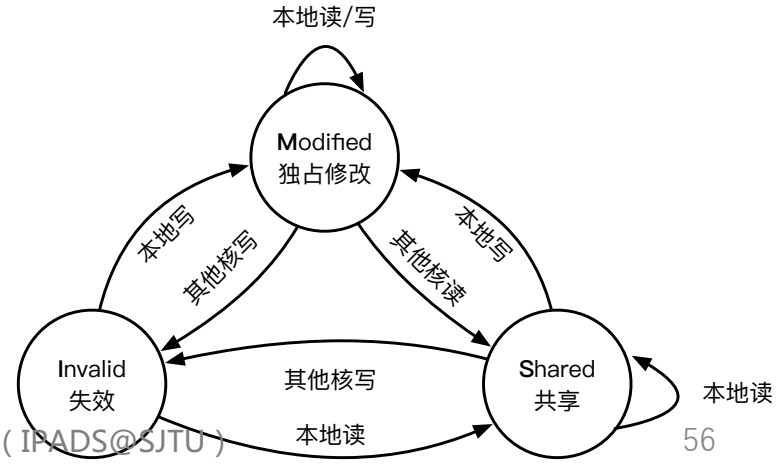
执行LD x

	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

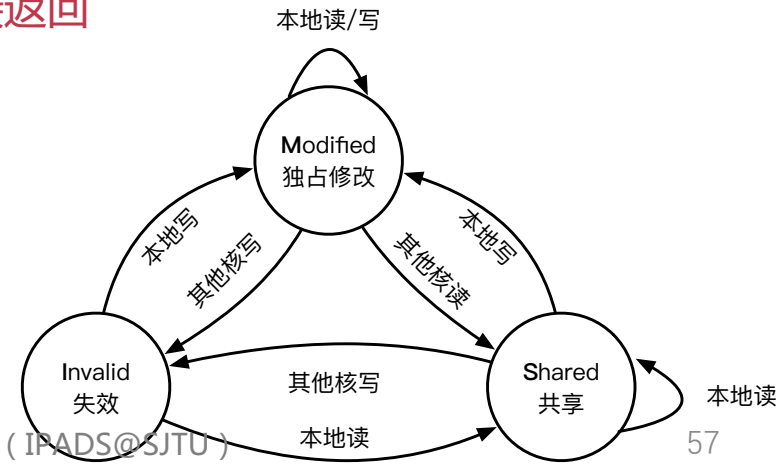
执行LD x

	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

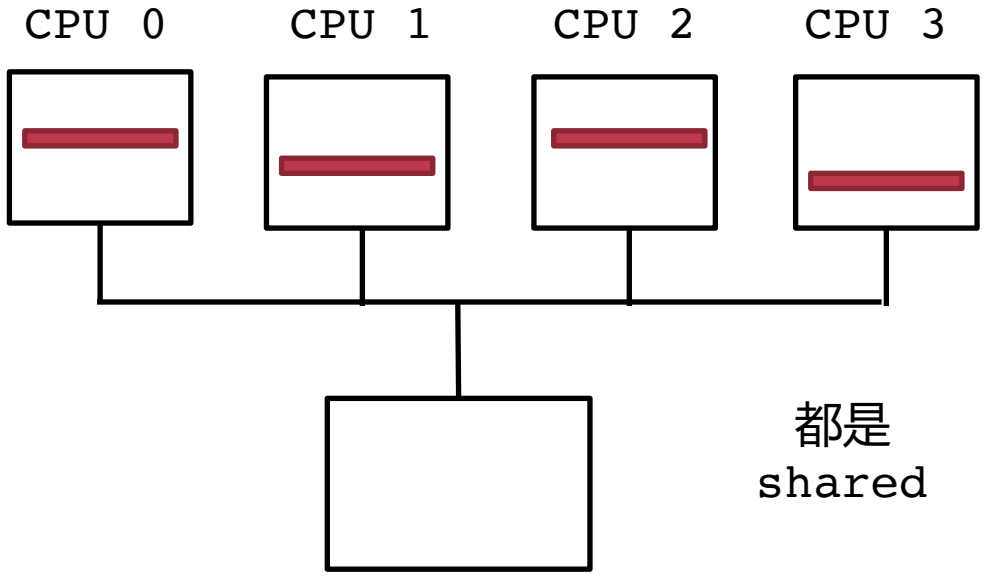
	Dirty	Bit Vector		
x	0	1	1	0



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



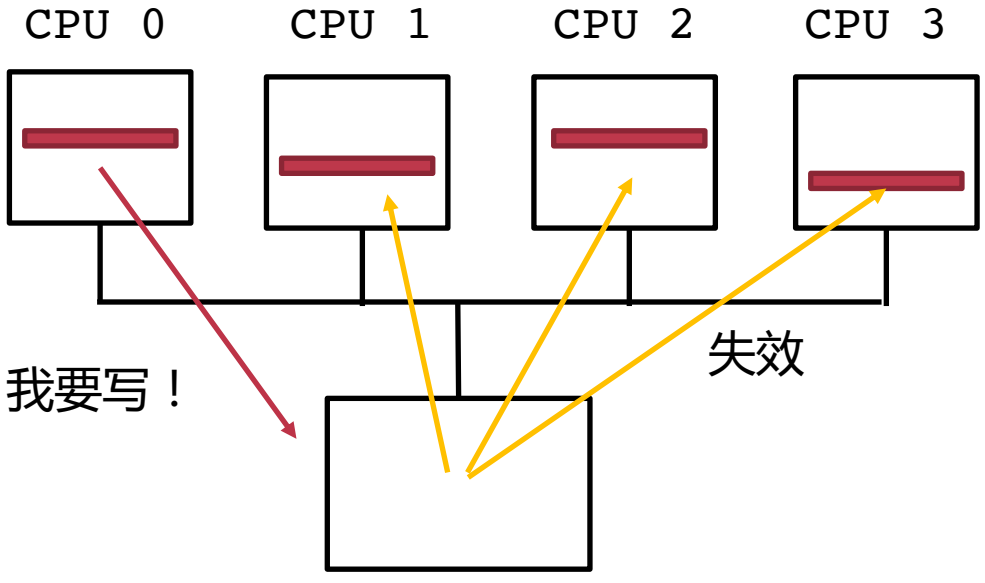
全局目录项

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



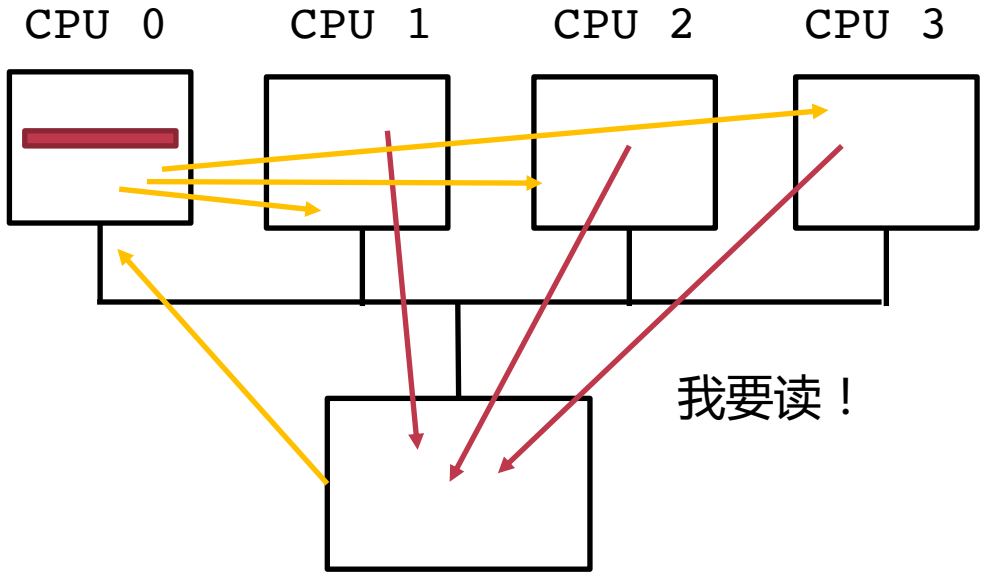
全局目录项

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



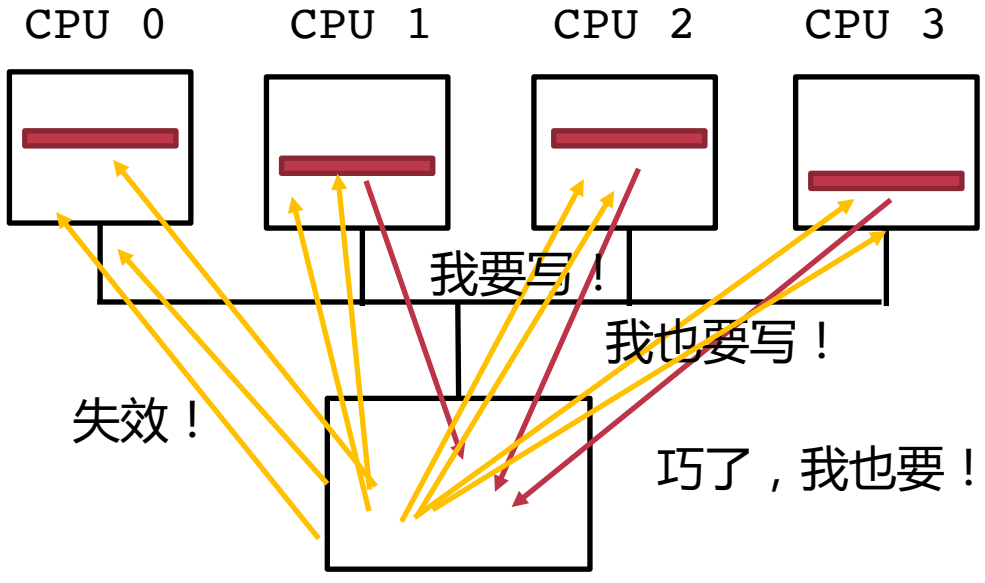
全局目录项

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.

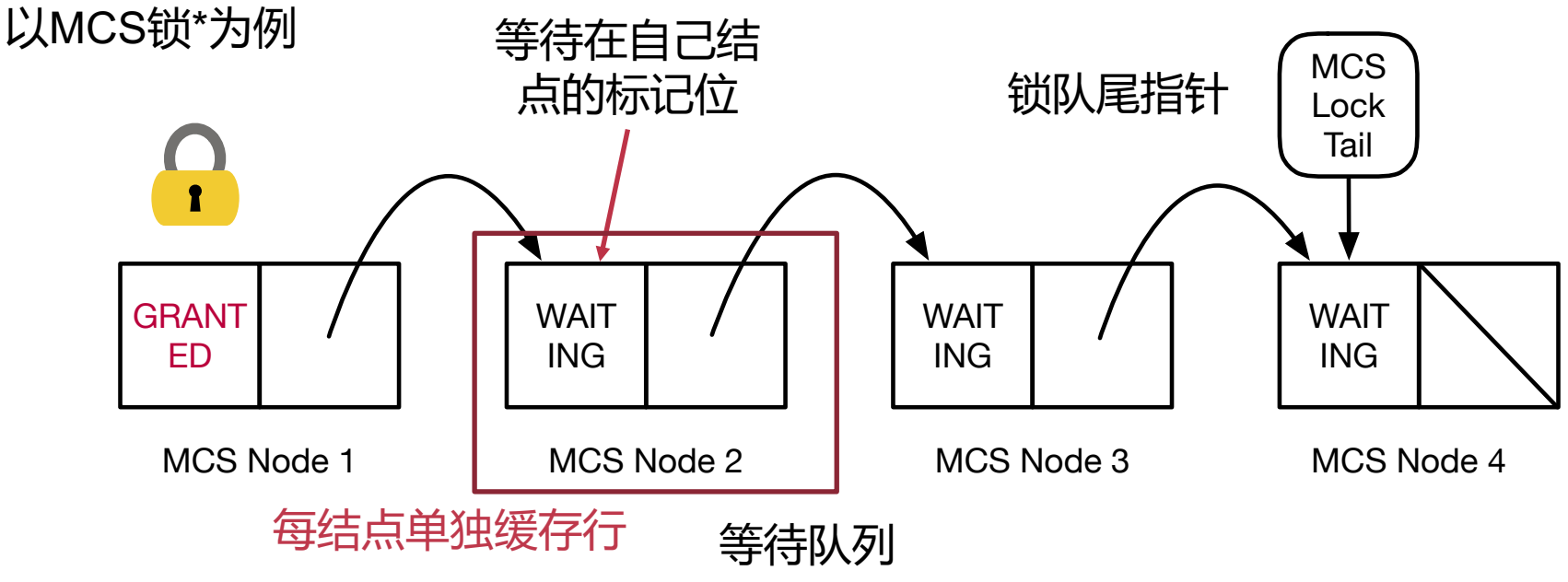
- if it does trigger in any half-way reasonably common setup (hardware/software), we most likely should work really hard at fixing the underlying problem, not the symptoms. 治标不治本

- 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.

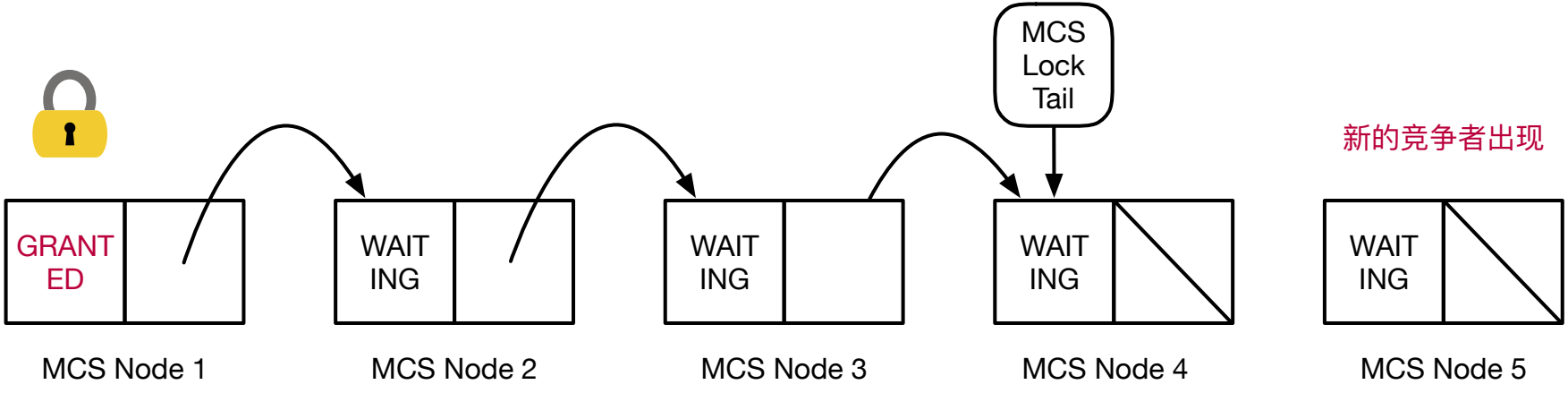
MCS实现示例

```
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;
}
```

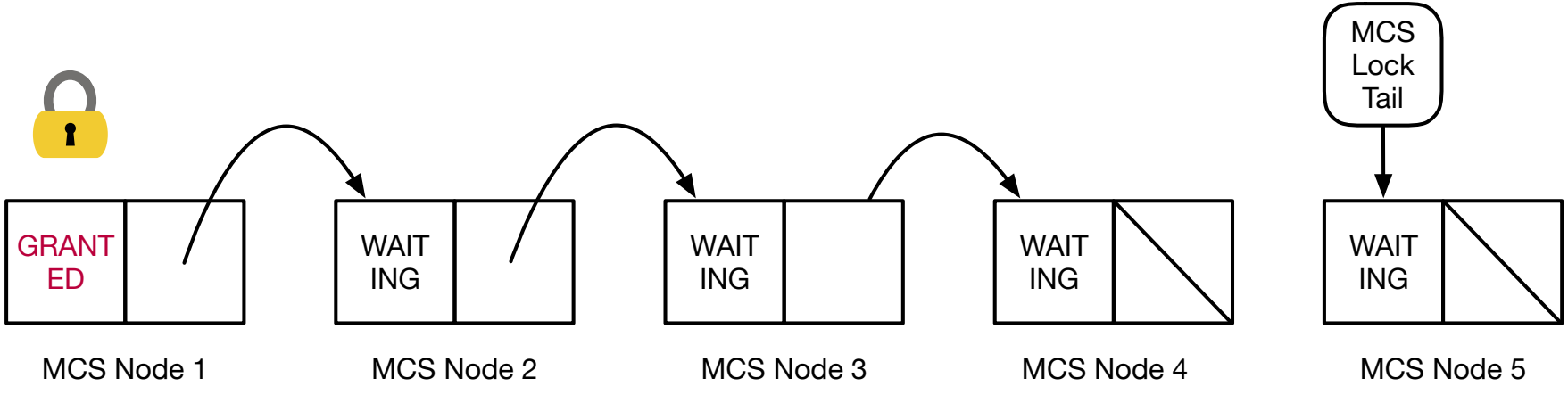
```
struct mcs_node {
        volatile struct mcs_node *next;
        volatile int flag;
} __attribute__((aligned(CACHELINE_SZ)));

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



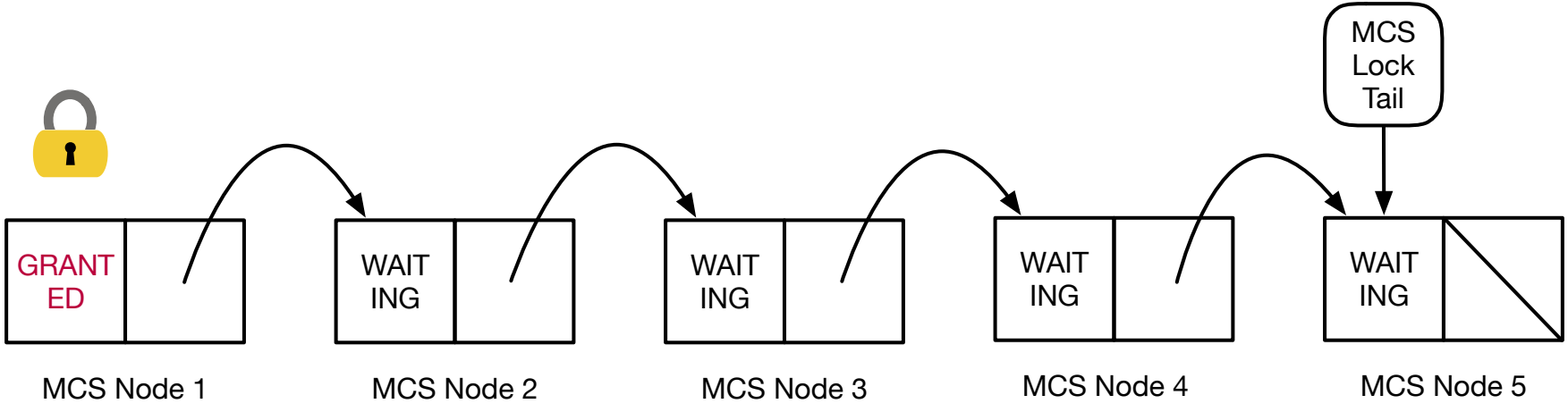
先填写自己结点的内容

MCS锁：新的竞争者加入等待队列



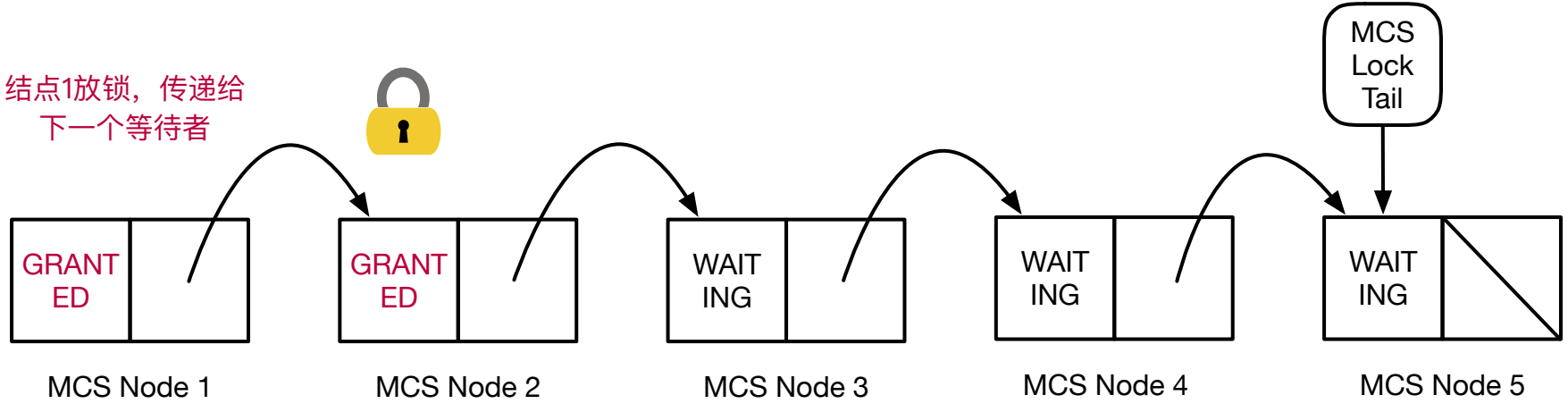
通过**原子操作**更新MCS锁的尾指针

MCS锁：新的竞争者加入等待队列

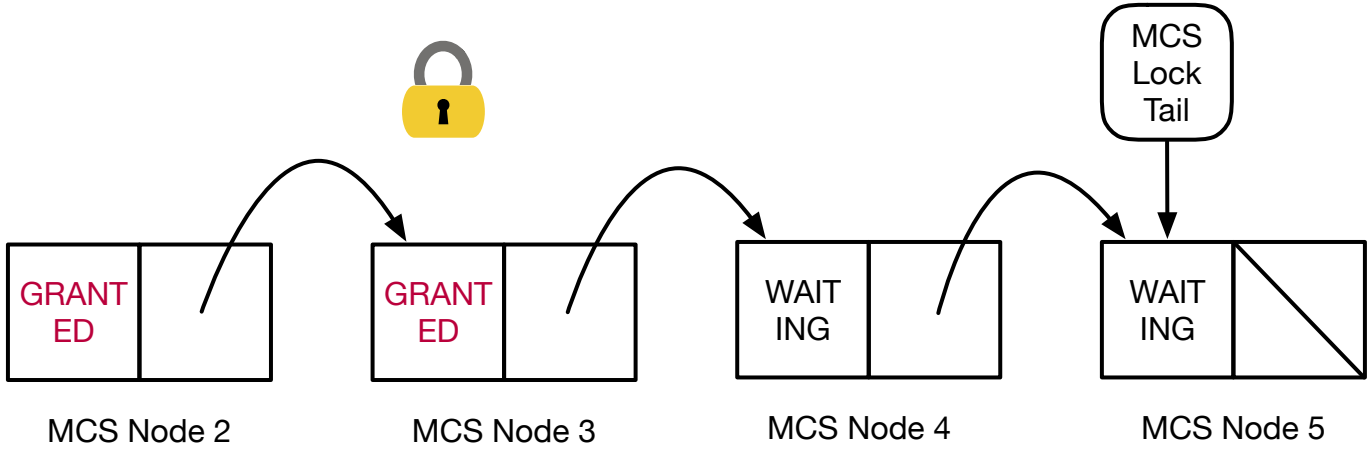


最后链接入等待队列

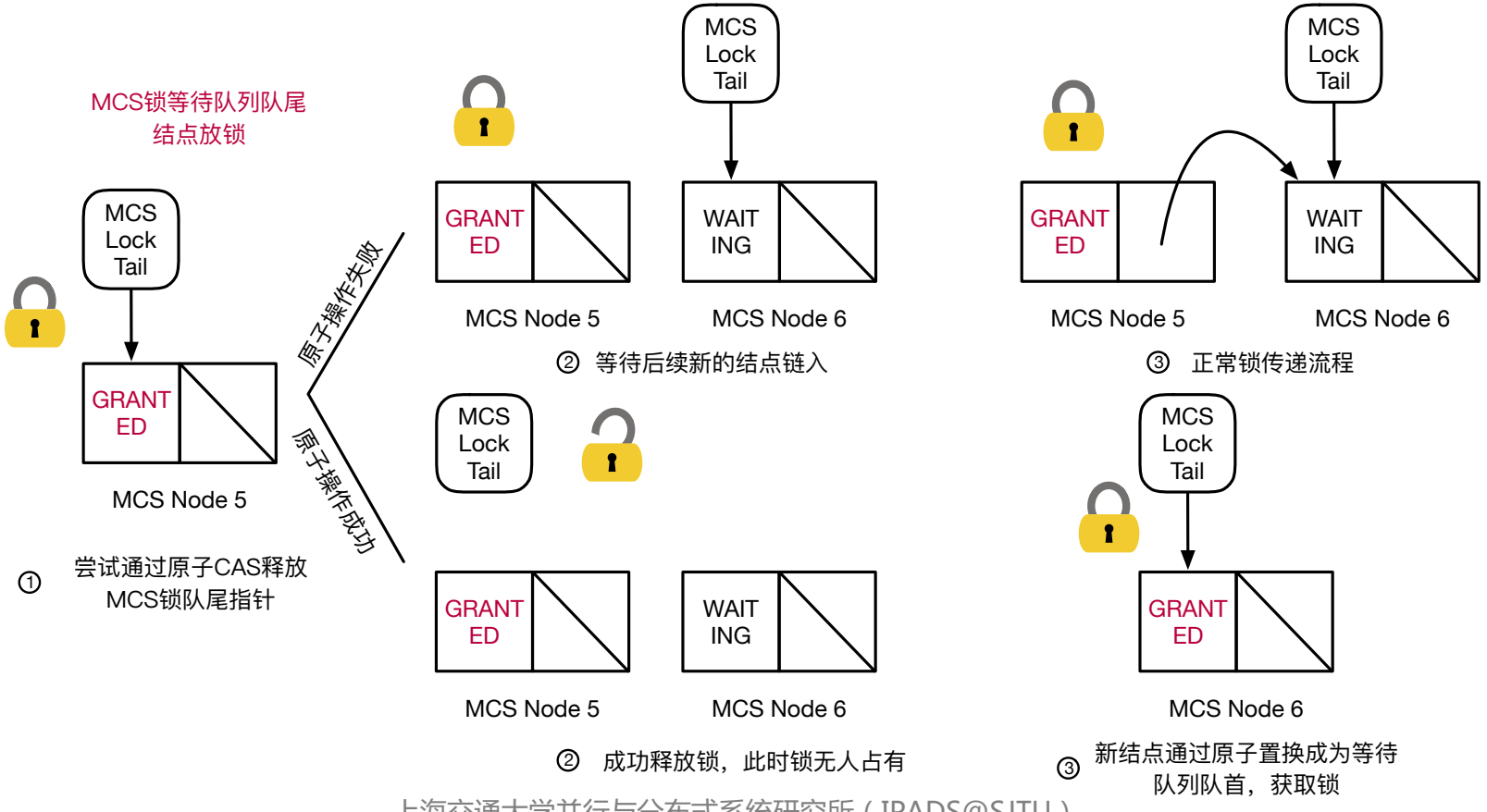
MCS锁：锁持有者的传递



MCS锁：锁持有者的传递

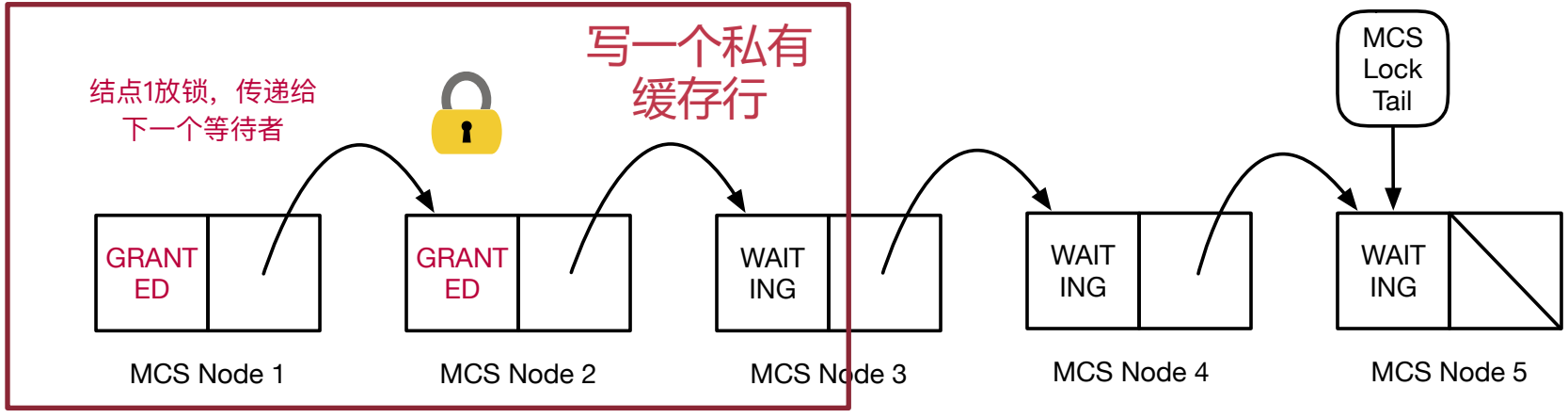


MCS锁：放锁流程

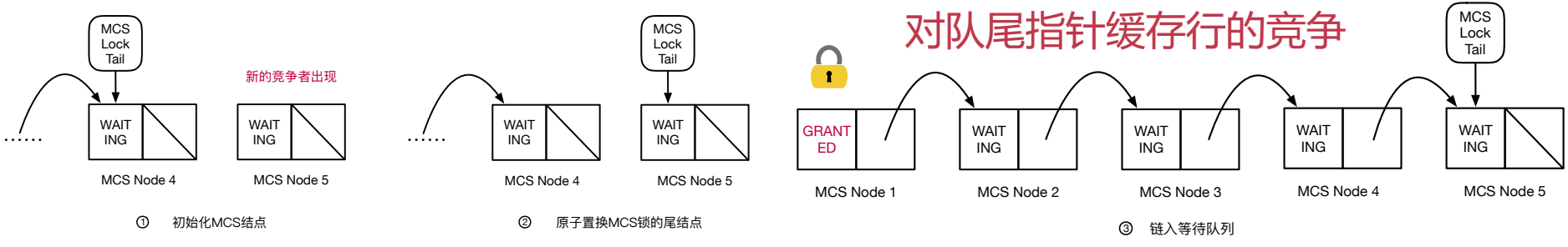

MCS锁等待队列队尾
结点放锁
MCS
Lock
Tail
GRANT
ED
MCS Node 5
原子操作失败
原子操作成功
① 尝试通过原子CAS释放
MCS锁队尾指针
MCS
Lock
Tail
GRANT
ED
WAIT
ING
MCS Node 5
MCS Node 6
② 等待后续新的结点链入
MCS
Lock
Tail
GRANT
ED
WAIT
ING
MCS Node 5
MCS Node 6
② 成功释放锁，此时锁无人占有
MCS
Lock
Tail
GRANT
ED
WAIT
ING
MCS Node 5
MCS Node 6
③ 正常锁传递流程
MCS
Lock
Tail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



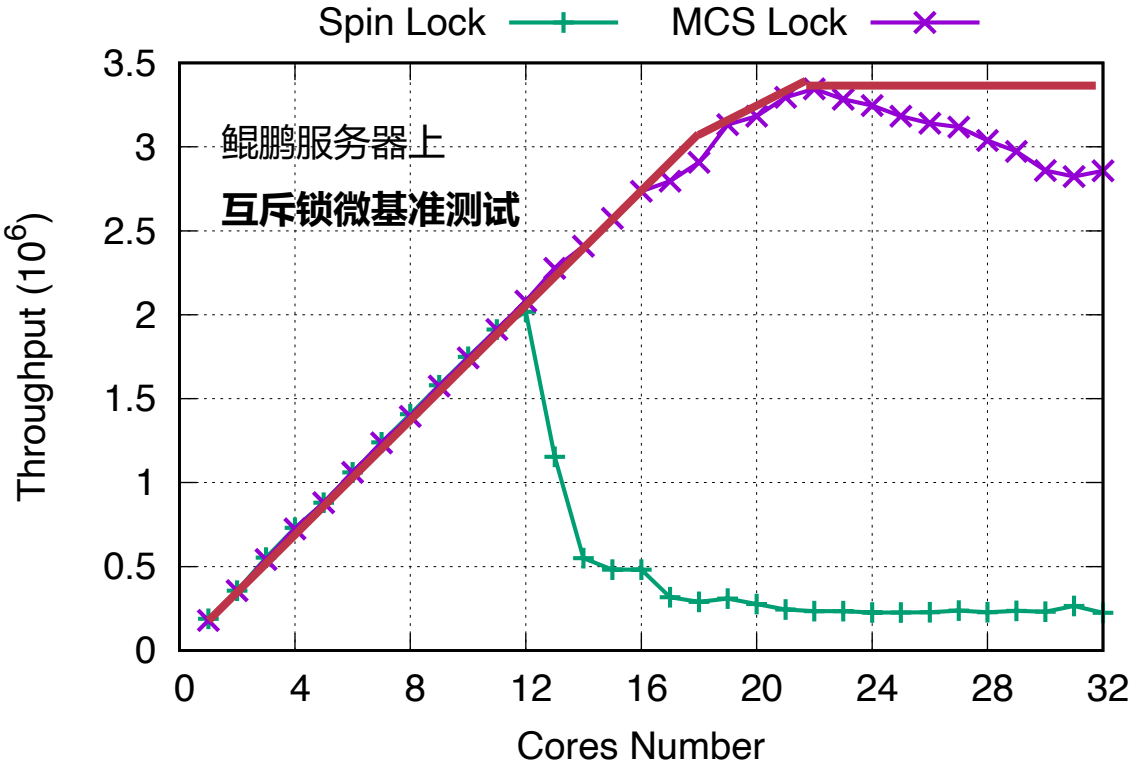
高竞争程度时的**关键路径**



高竞争程度时**关键路径之外**

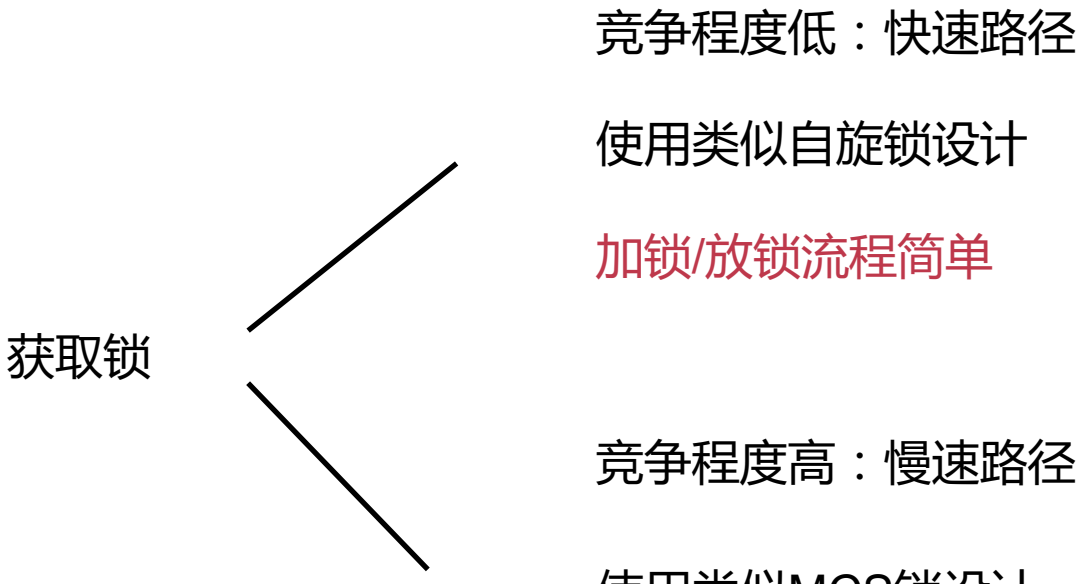
MCS锁：性能分析

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!

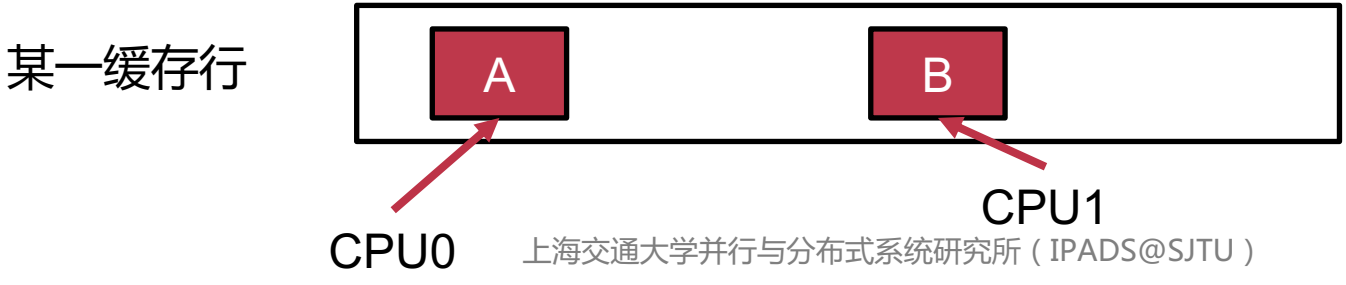
Linux Kernel中的可扩展锁：QSpinlock*



*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



Thanks