中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

多核与多处理器

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

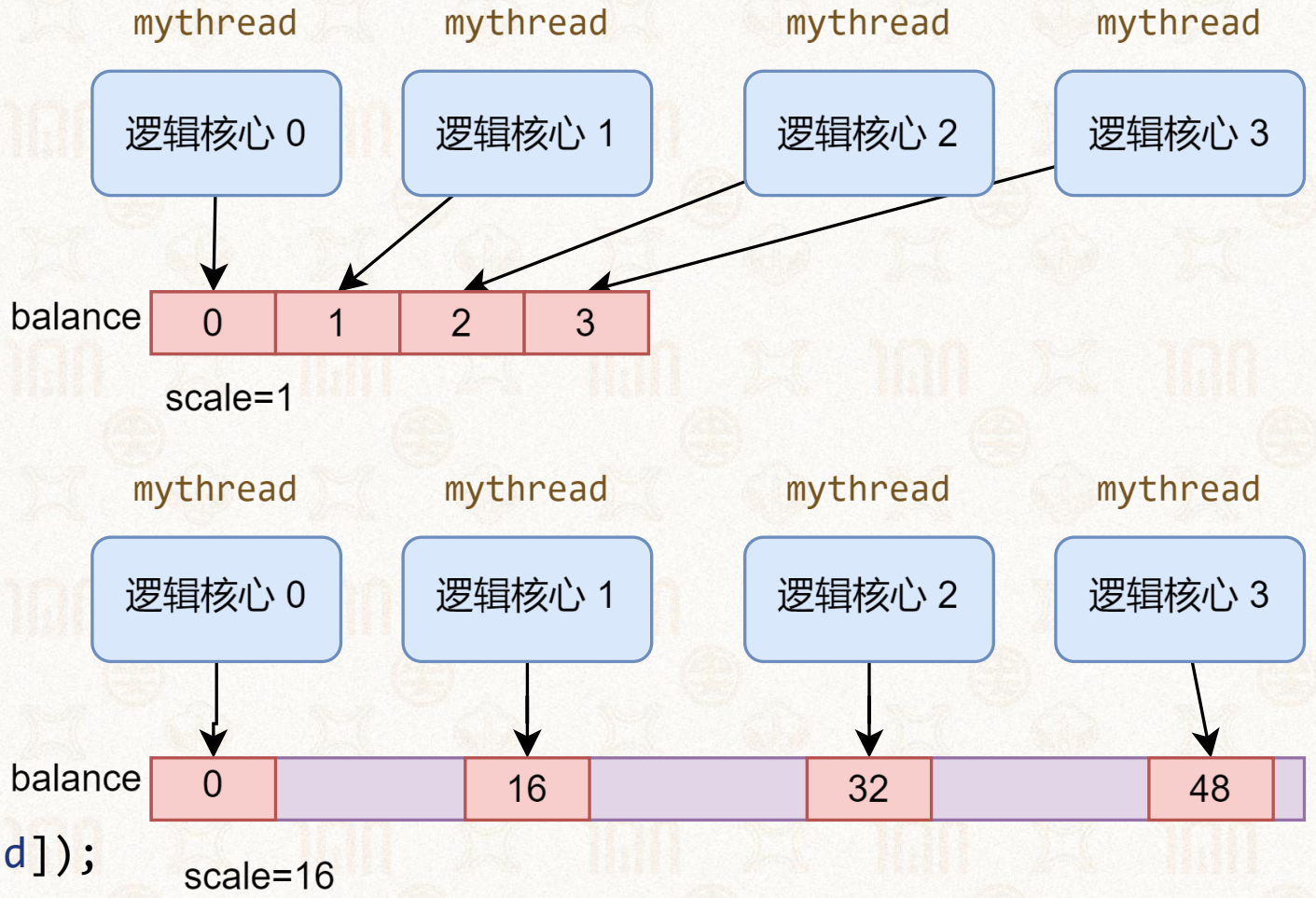
助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

不简单的代码

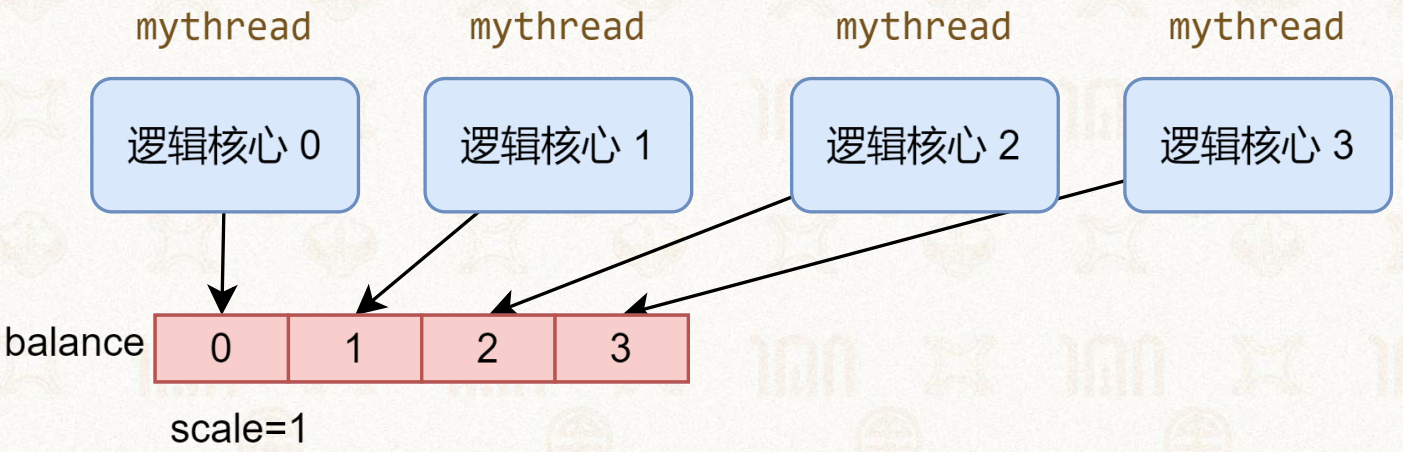
```
int num_threads = 4;
int scale = 1;
int size = 200000000;
int* balance;

void *mythread(void *arg) {
    int tid = *(int *) (arg);
    // tid = 0,1,2,3...
    tid *= scale;

    for (int i = 0; i < size; i++) {
        balance[tid]++;
    }

    printf("Balance is % d\n", balance[tid]);
    return NULL;
}
int main(int argc, char *argv[]) {
    balance = (int*) malloc(num_threads * sizeof(int) * scale);
    // 以下代码是启动num_threads个线程运行mythread，然后记录整体运行时间
```

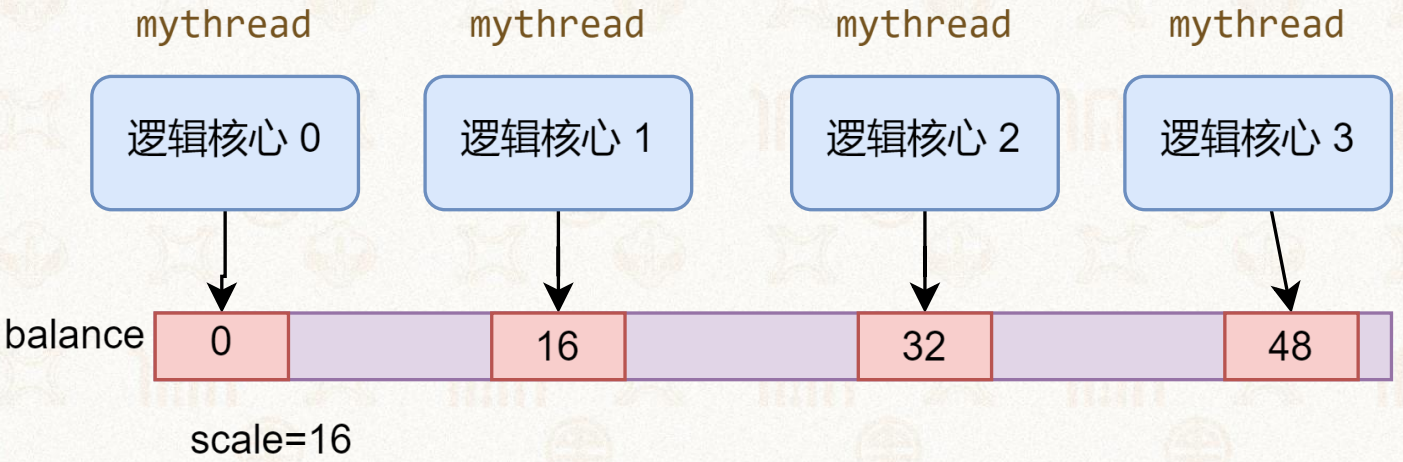


主观题 10分

设置

刚才那段代码，在scale分别为1和16时，程序运行时间有没有很大差别，为什么?





作答

单选题 1分

设置

此题未设置答案，请点击右侧设置按钮

两段代码谁跑得快

```
int size = 200000000;

struct {
    int a; // thread_0
    int b; // thread_1
    char c[64];
} data;

void *thread_0(void *arg) {
    for(int i = 0; i < size; i++) {
        data.a = 0;
    }
}
void *thread_1(void *arg) {
    for(int i = 0; i < size; i++) {
        data.b = 1;
    }
}
```

```
int size = 200000000;

struct {
    int a; // thread_0
    char c[64];
    int b; // thread_1
} data;

void *thread_0(void *arg) {
    for(int i = 0; i < size; i++) {
        data.a = 0;
    }
}
void *thread_1(void *arg) {
    for(int i = 0; i < size; i++) {
        data.b = 1;
    }
}
```

左边快　　都差不多　　右边快

提交

单选题 1分

设置

此题未设置答案，请点击右侧设置按钮

两段代码谁跑得快

```
struct {
    int a; // thread_0
    int b; // thread_1
    char c[64];
} data;

void *thread_0(void *arg) {
    int result;
    for(int i = 0; i < size; i++) {
        result = data.a;
    }
}
void *thread_1(void *arg) {
    int result;
    for(int i = 0; i < size; i++) {
        result = data.b;
    }
}
```

```
struct {
    int a; // thread_0
    char c[64];
    int b; // thread_1
} data;

void *thread_0(void *arg) {
    int result;
    for(int i = 0; i < size; i++) {
        result = data.a;
    }
}
void *thread_1(void *arg) {
    int result;
    for(int i = 0; i < size; i++) {
        result = data.b;
    }
}
```

左边快　　都差不多　　右边快

提交

大纲

- 多核性能问题

- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性

- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发

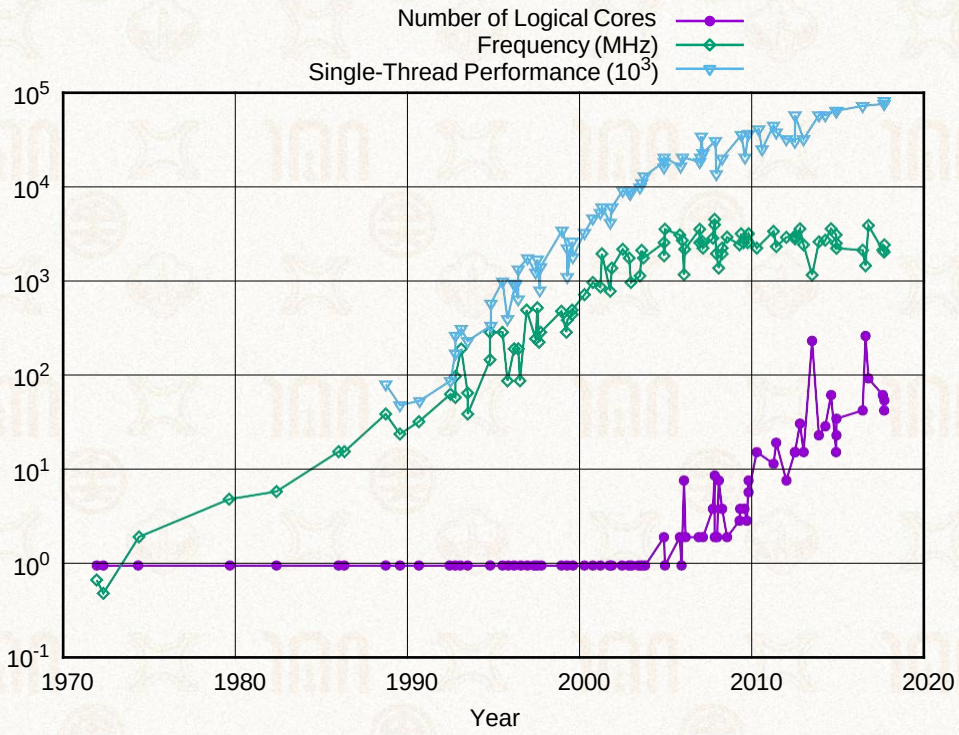
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)

- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计

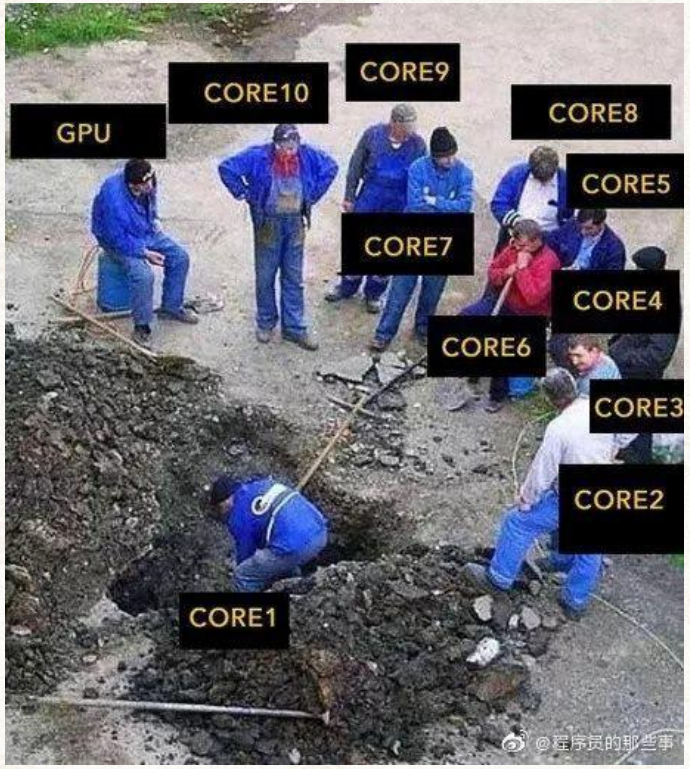
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

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CPU核数来提升软件的性能
- 桌面/移动平台均向多核迈进



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

操作系统在多处理器多核环境下面临的问题

正确性保证

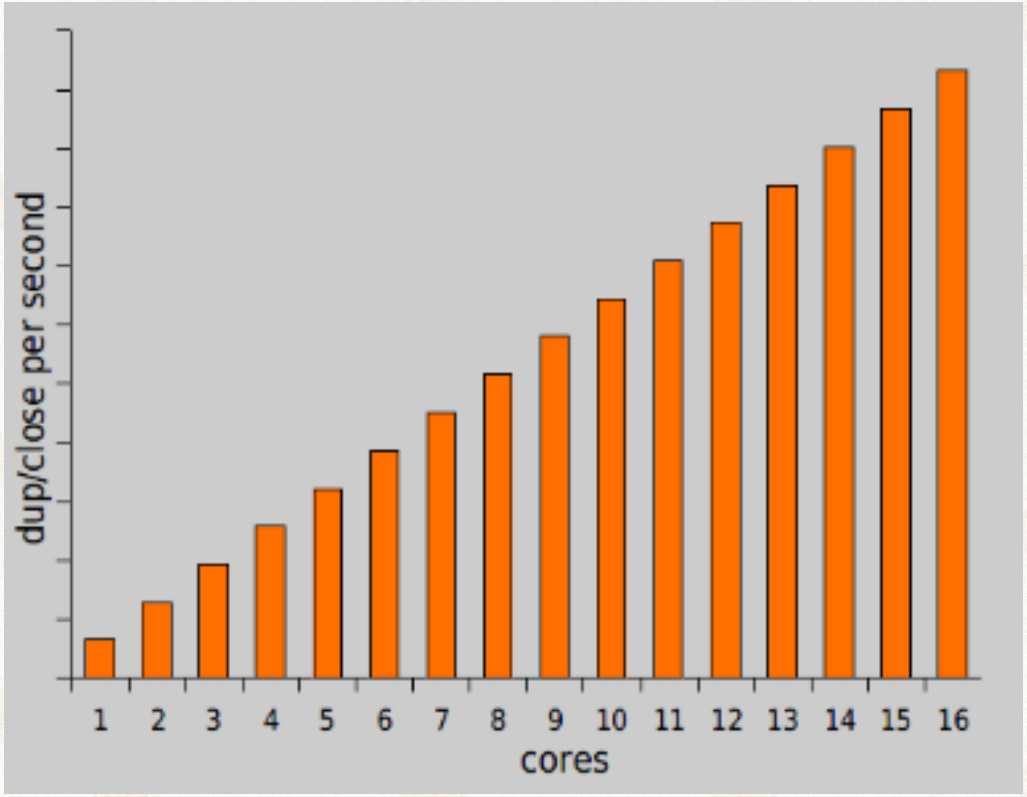
- 对共享资源的竞争导致错误
- 操作系统提供同步原语供开发者使用
- 使用同步原语带来新的问题

性能保证

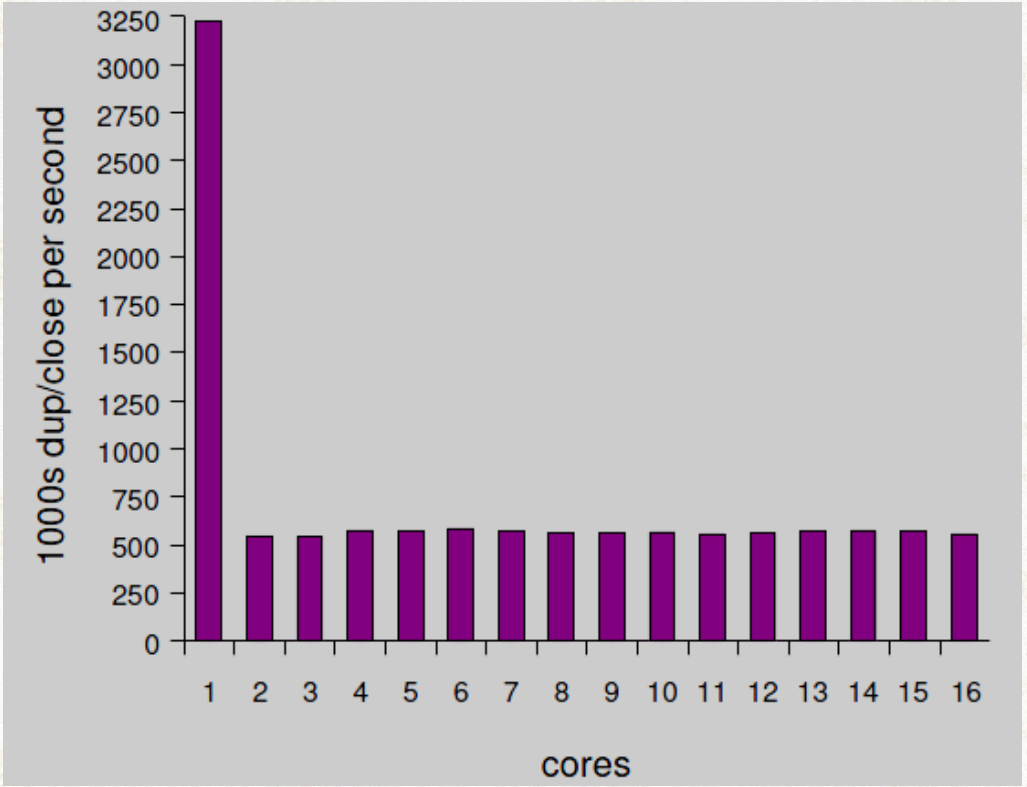
- 多核多处理器硬件与特性
- 可扩展性问题导致性能断崖
- 系统软件设计如何利用硬件特性

- 现在讨论

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 (阿姆达尔定律)

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

加速比

可以并行部分代码占比

同时执行的核心数

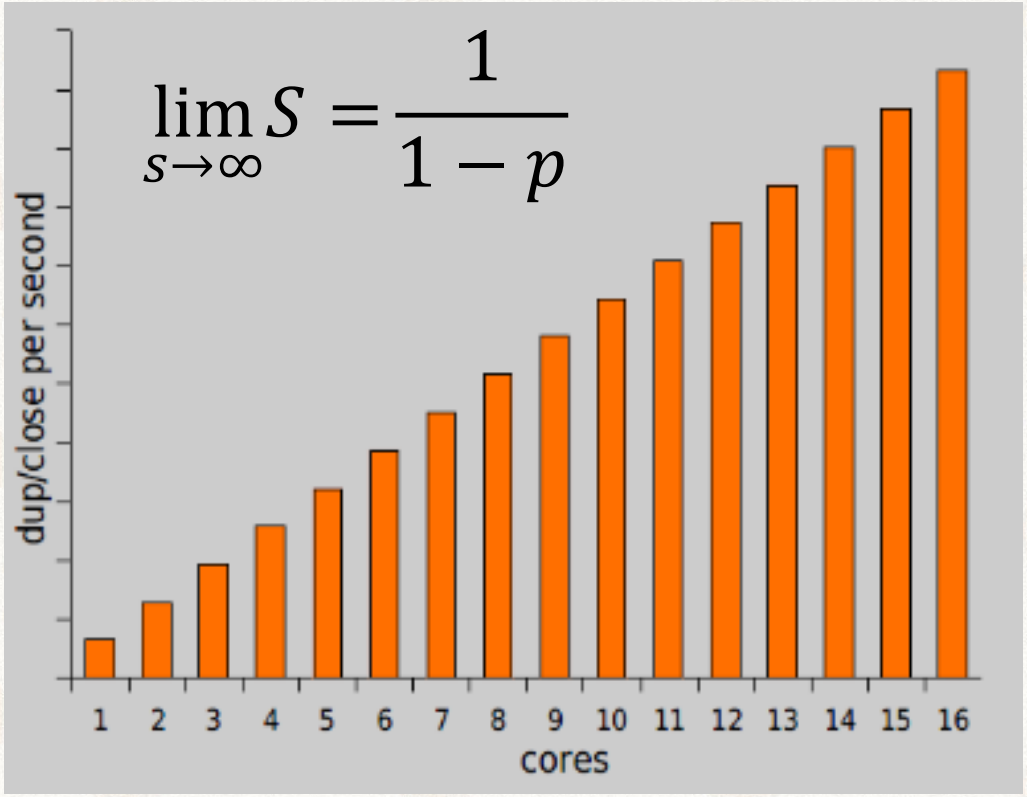
当核心数增加时... $\lim_{s \to \infty} S = \frac{1}{1-p}$

```
for(int i = 0; i < 100; i++) {
    a[i] = func(a[i], i);
}
```

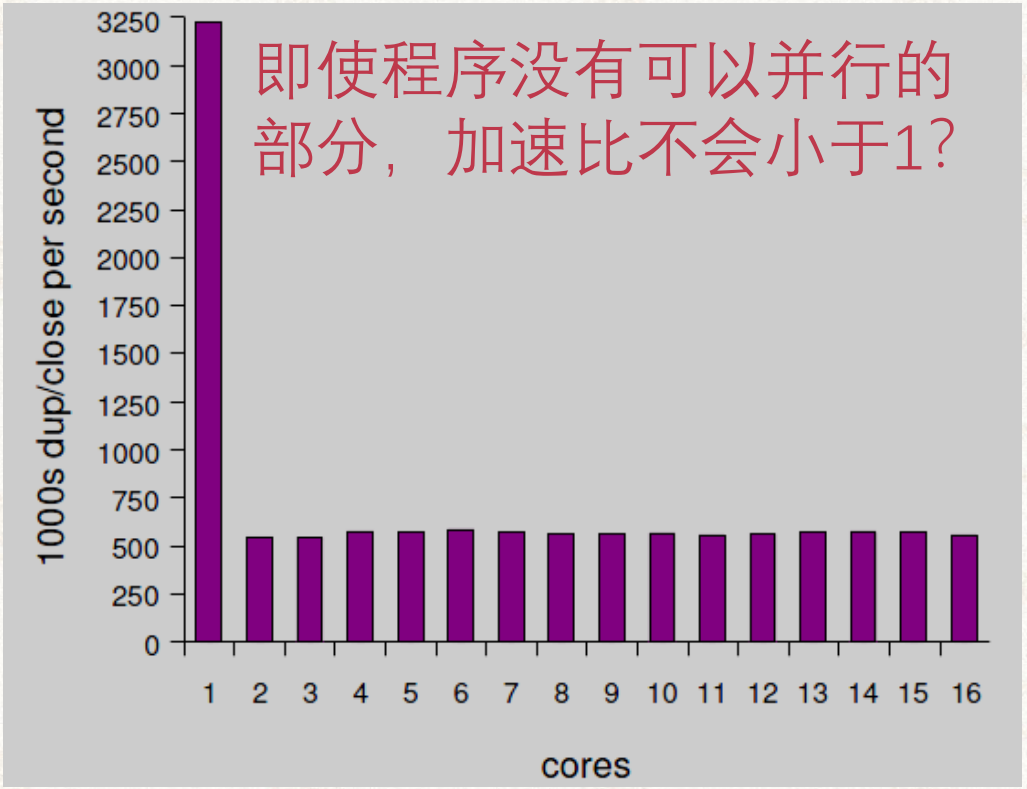
```
for(int i = 1; i < 100; i++) {
    a[i] = func(a[i - 1], i);
}
```

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

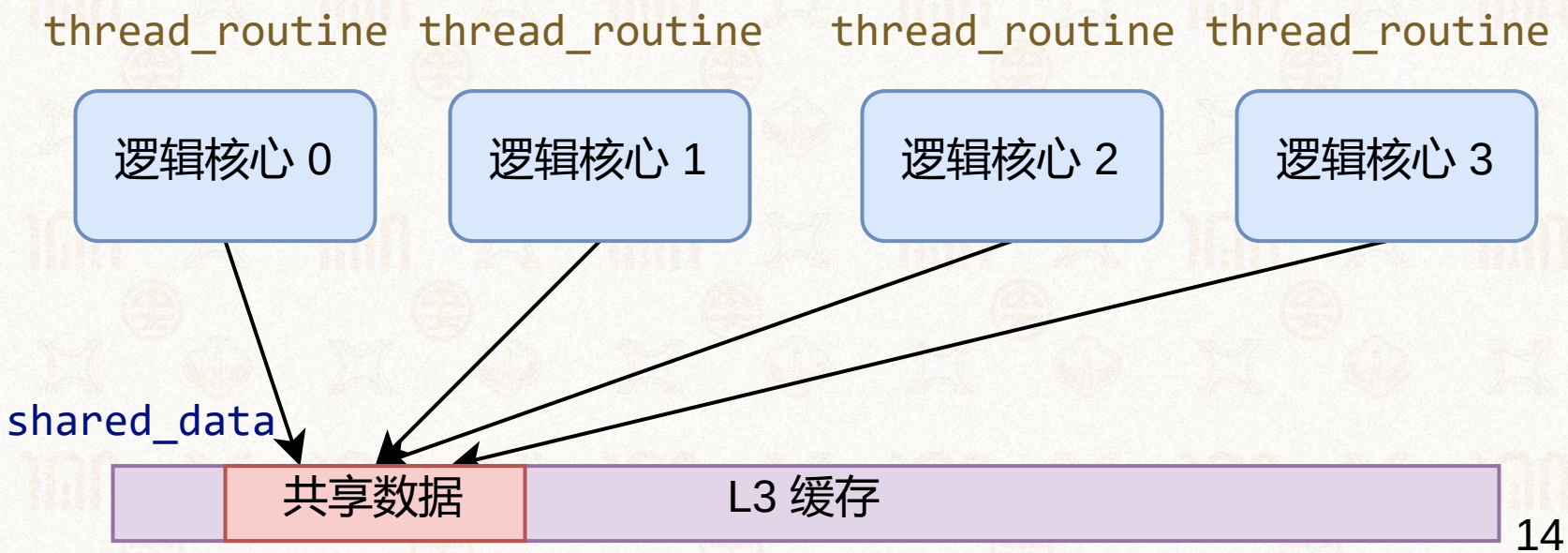


实际fd性能

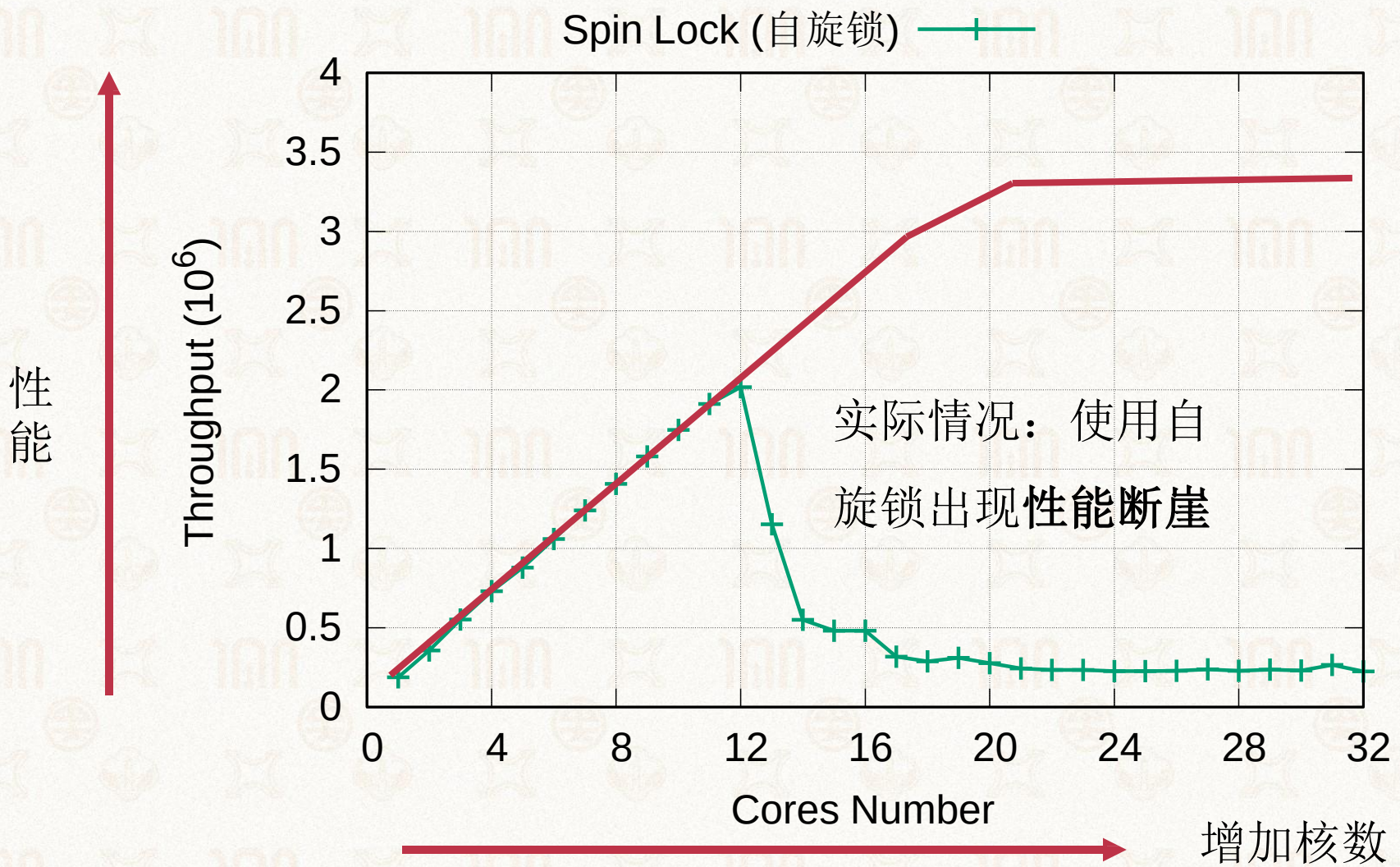
互斥锁微基准测试

```
struct lock *glock;
unsigned long gcnt = 0;
char shared_data[CACHE_LINE_SIZE];
void *thread_routine(void *arg) {
    while (1) {
        lock(glock);
        // 进入临界区
        gcnt = gcnt + 1;
        // 访问缓存行中的共享的数据
        visit_shared_data(shared_data, 1);
        unlock(glock);
        interval();
    }
}
```

使用微基准测试来复现这个现象

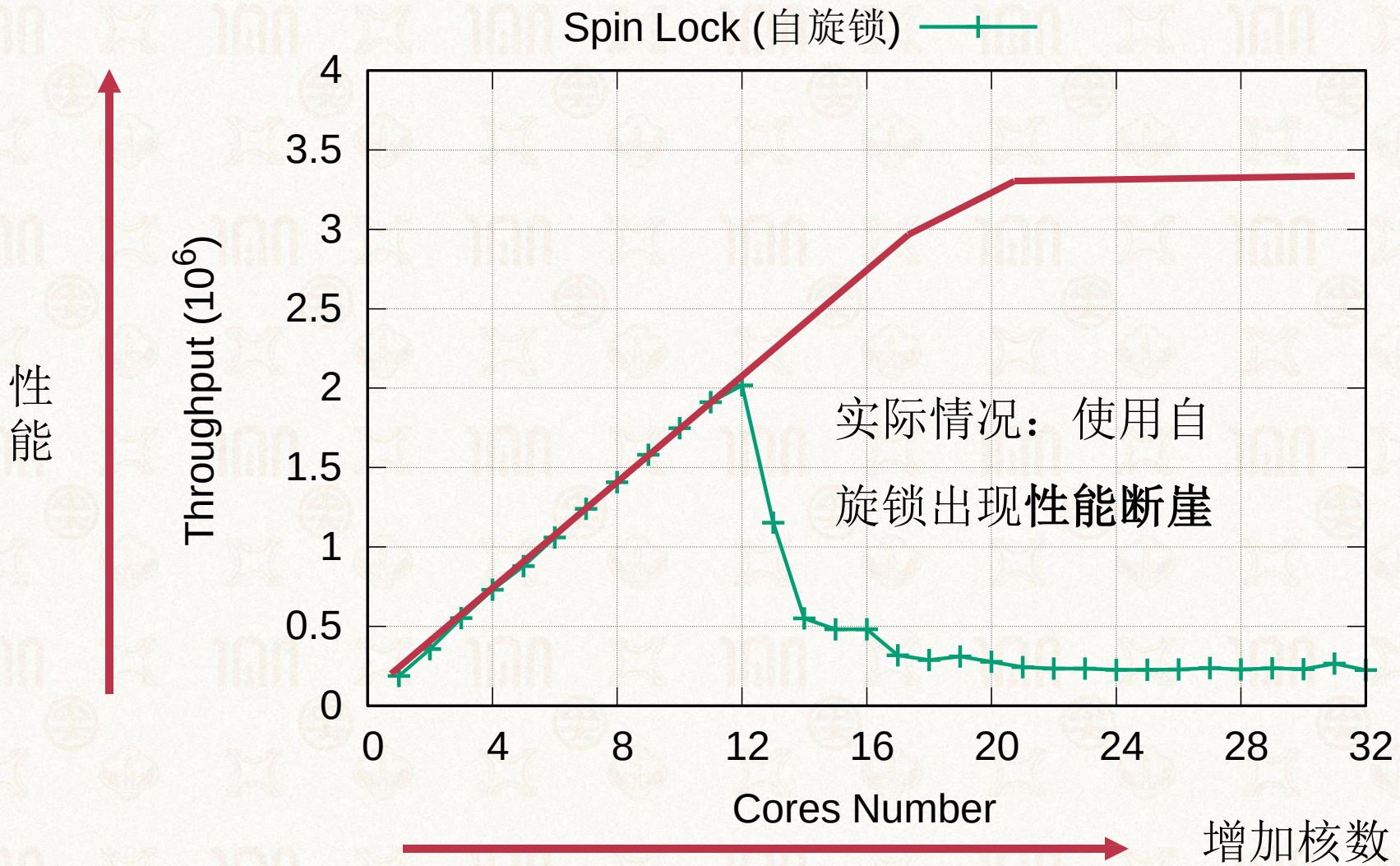


可扩展性断崖



合理的理想性能曲线

可扩展性断崖



合理的理想性能曲线

为什么会出现这种情况？

扩展性差的锁是非常危险的

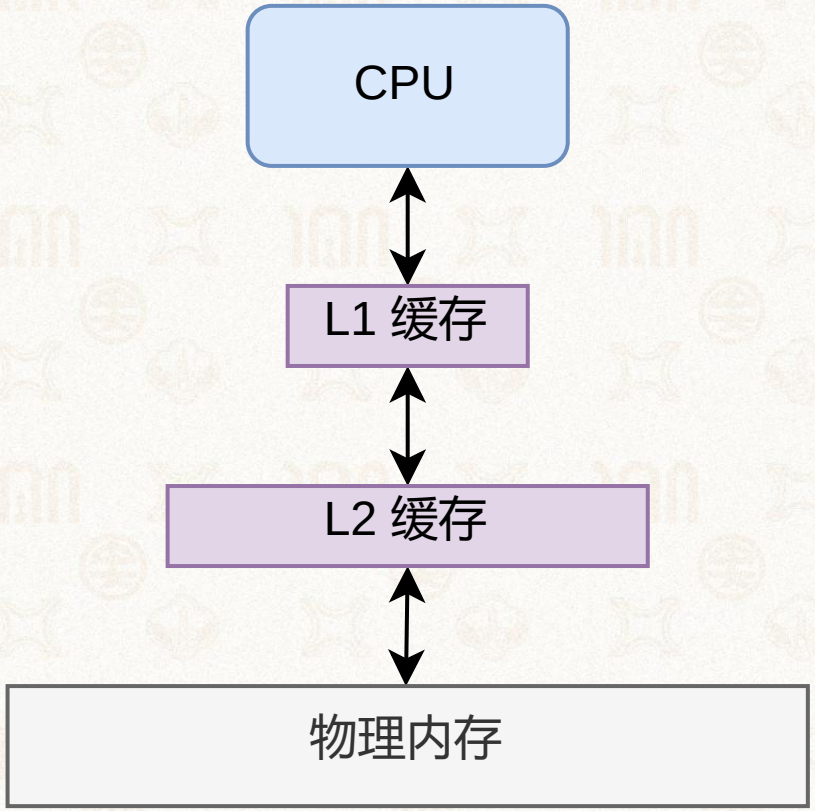
Boyd-Wickizer, Silas, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. "Non-scalable locks are dangerous." In Proceedings of the Linux Symposium, pp. 119-130. 2012.

大纲

- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

高速缓存(cache)回顾

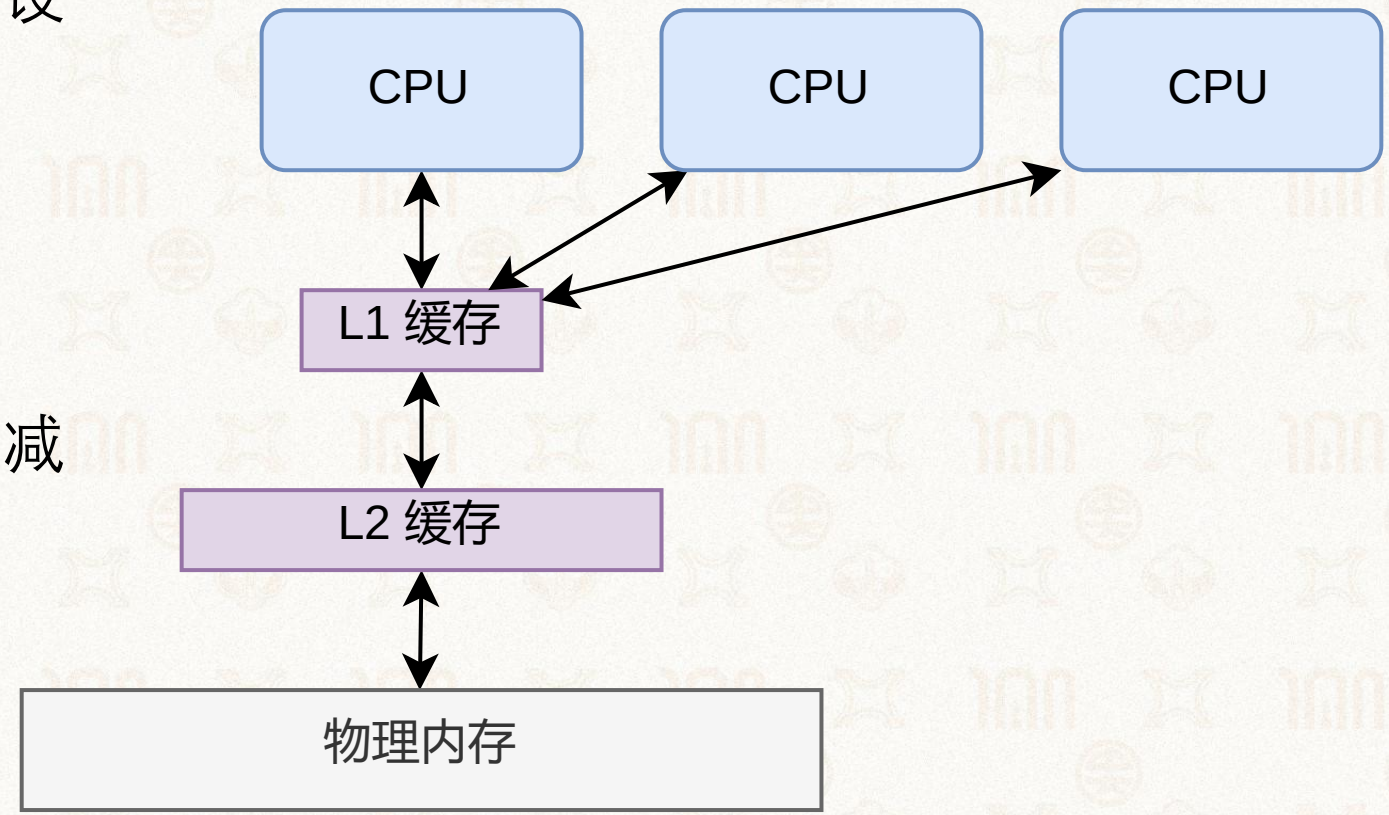
- 多级缓存：
  - 靠近CPU贵，速度快，容量小
  - 远离CPU便宜，速度慢，容量大
- 读操作：
  - 逐层向下找
  - 没找到从内存中读取，放到缓存中
- 写操作：
  - 直写/写回策略
  - 写入高速缓存，替换时写回



多处理器多核环境中的缓存结构

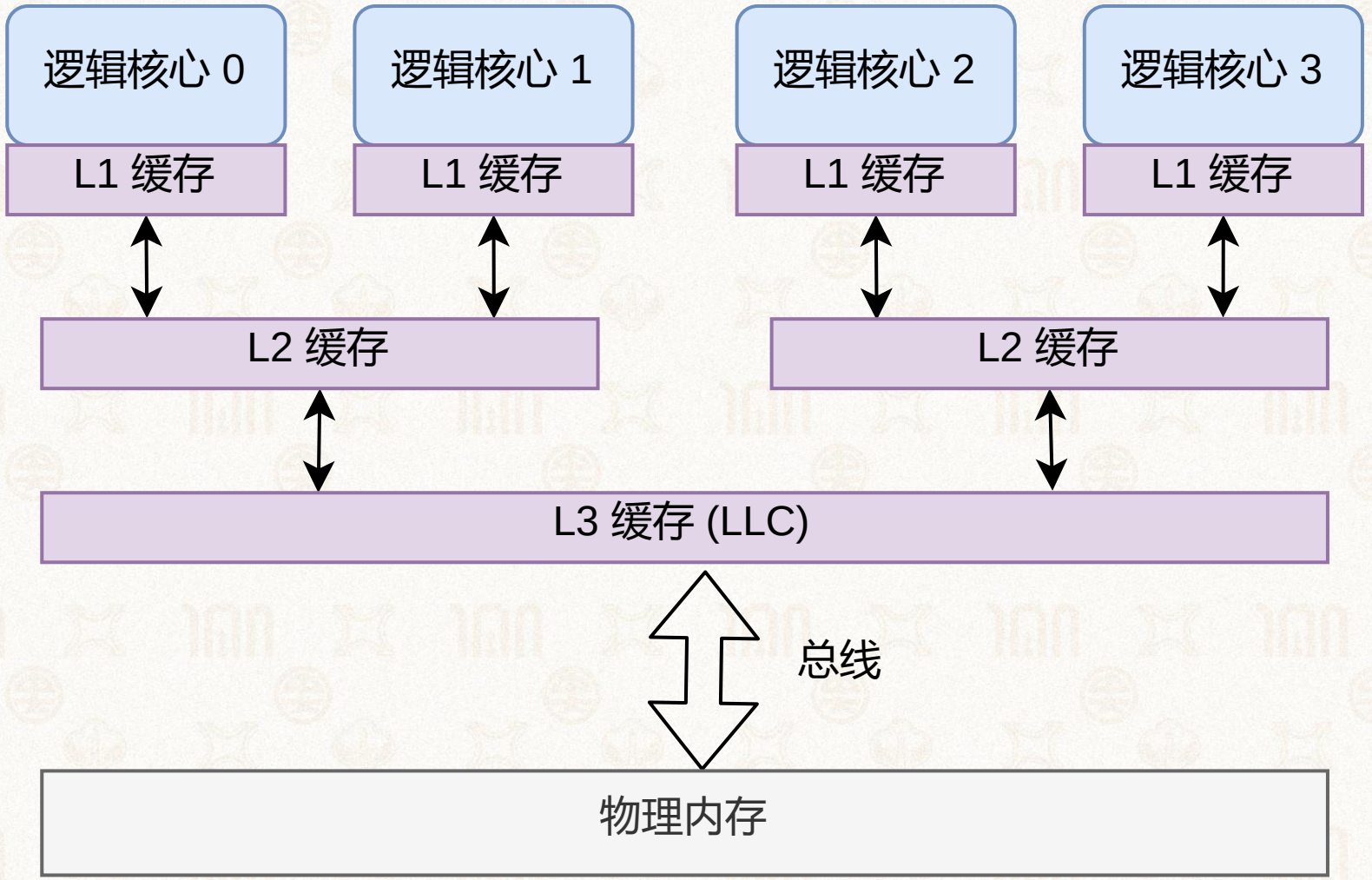
- 简单的解决方案：
  - 将多核当成一个核心，共享缓存设计

- 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

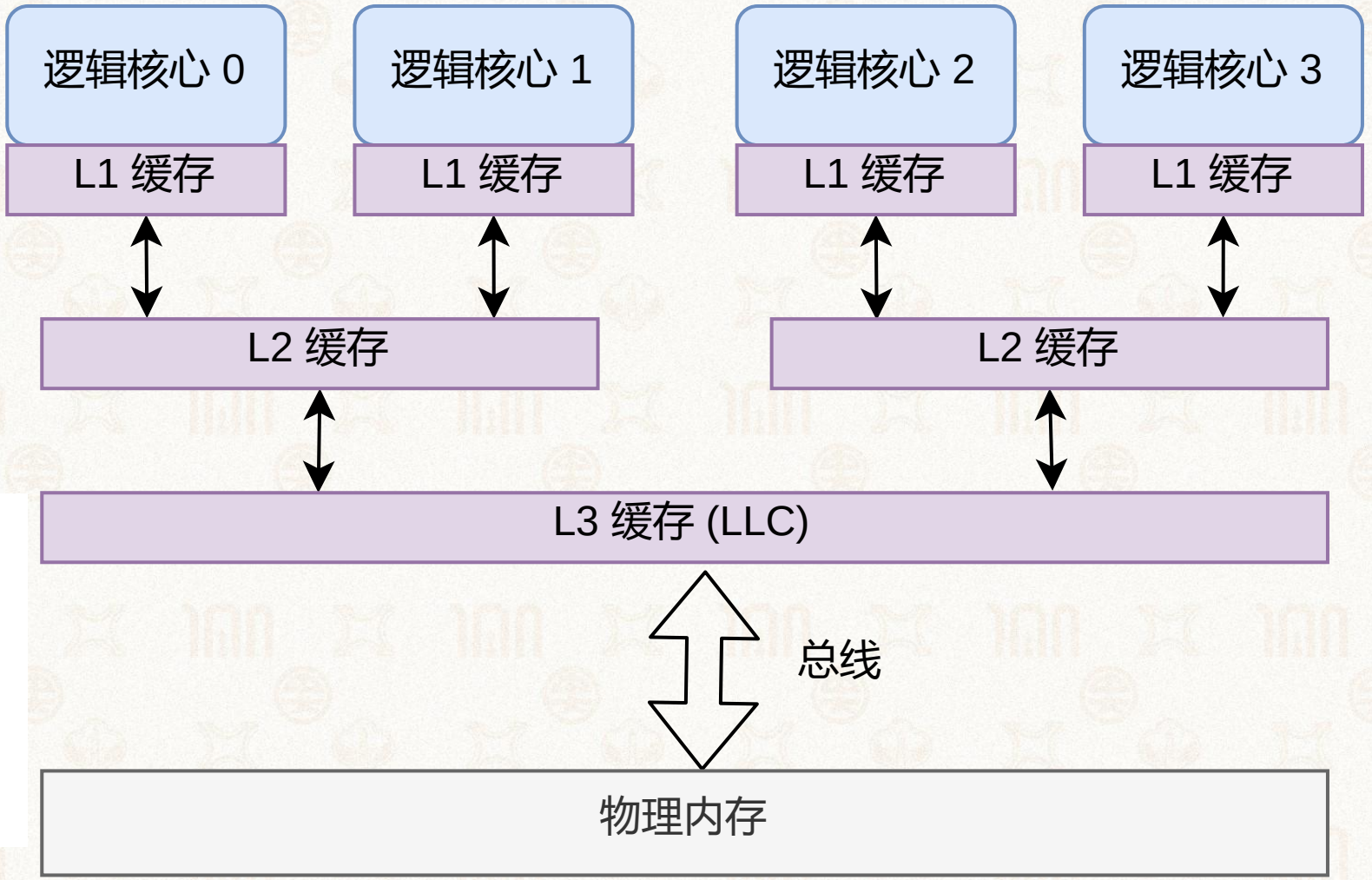
- 多级缓存：
  - 每个核心有自己的私有高速缓存（L1 Cache）
  - 多个核心共享一个二级高速缓存（L2 Cache）
  - 所有核心共享一个最末级高速缓存（Last Level Cache, LLC）



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（Last Level Cache, LLC）

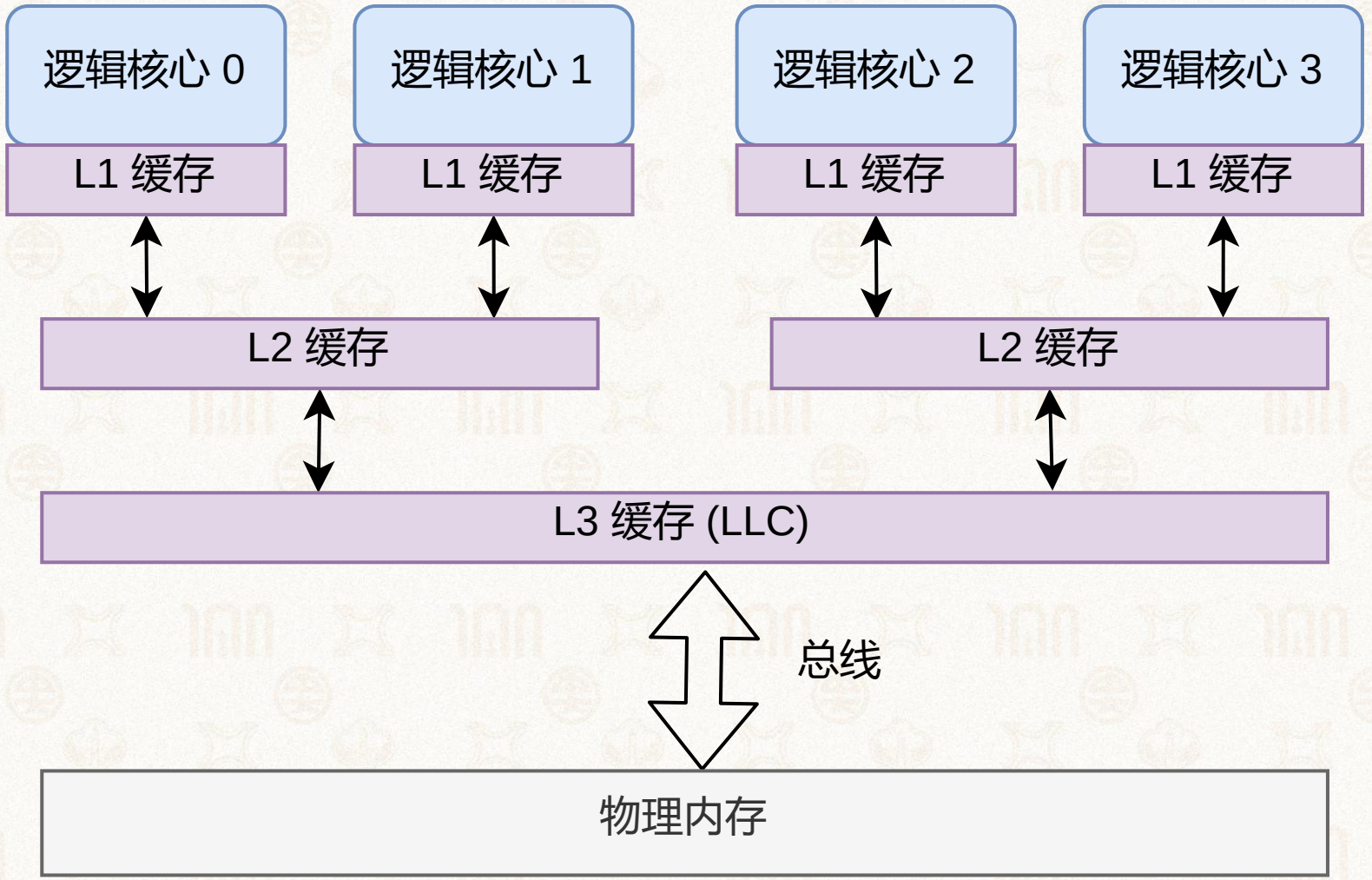
来自Windows系统的截图：



多核环境中的缓存结构

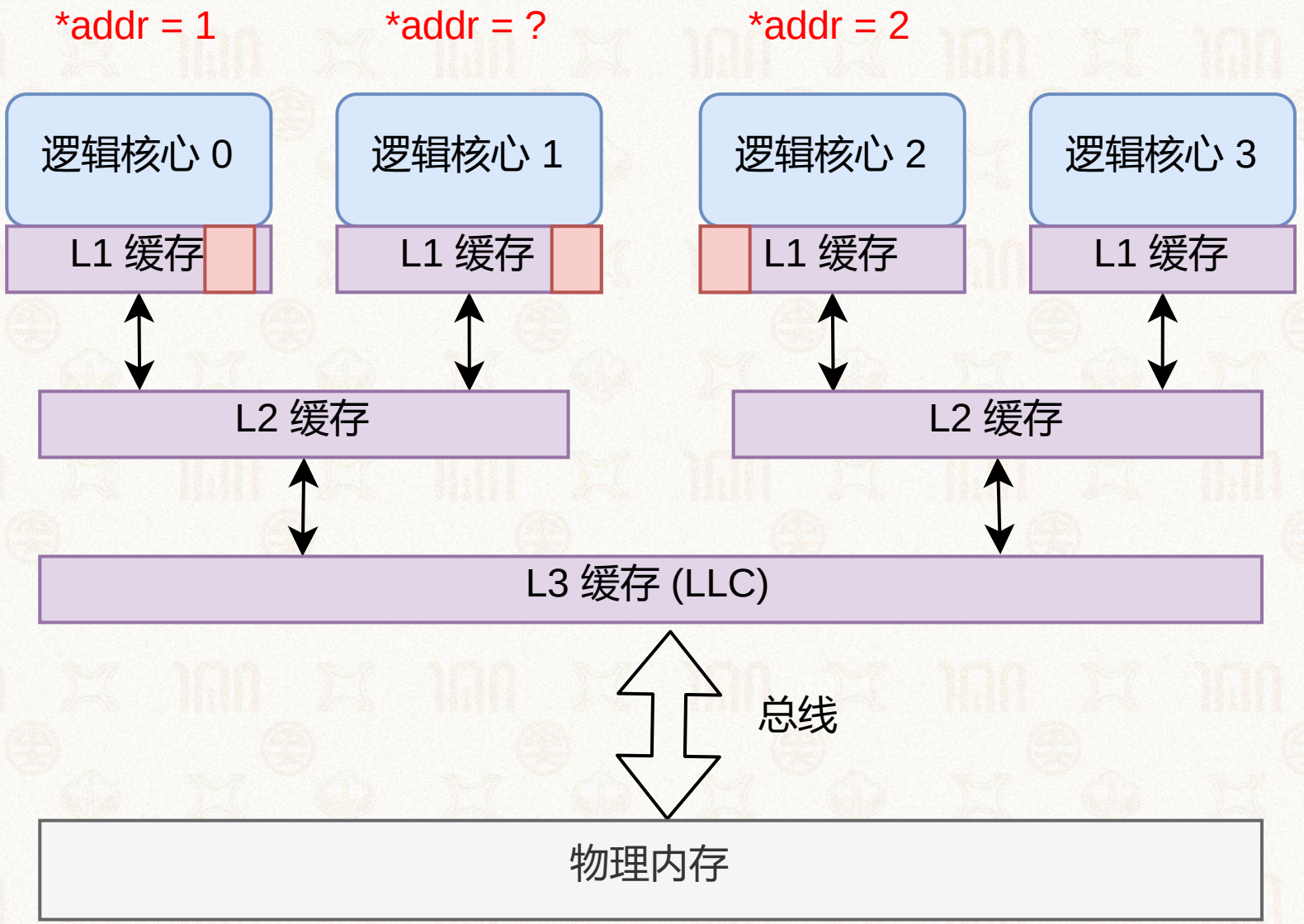
- 多级缓存：
  - 每个核心有自己的私有高速缓存（L1 Cache）
  - 多个核心共享一个二级高速缓存（L2 Cache）
  - 所有核心共享一个最末级高速缓存（Last Level Cache, LLC）

- 非一致缓存访问（Non-Uniform Cache Access, NUCA）



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（Last Level Cache, LLC）
- 非一致缓存访问（Non-Uniform Cache Access, NUCA）
- 数据一致性问题

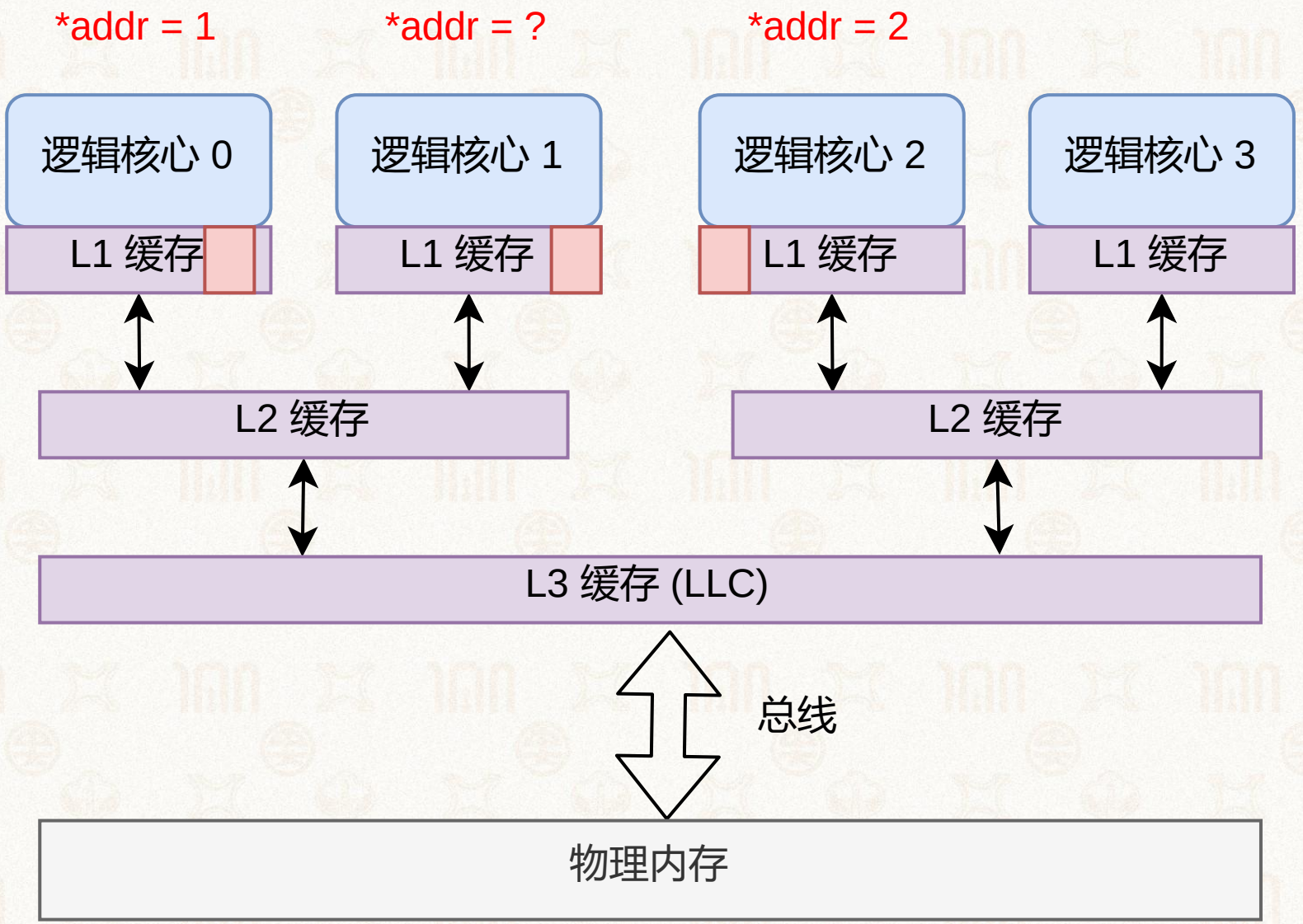


缓存一致性

- 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
  - 窥探式缓存一致性协议
  - 目录式缓存一致性协议

- 具体怎么做?
  - 缓存行处于不同状态
  - 不同状态之间迁移
  - 所有地读/写缓存行操作遵循协议流程

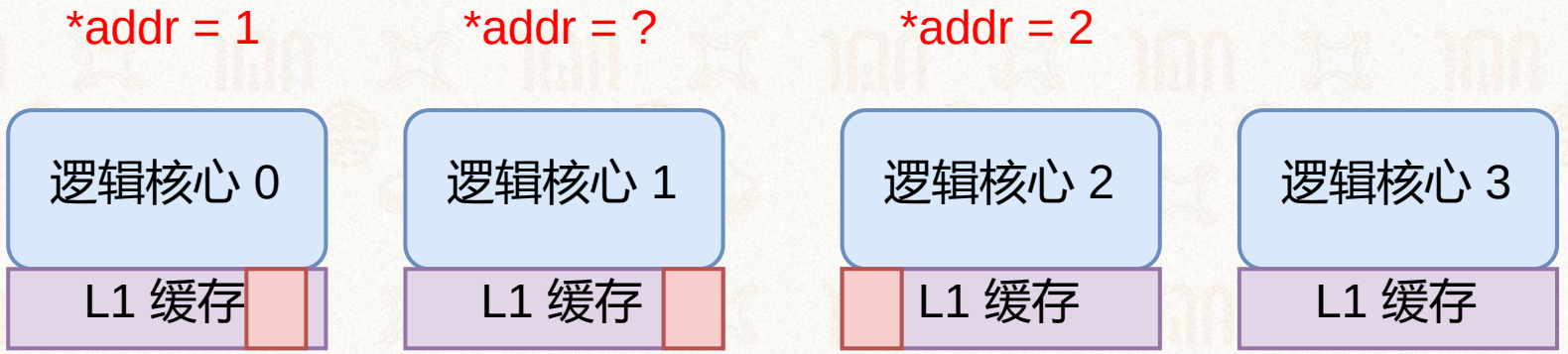


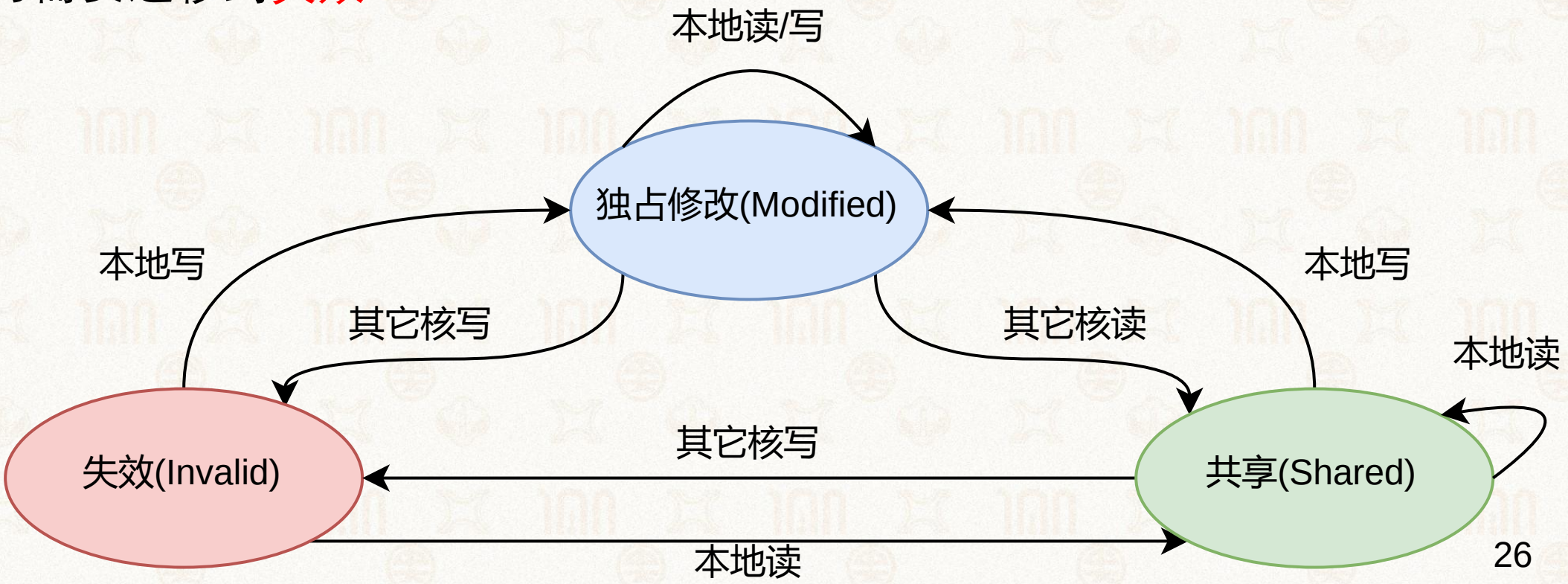
大纲

- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

缓存一致性：MSI状态迁移

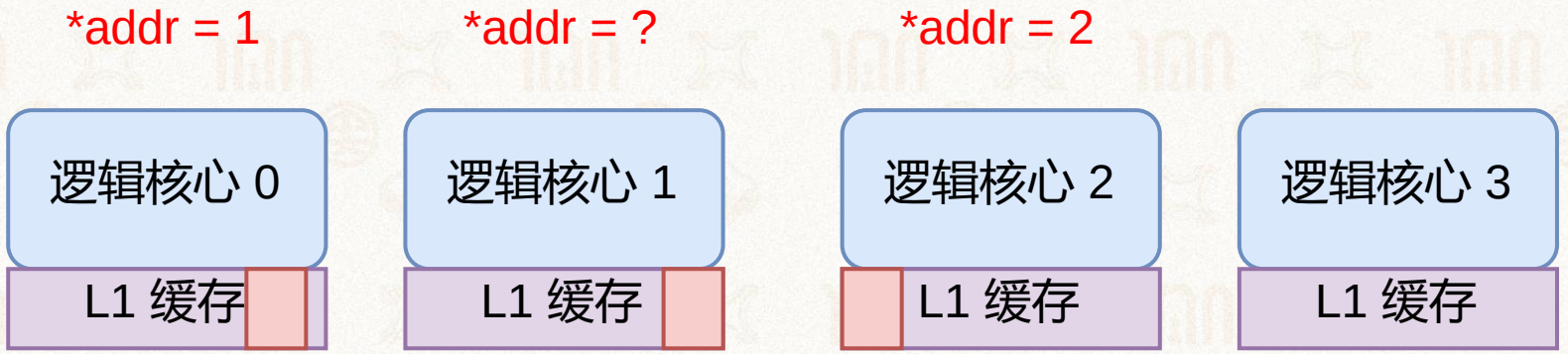
- 独占修改
  - 该核心独占拥有缓存行
  - 本地可**读**可**写**
  - 其他核**读**需要迁移到共享
  - 其他核**写**需要迁移到失效

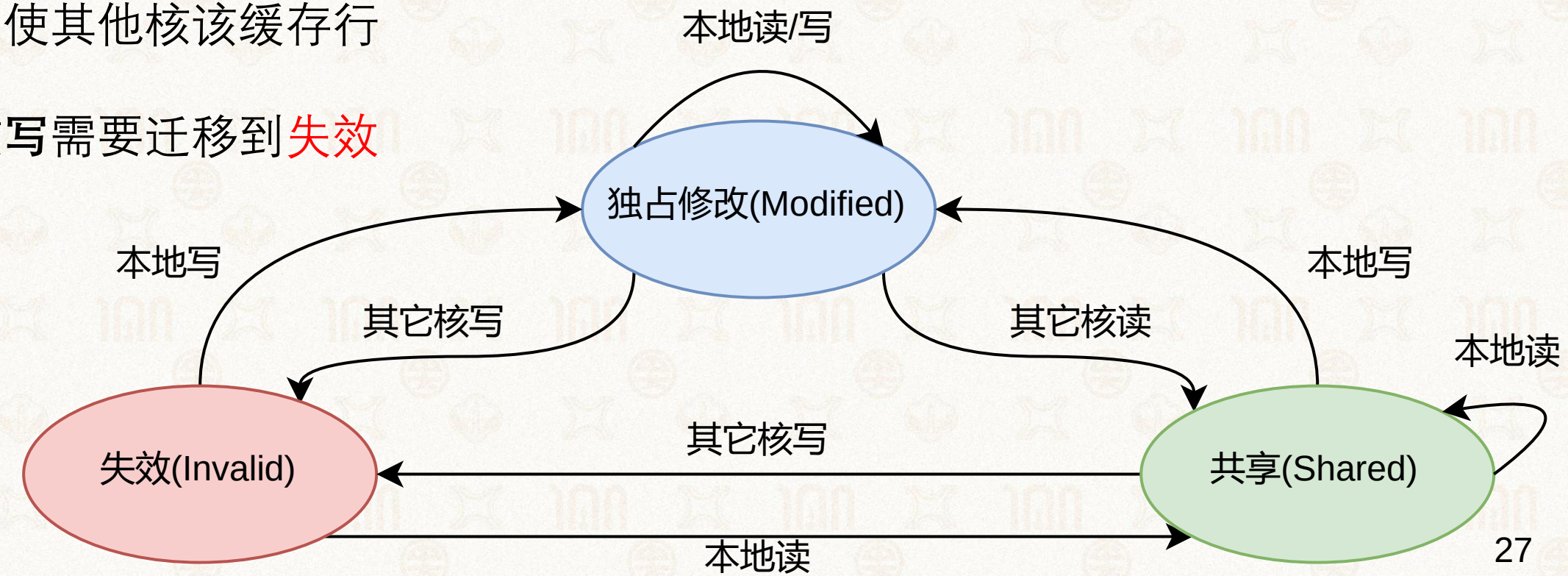




缓存一致性：MSI状态迁移

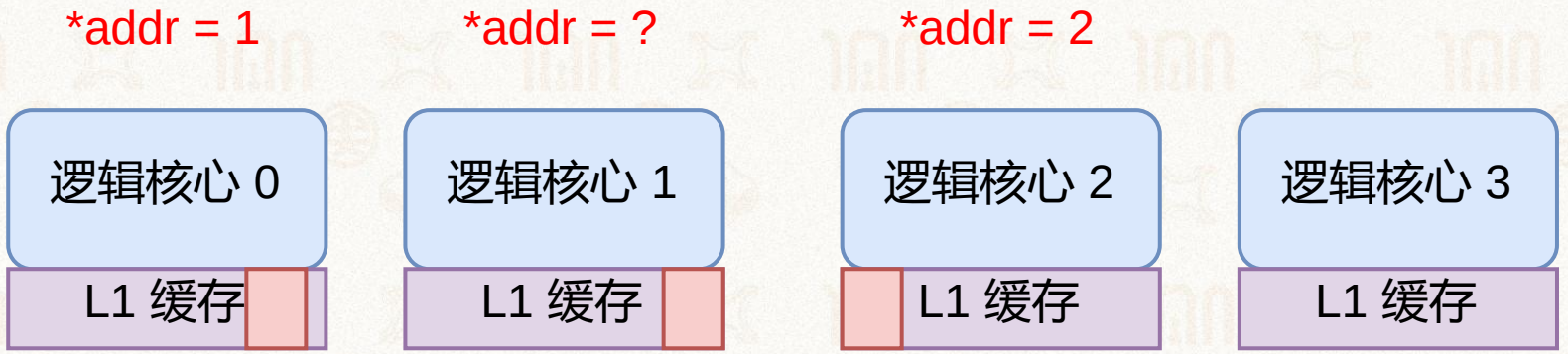
- 共享
  - 可能多个核同时有缓存行的拷贝
  - 本地可**读**
  - 本地**写**需要迁移到独占修改，并使其他核该缓存行失效
  - 其他核**写**需要迁移到失效

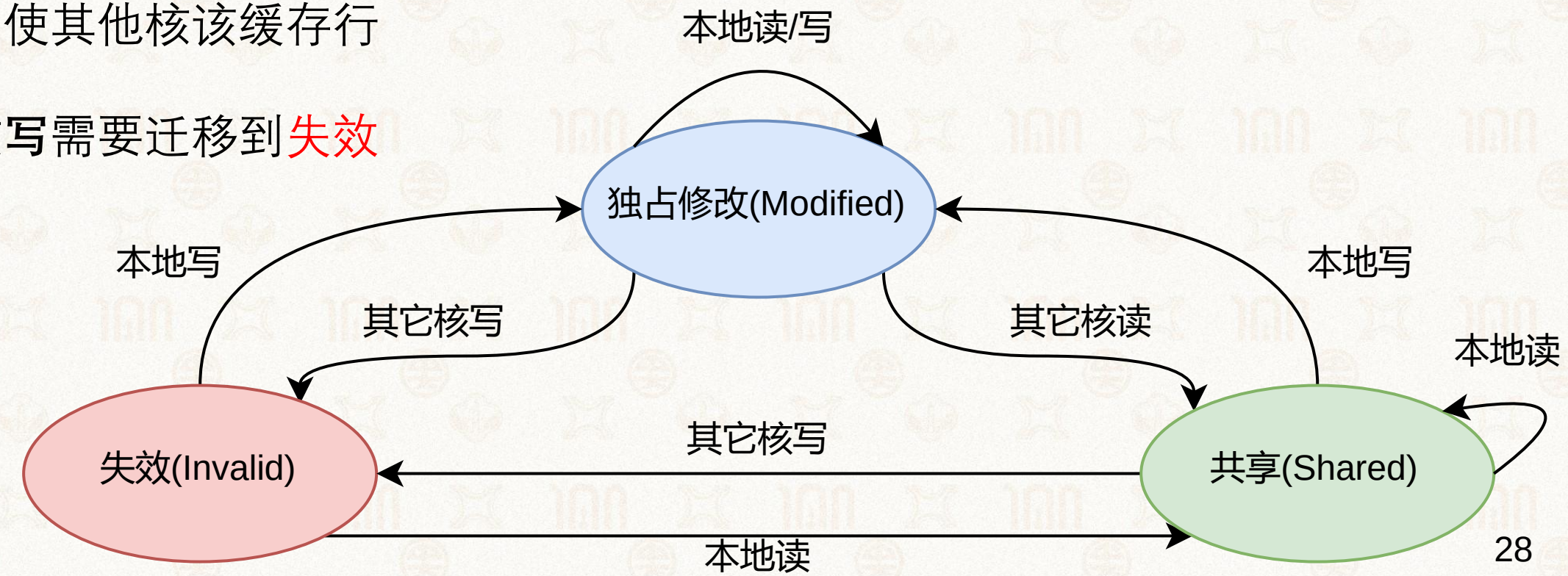




缓存一致性：MSI状态迁移

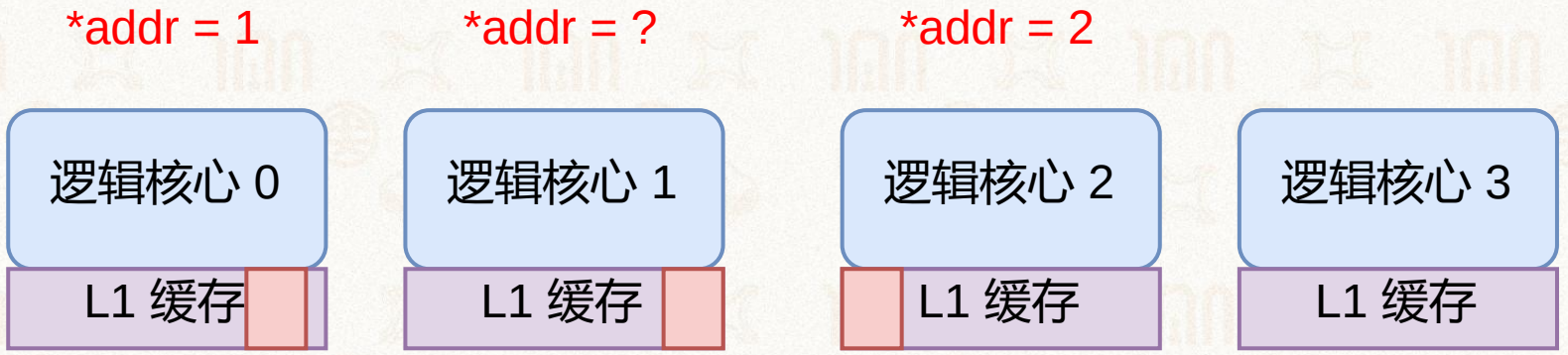
- 共享
  - 可能多个核同时有缓存行的拷贝
  - 本地可**读**
  - 本地**写**需要迁移到独占修改，并使其他核该缓存行失效
  - 其他核**写**需要迁移到失效

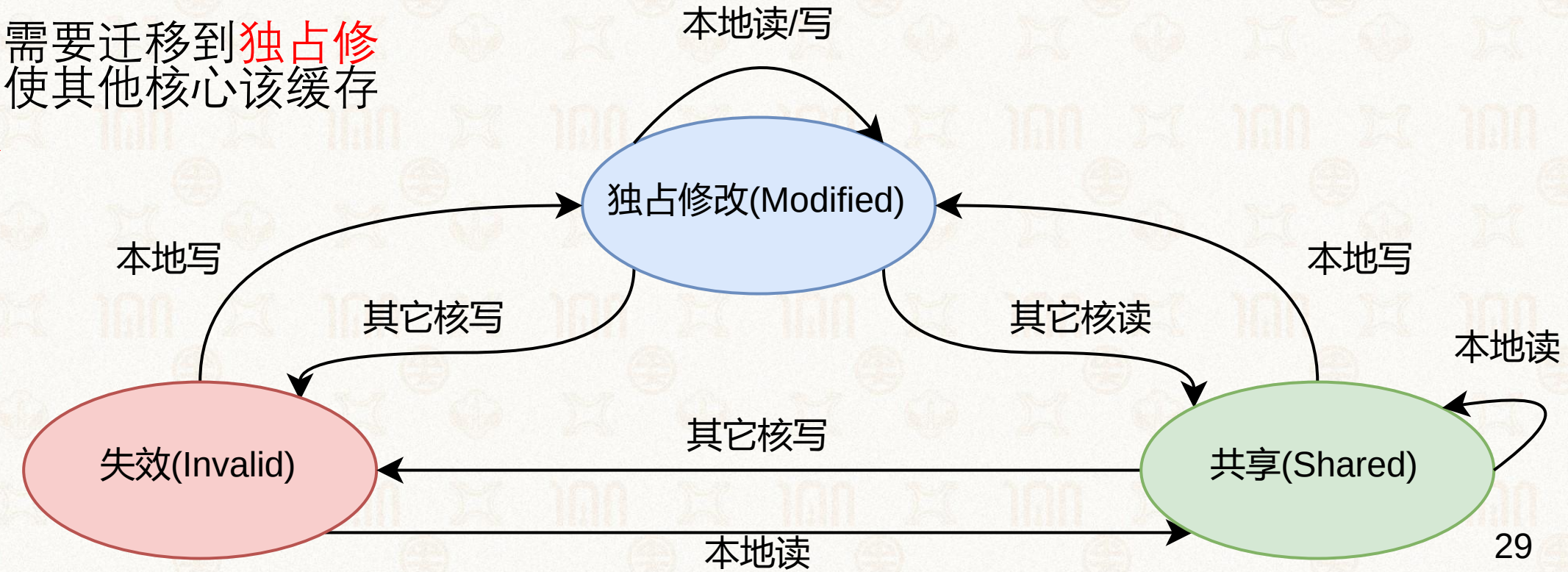




缓存一致性：MSI状态迁移

- 失效
  - 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





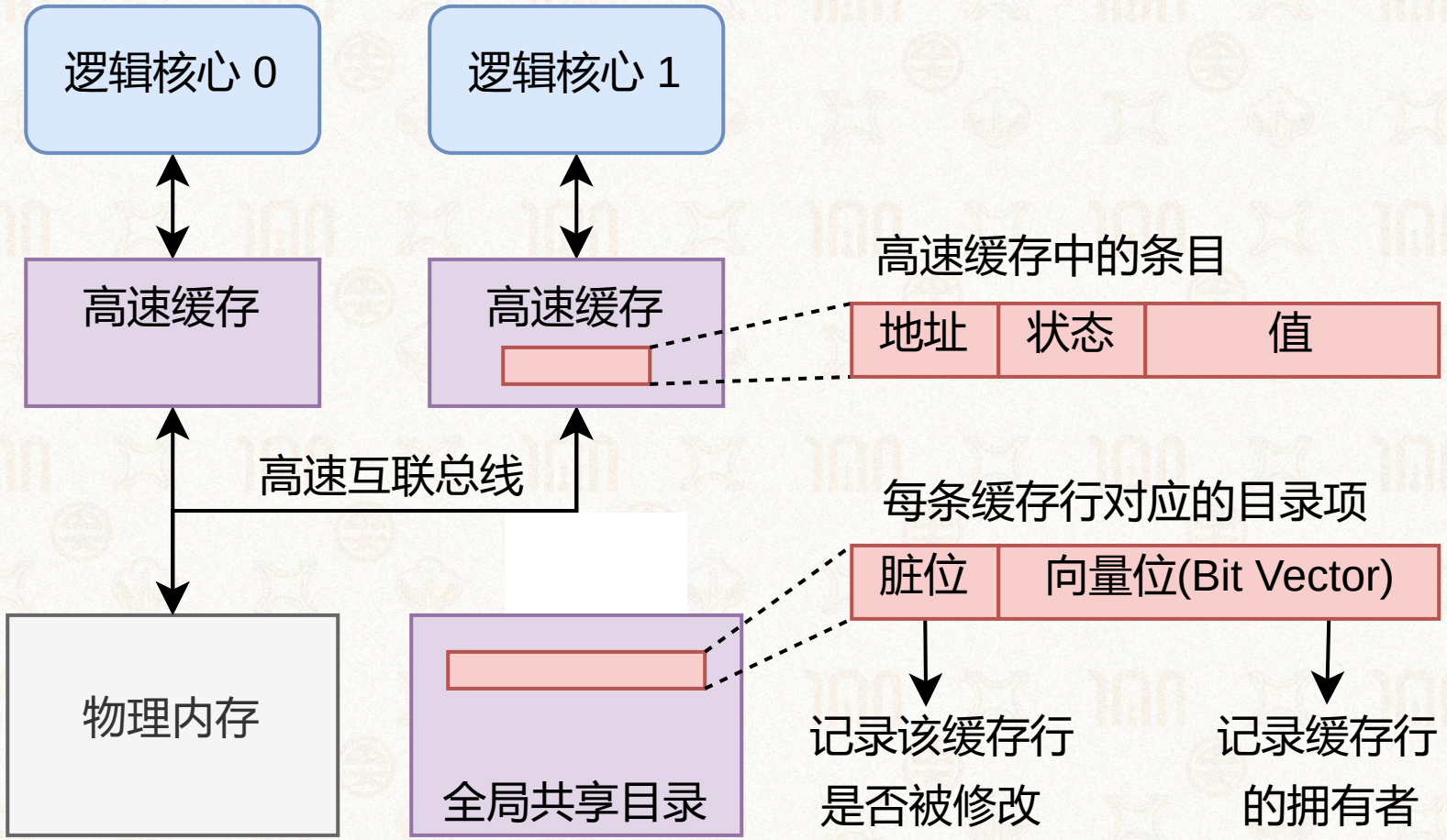
大纲

- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

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：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

CPU 0

	状态	内容
X:	S	666

CPU 1

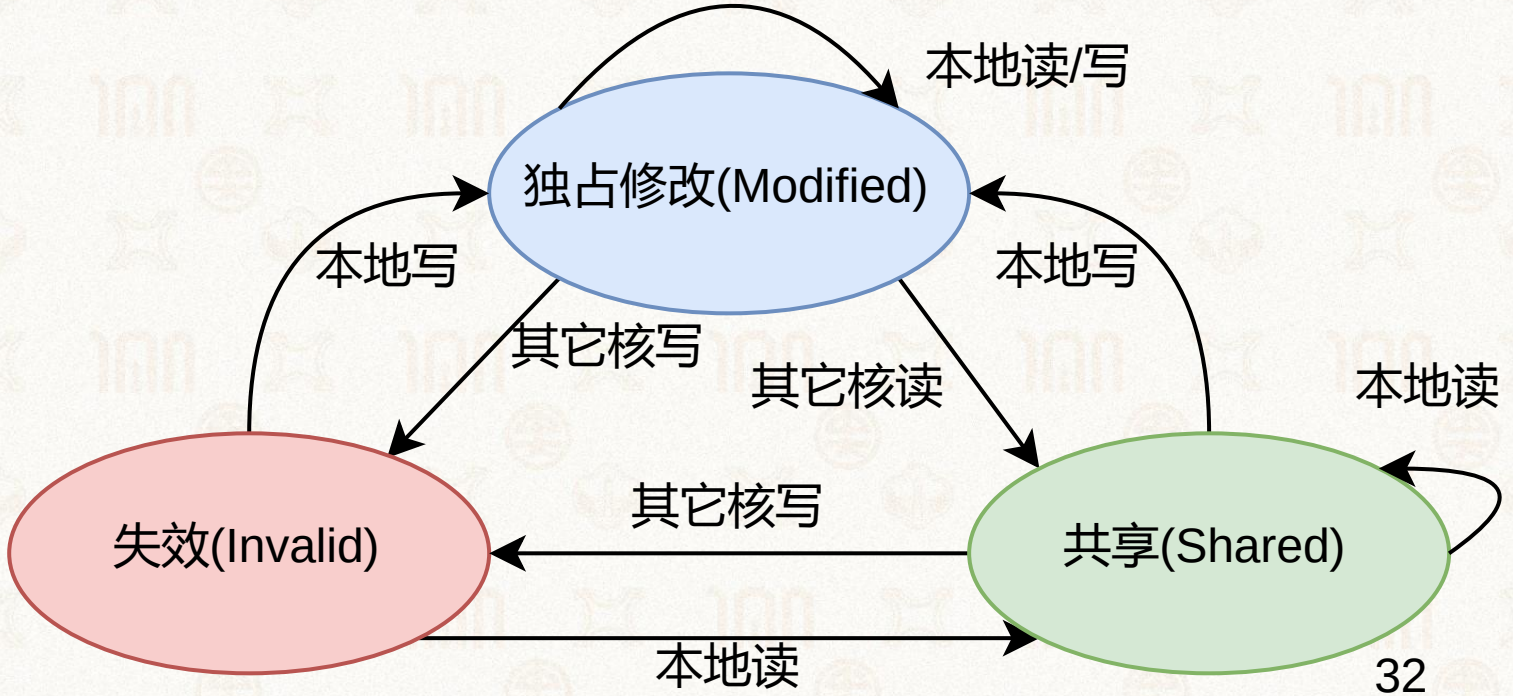
状态	内容
S	666

CPU 2

状态	内容
S	666

全局共享目录项

	脏位	向量位		
X:	0	1	1	1



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

执行 STR X, 233

CPU 0

	状态	内容
X:	S	666

CPU 1

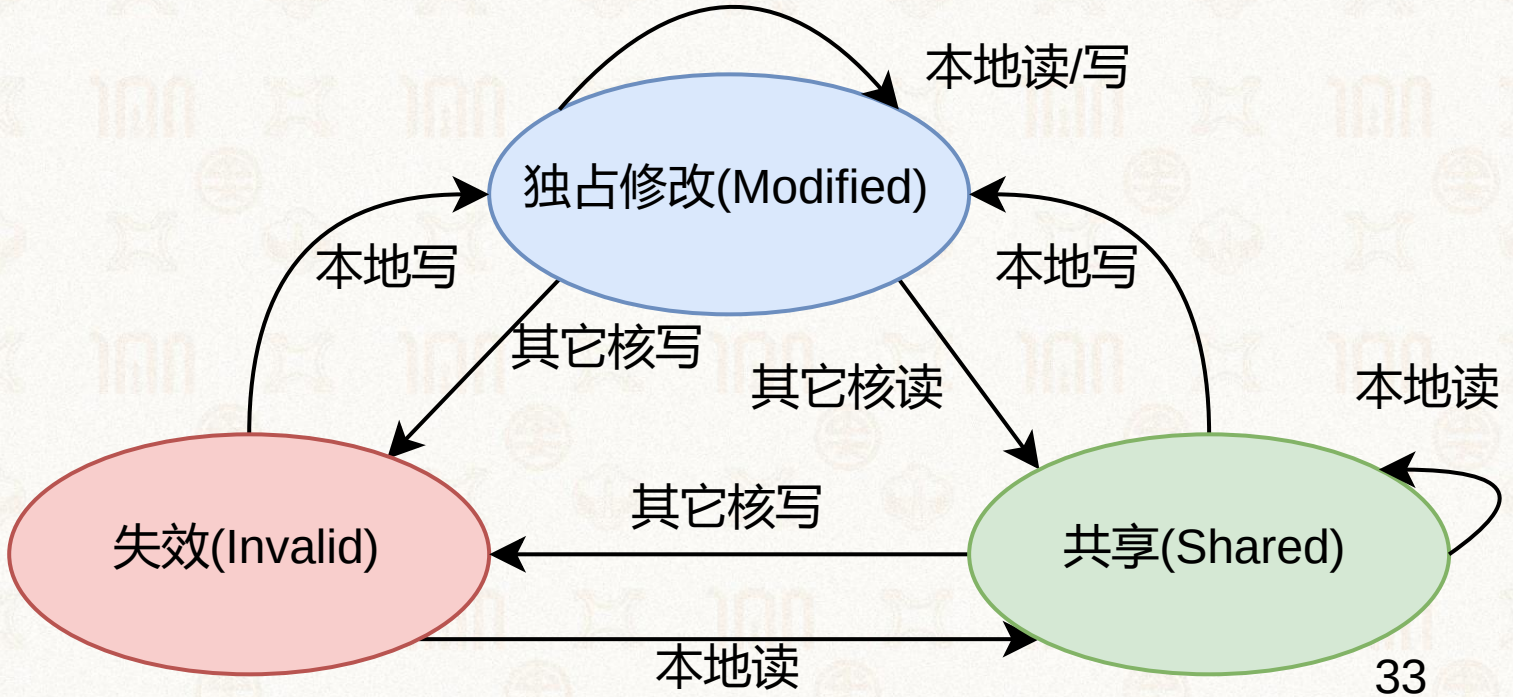
状态	内容
S	666

CPU 2

状态	内容
S	666

全局共享目录项

	脏位	向量位		
X:	0	1	1	1



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

执行 STR X, 233

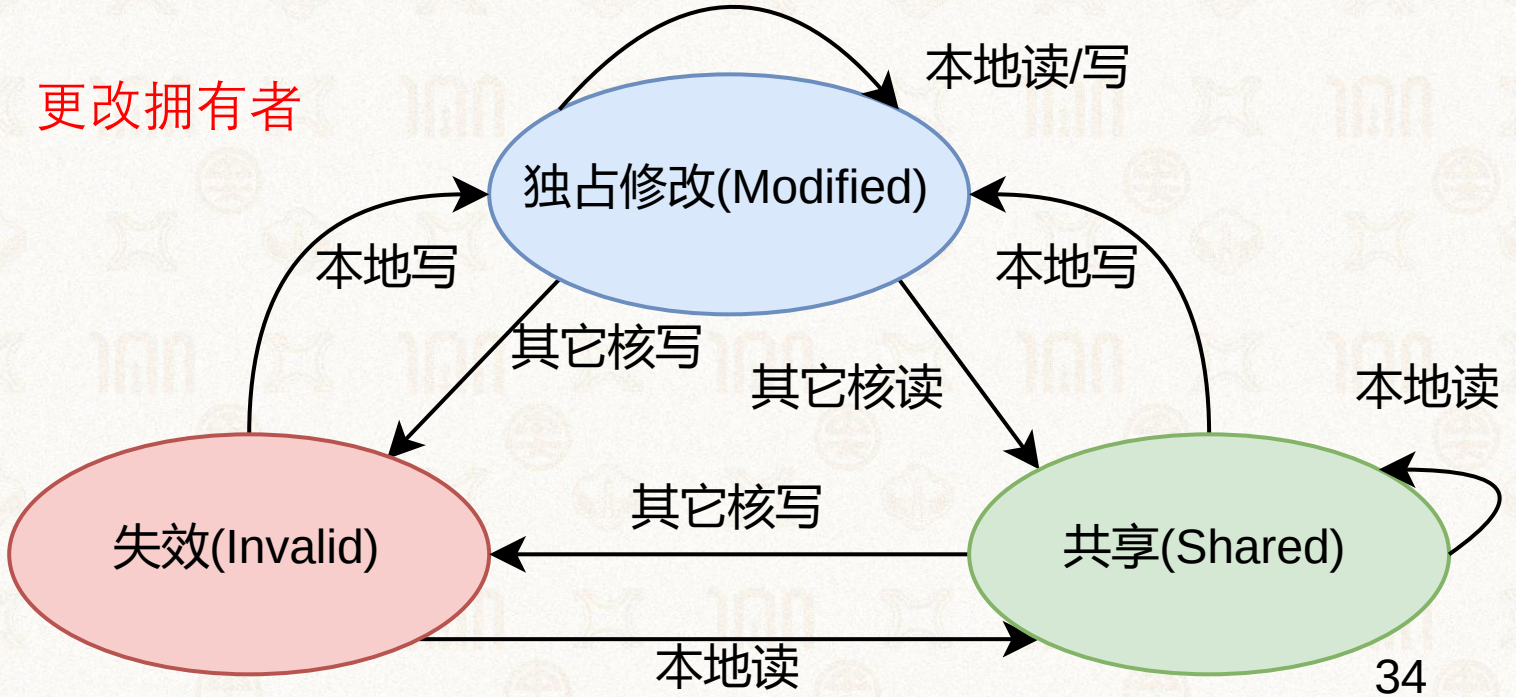
	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
X:	S	666	S	666	S	666

查看目录项，设置脏位，更改拥有者

全局共享目录项

	脏位	向量位		
X:	1	1	0	0

向量位：1表示对应CPU拥有该缓存行



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

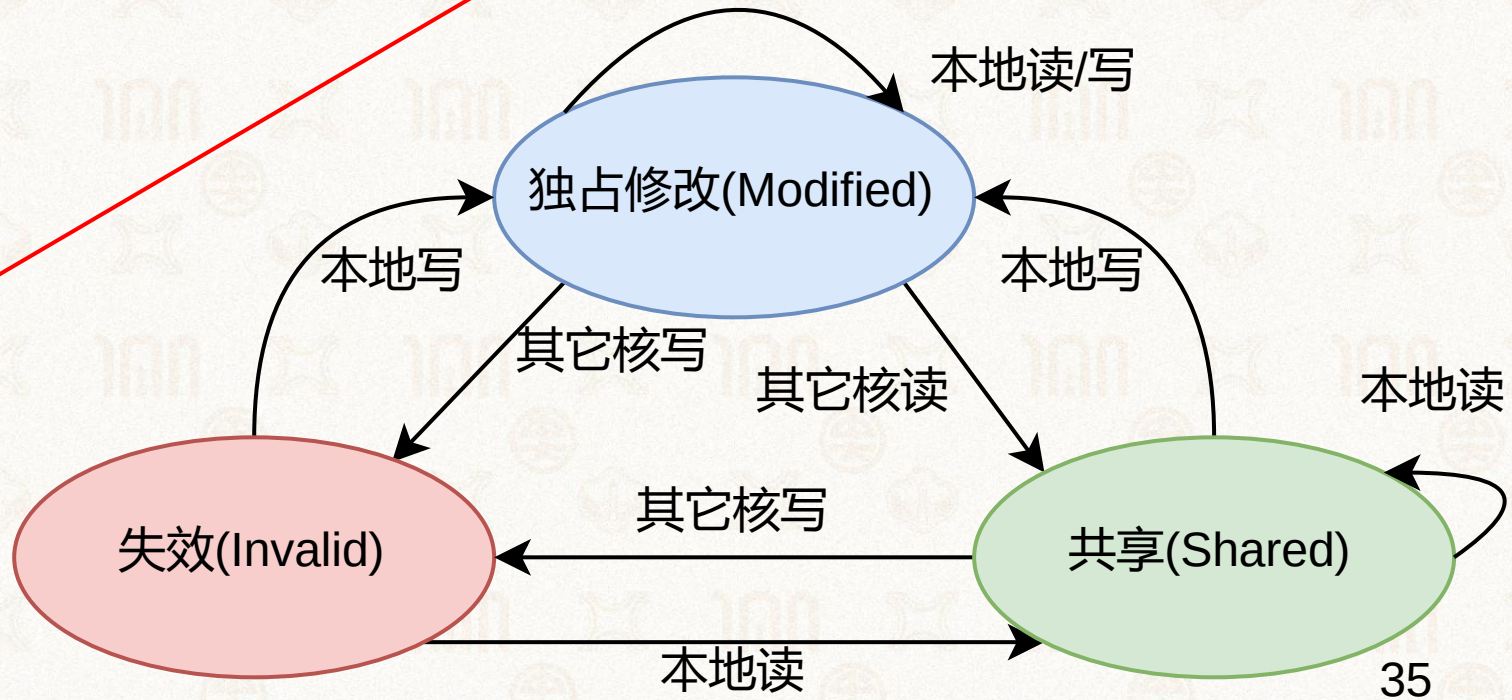
执行 STR X, 233

	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

	脏位	向量位		
X:	1	1	0	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

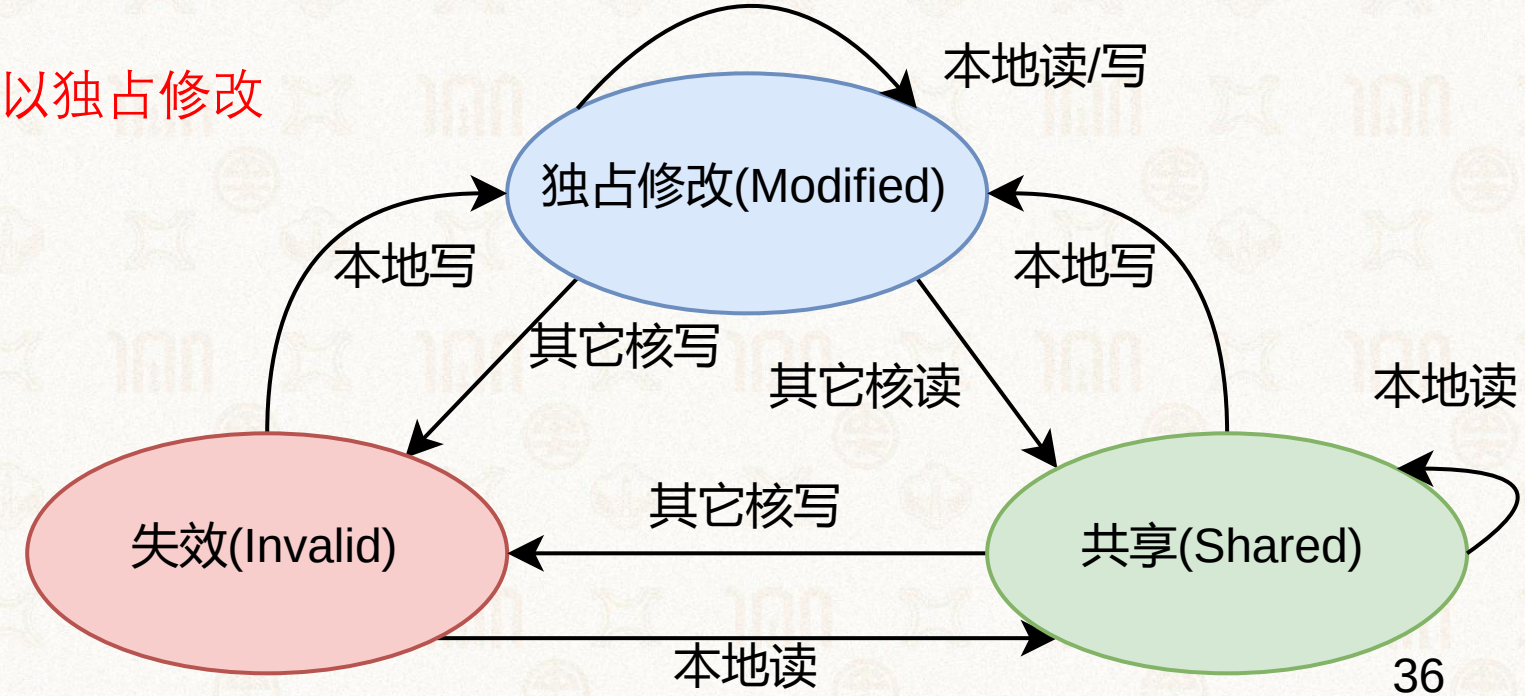
执行 STR X, 233

	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

	脏位	向量位		
X:	1	1	0	0



目录式缓存一致性：示例

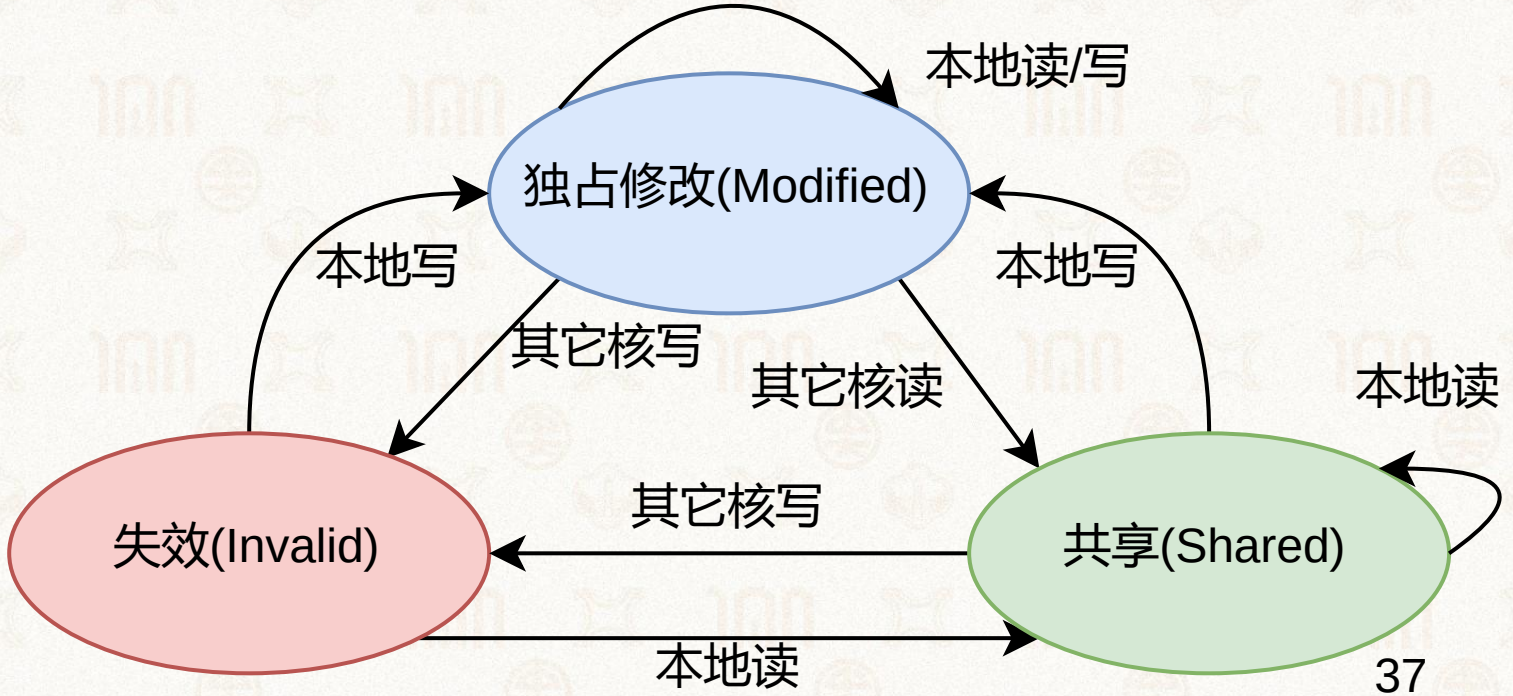
- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

执行 STR 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

全局共享目录项

	脏位	向量位		
X:	1	1	0	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

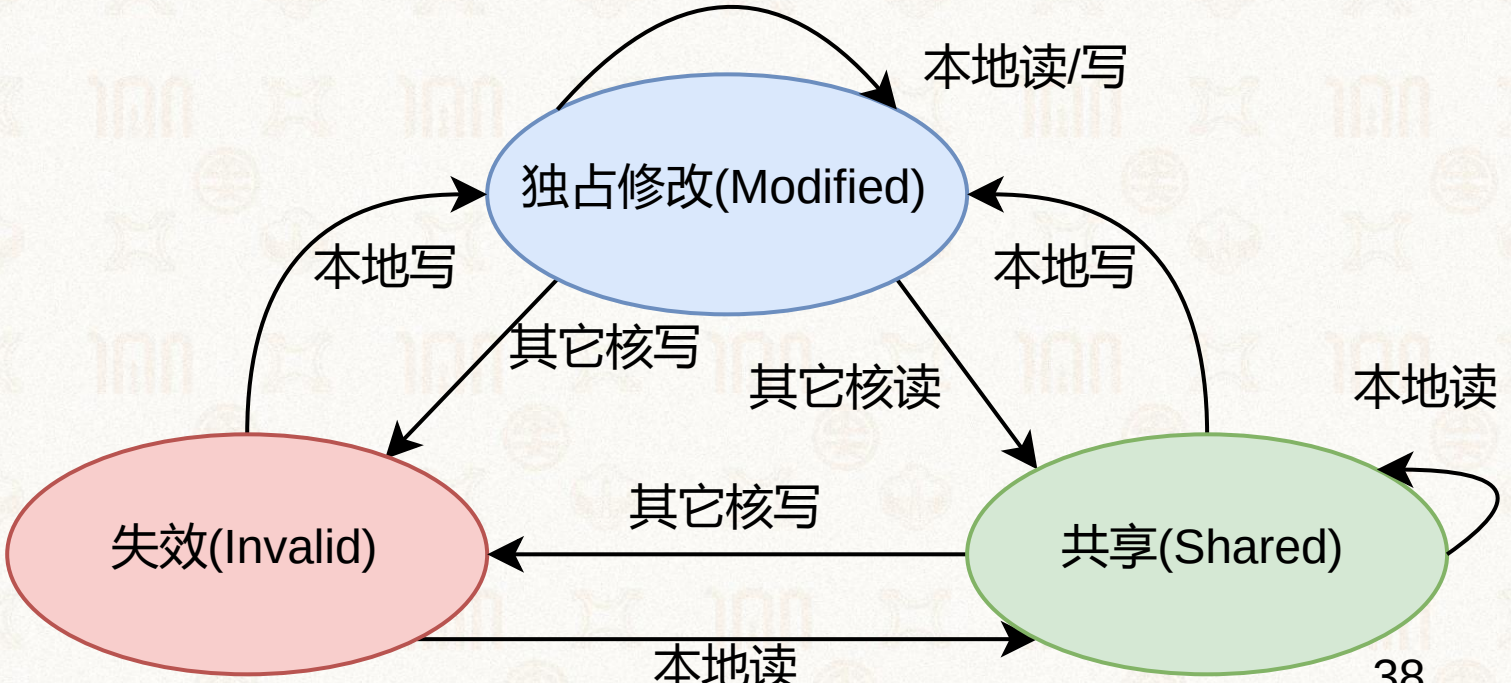
执行 STR X, 888

	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，修改脏位，更改拥有者

全局共享目录项

	脏位	向量位		
X:	1	0	1	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

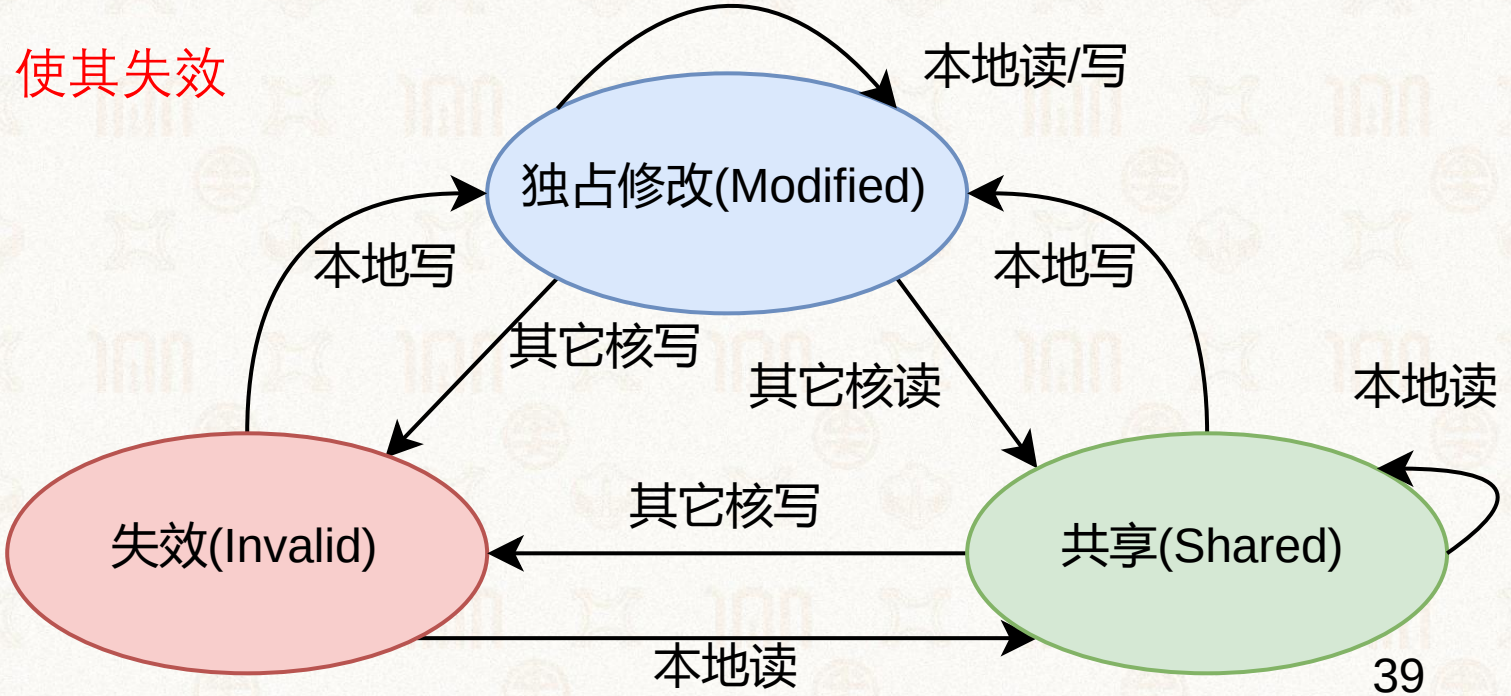
执行 STR X, 888

	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

	脏位	向量位		
X:	1	0	1	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

执行 STR X, 888

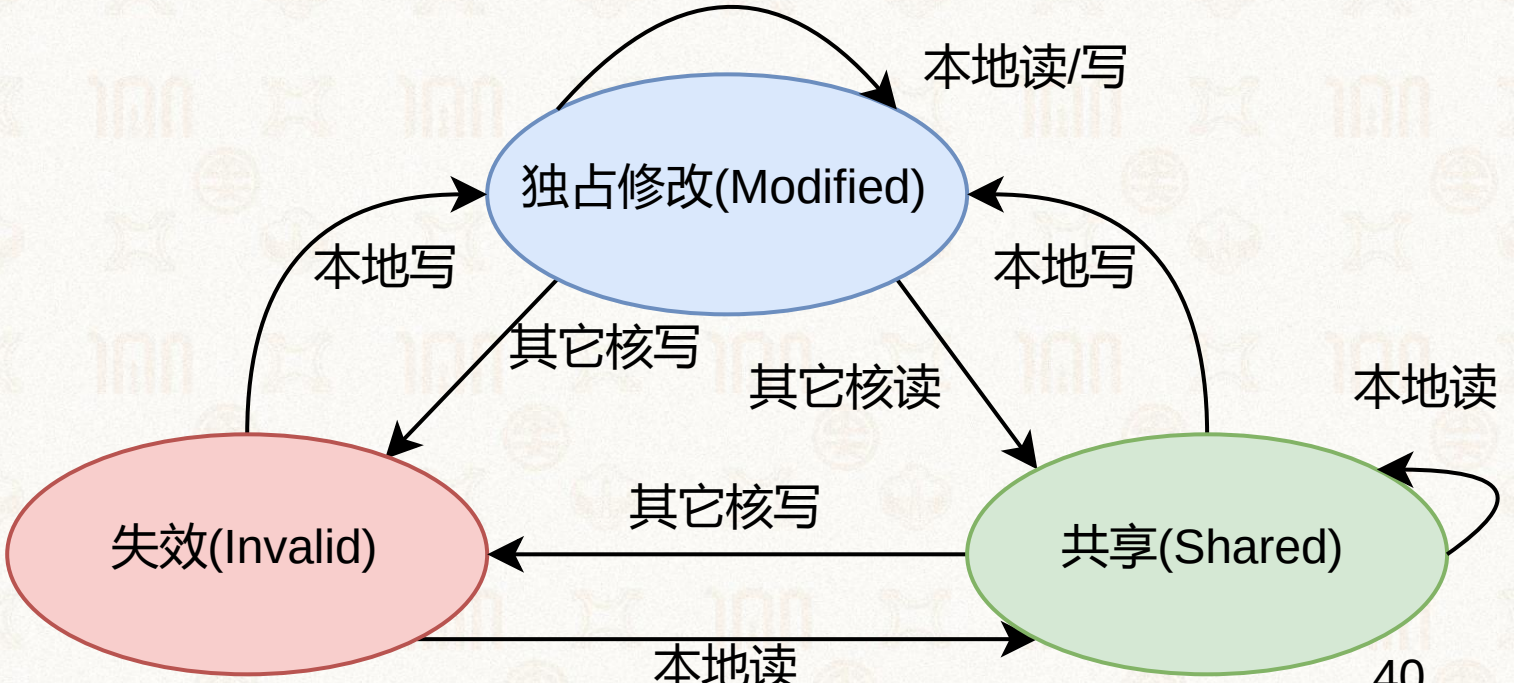
	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
X:	I	233	M	888	I	666

回复CPU1，可以独占修改

全局共享目录项

	脏位	向量位		
X:	1	0	1	0

这样233是不是就丢了？正确么？



目录式缓存一致性：示例

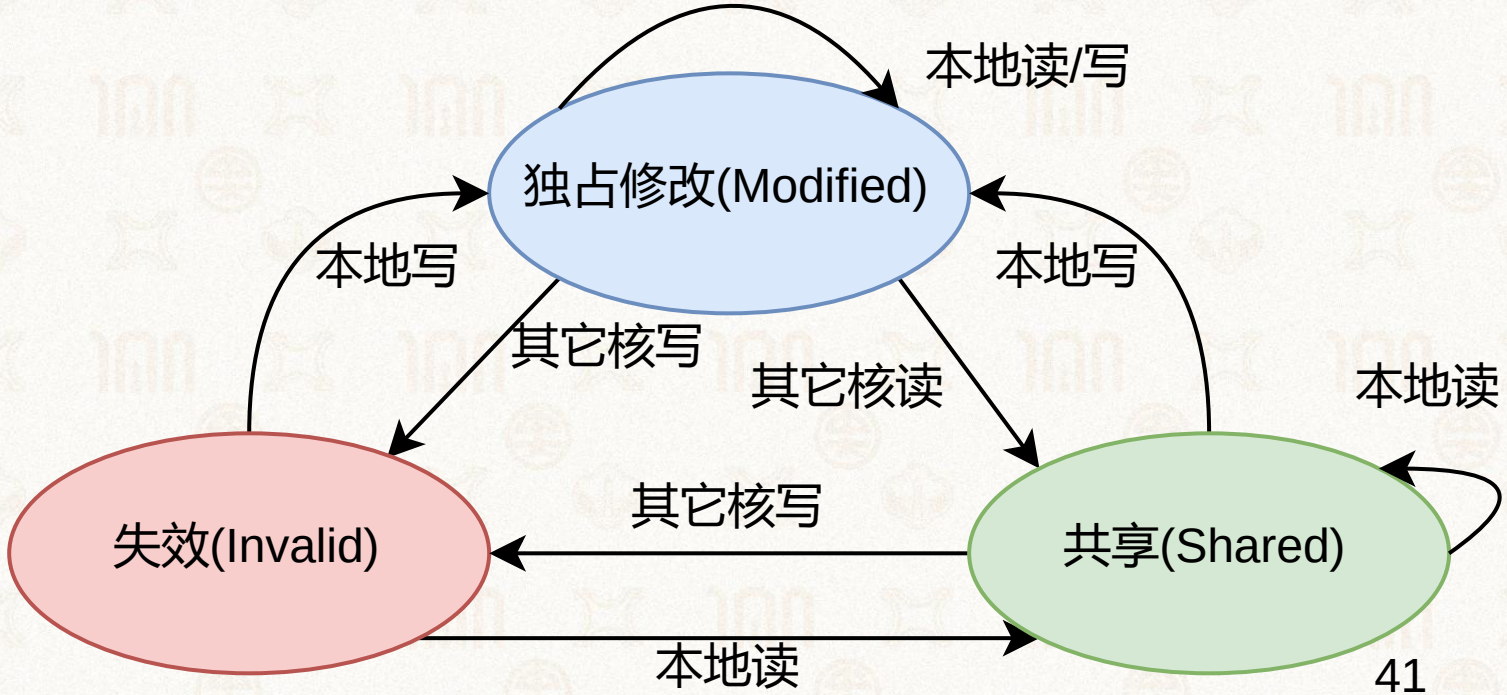
- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

执行 LDR 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

全局共享目录项

	脏位	向量位		
X:	1	0	1	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

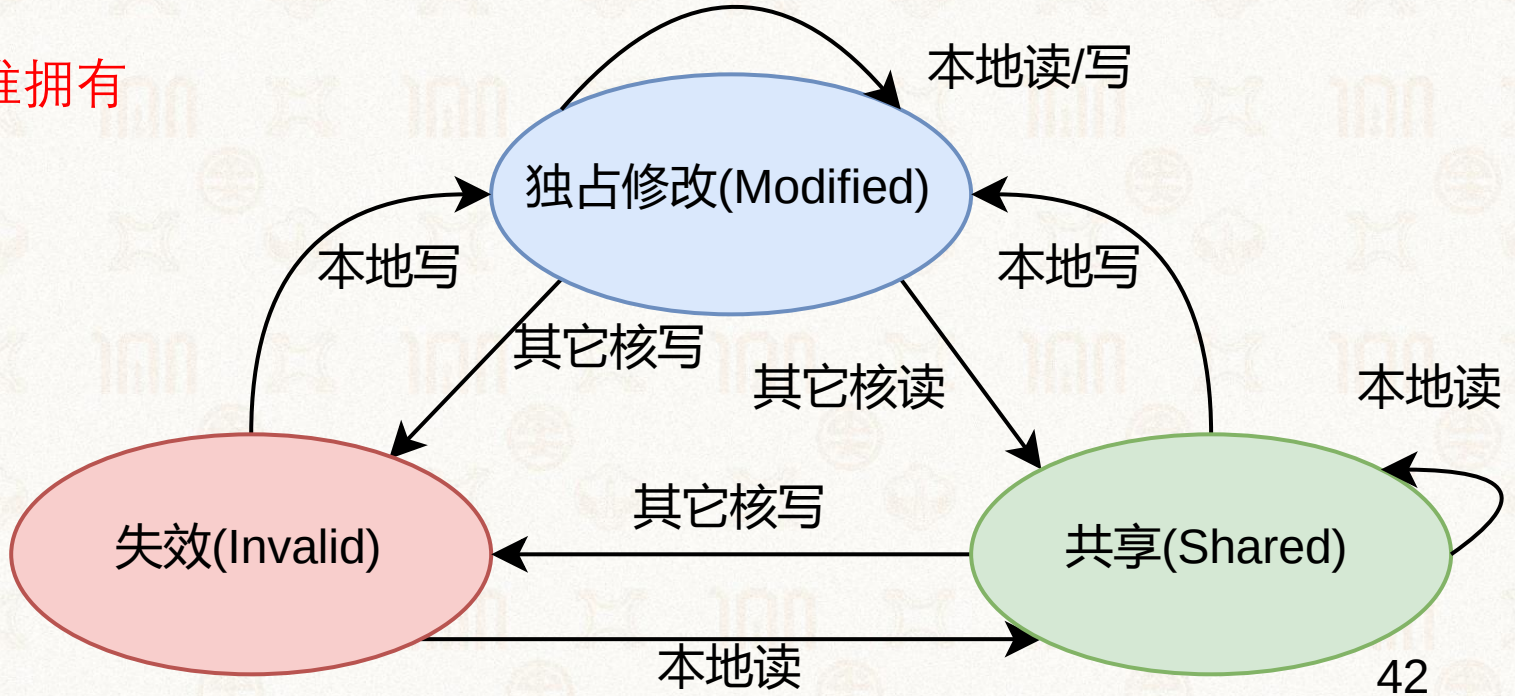
执行 LDR X

	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

	脏位	向量位		
X:	1	0	1	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

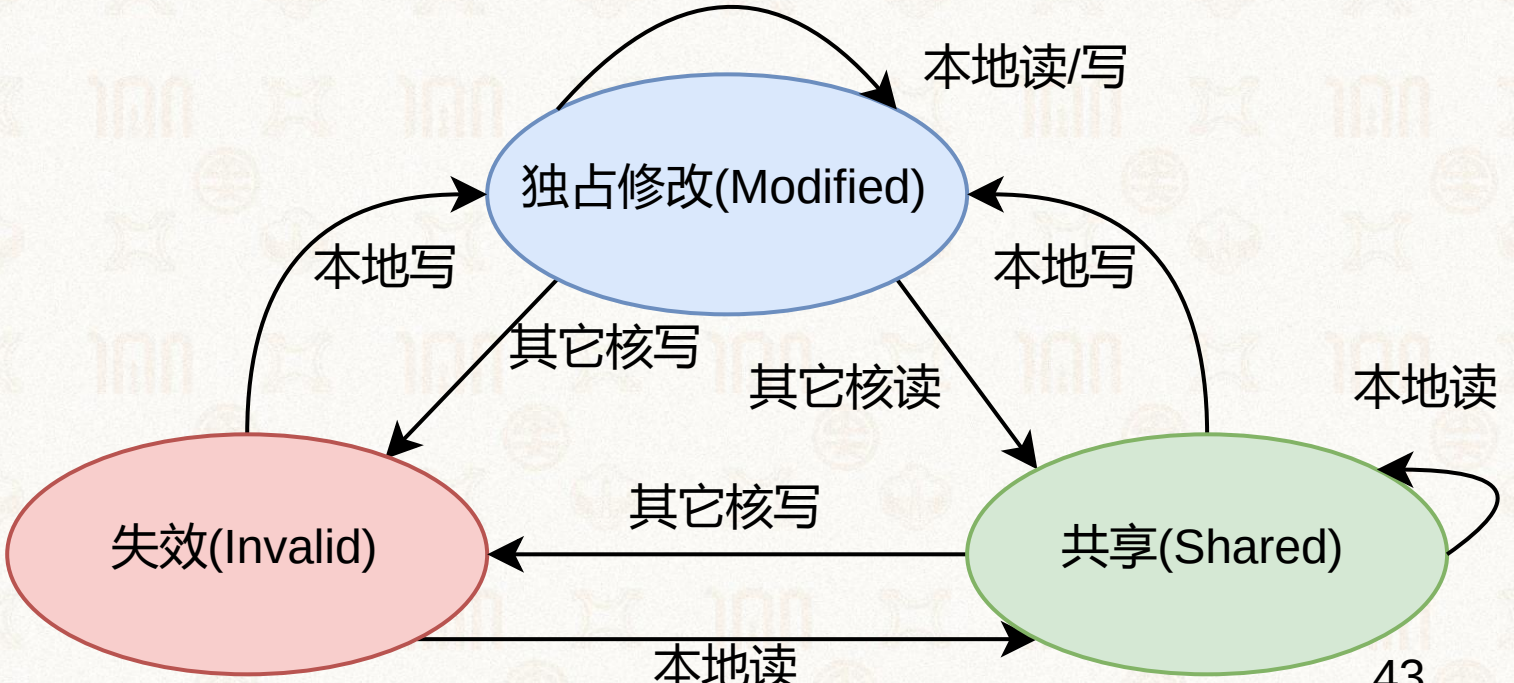
执行 LDR X

	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

	脏位	向量位		
X:	0	1	1	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

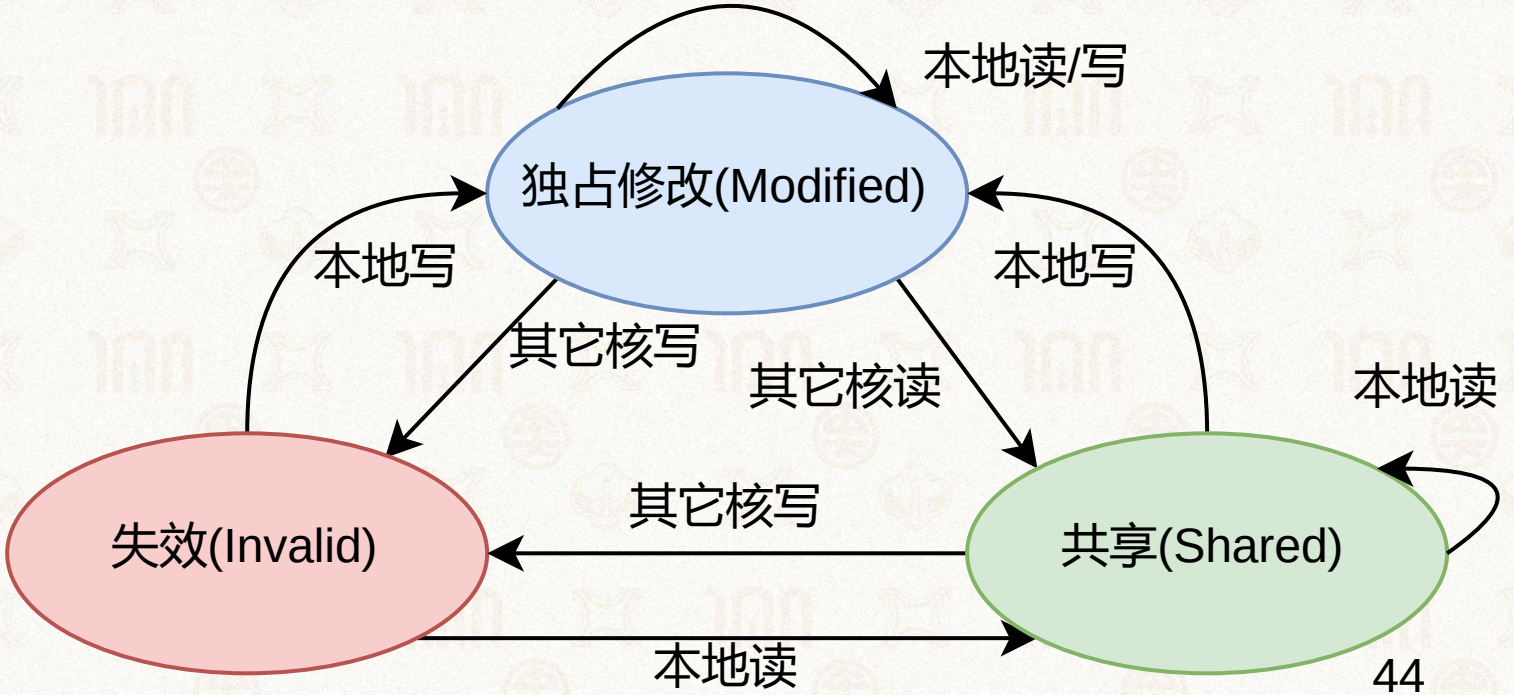
执行 LDR X

	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

	脏位	向量位		
X:	0	1	1	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

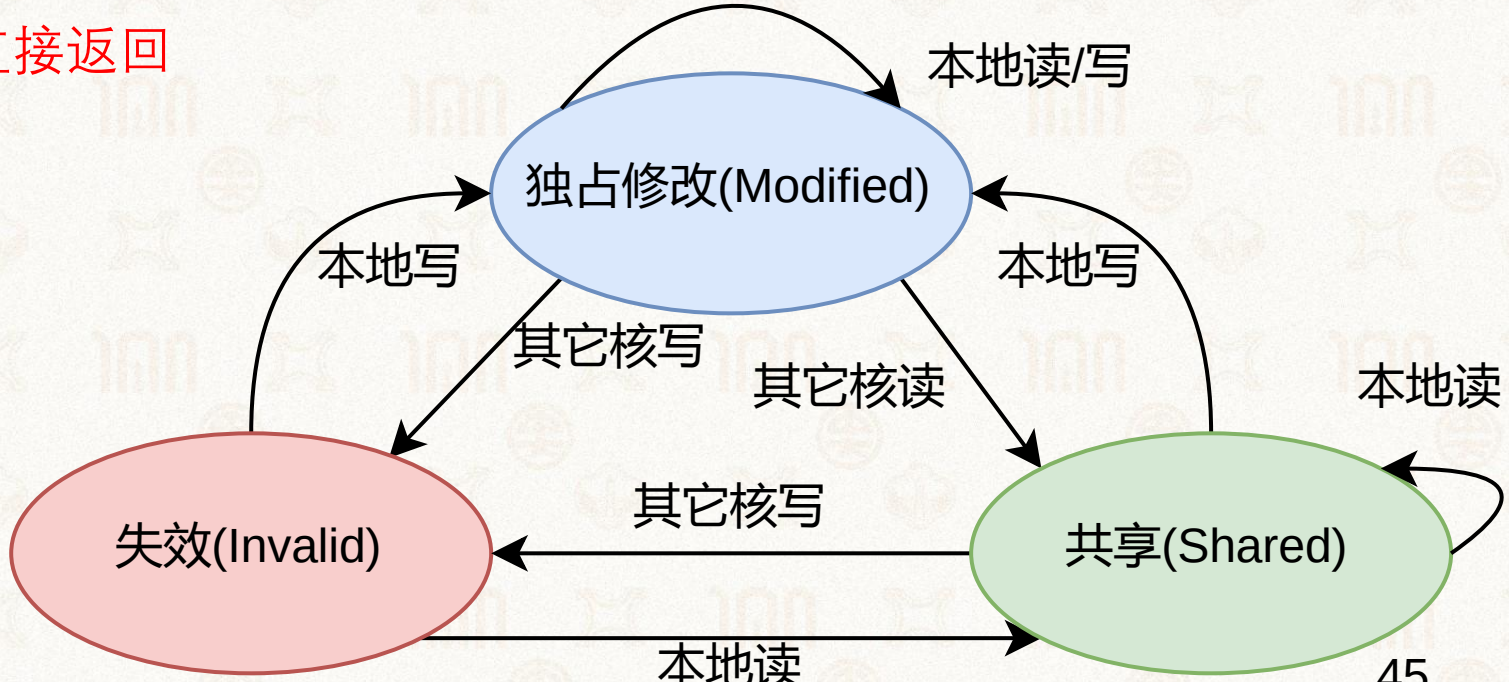
执行 LDR X

	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

	脏位	向量位		
X:	0	1	1	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

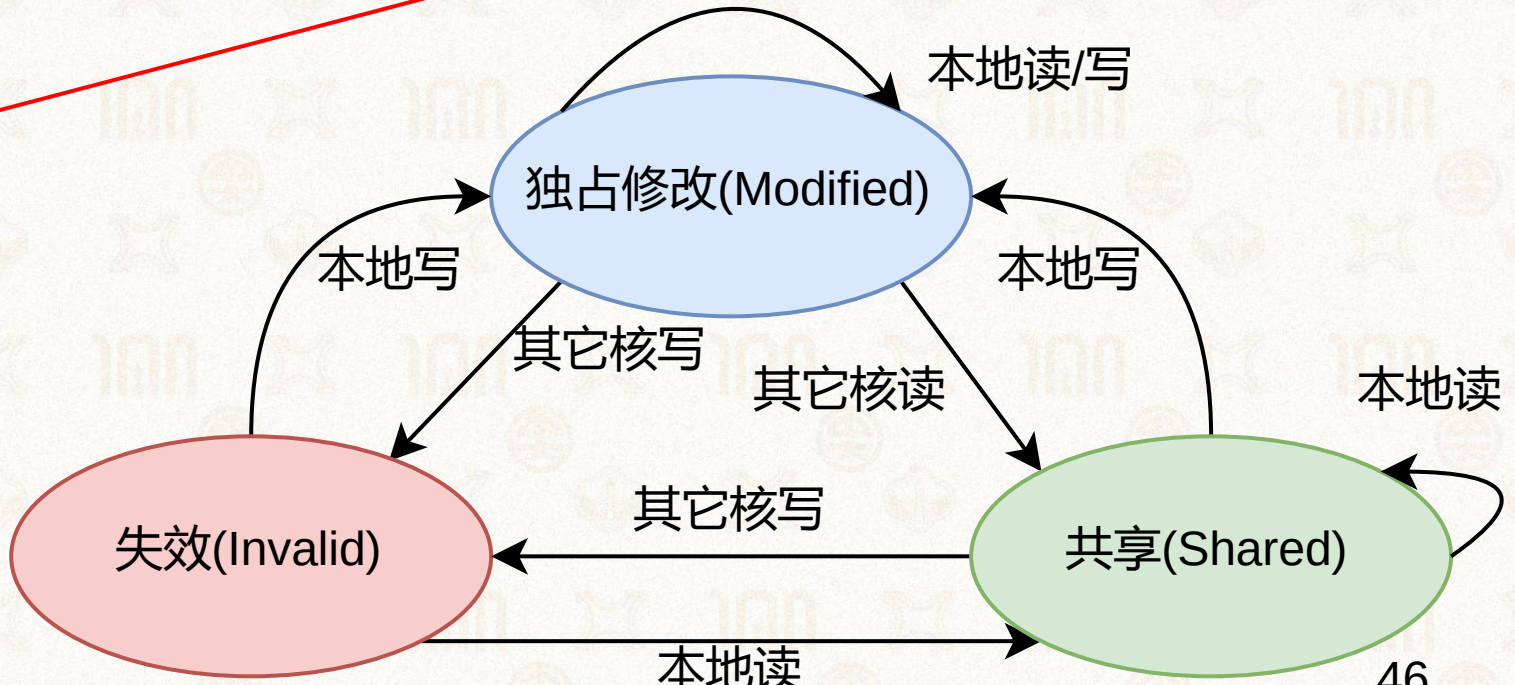
执行 LDR X

	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

发现失效，去目录找谁拥有

全局共享目录项

	脏位	向量位		
X:	0	1	1	0



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

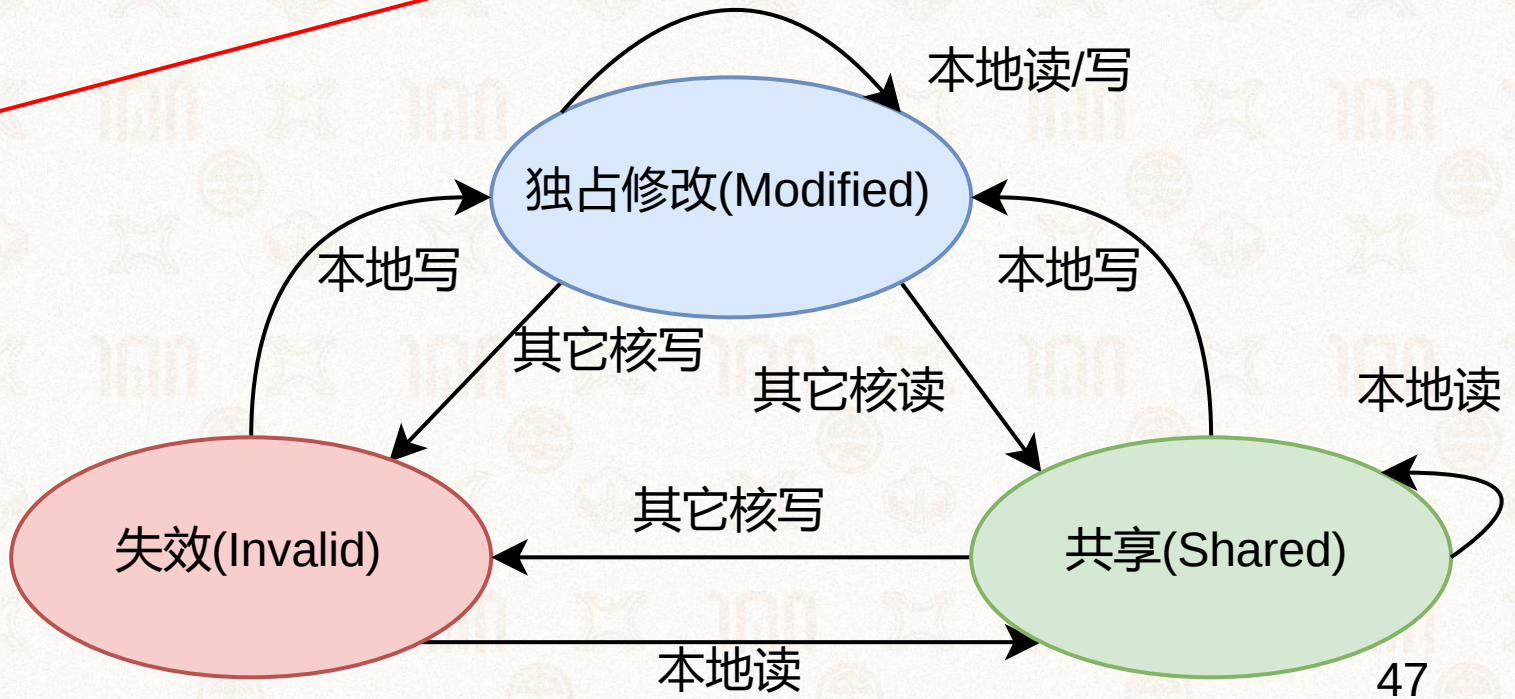
执行 LDR X

	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

更新目录，并让拥有者(cpu0或者cpu1)
给cpu2发送最新的值

全局共享目录项

	脏位	向量位		
X:	0	1	1	1



目录式缓存一致性：示例

- 只关注变量X所在缓存行，3个CPU，一个全局共享目录

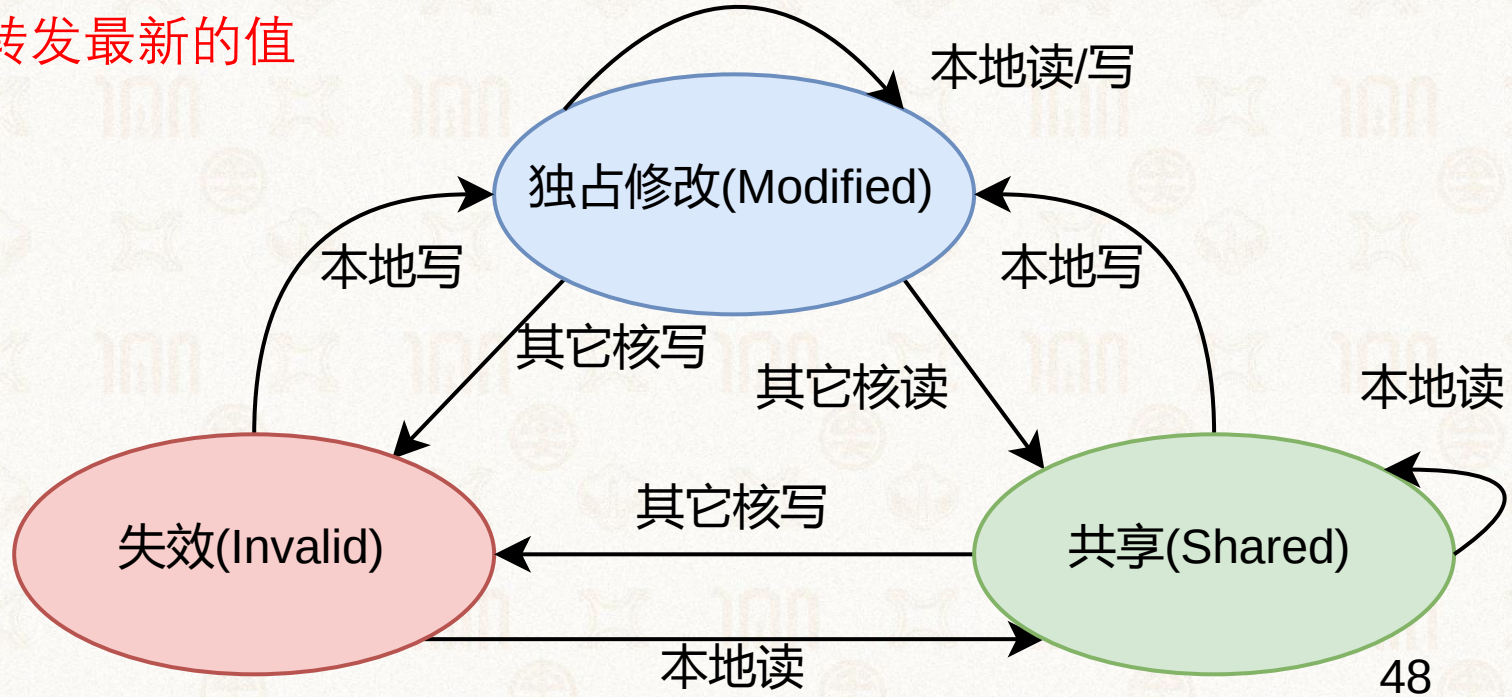
执行 LDR X

	CPU 0		CPU 1		CPU 2	
	状态	内容	状态	内容	状态	内容
X:	S	888	S	888	S	888

转发最新的值

全局共享目录项

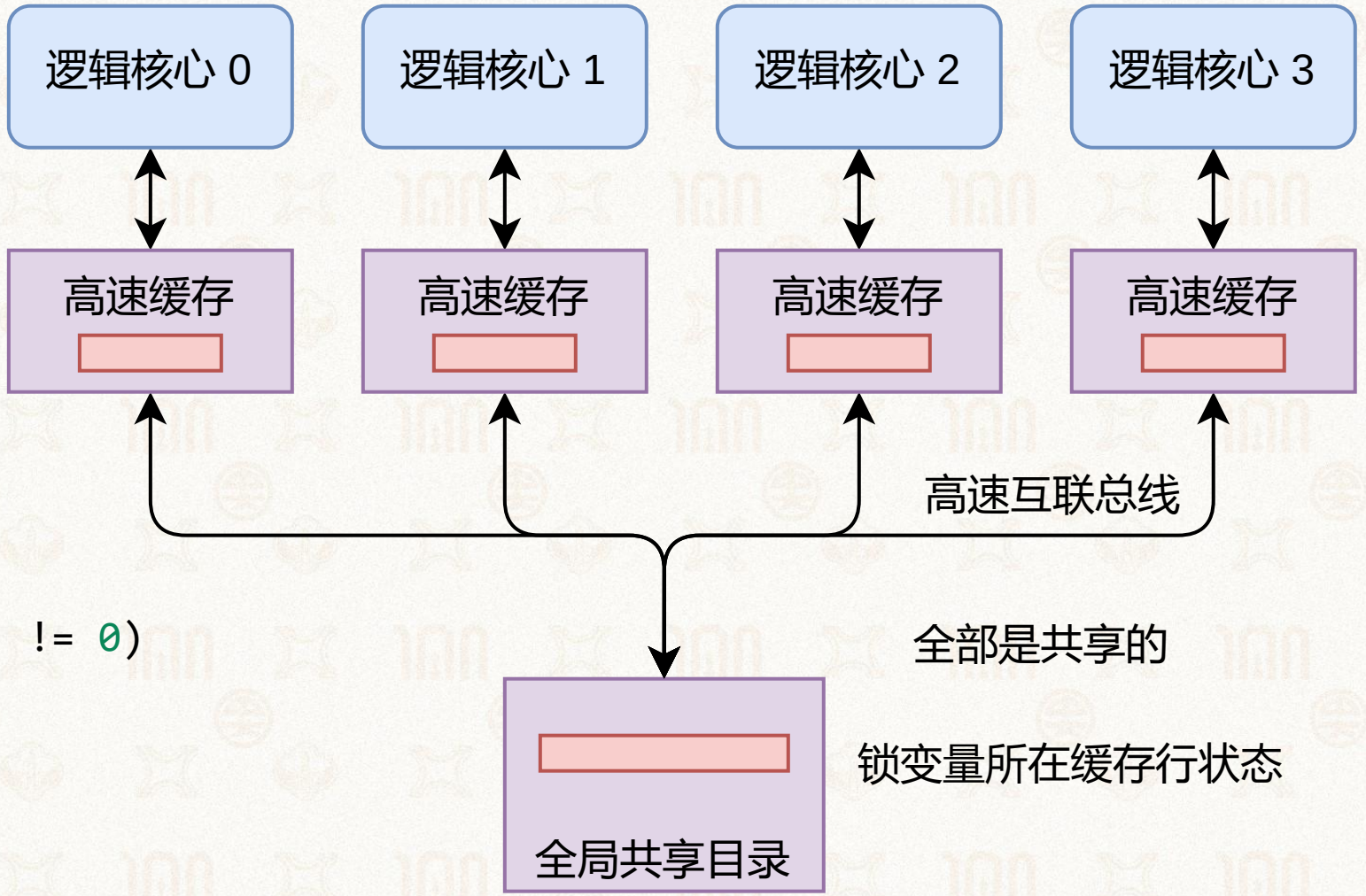
	脏位	向量位		
X:	0	1	1	1



大纲

- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

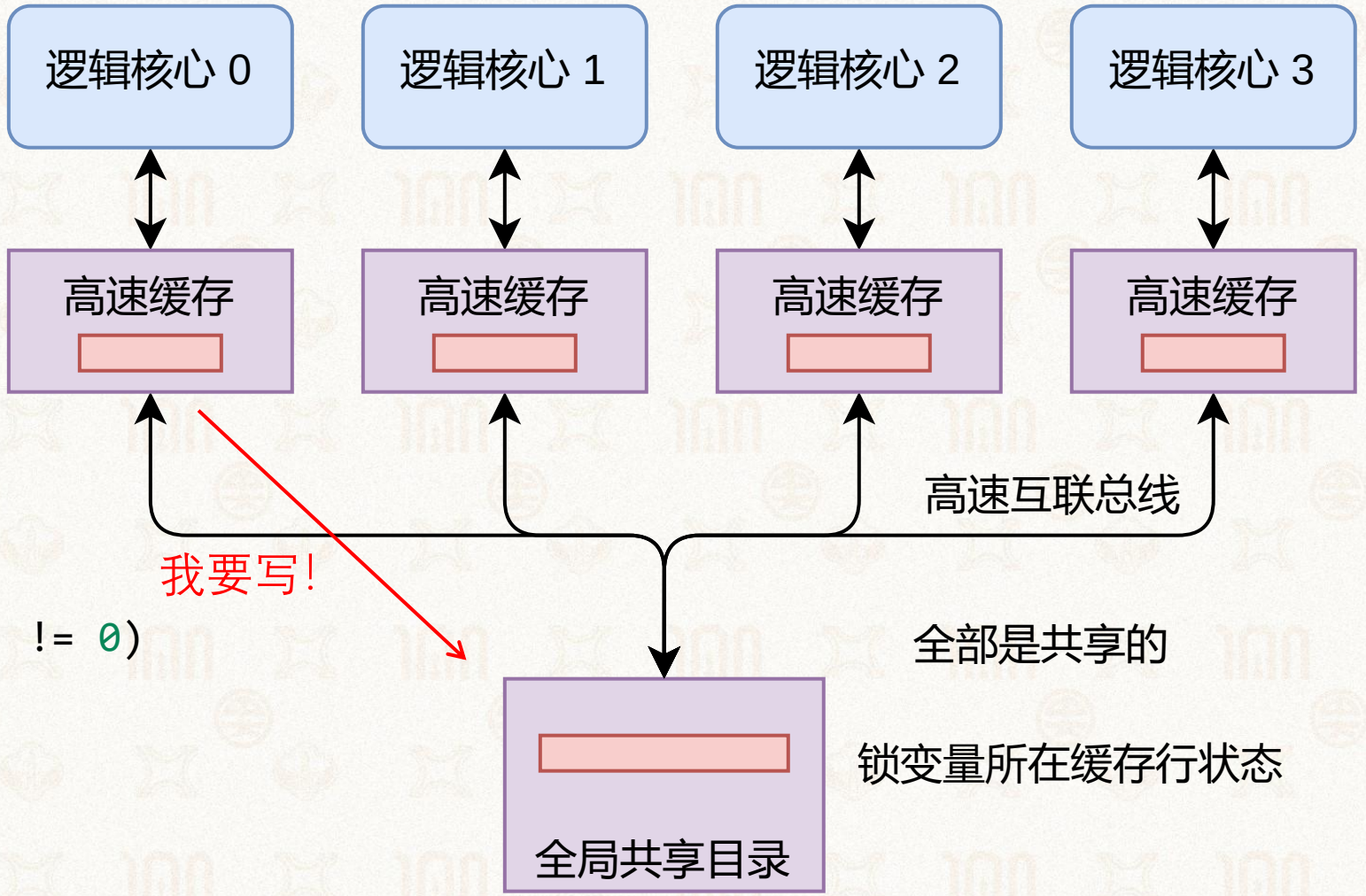
回到可扩展性断崖



- 自旋锁实现：

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

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

可扩展性断崖背后的原因



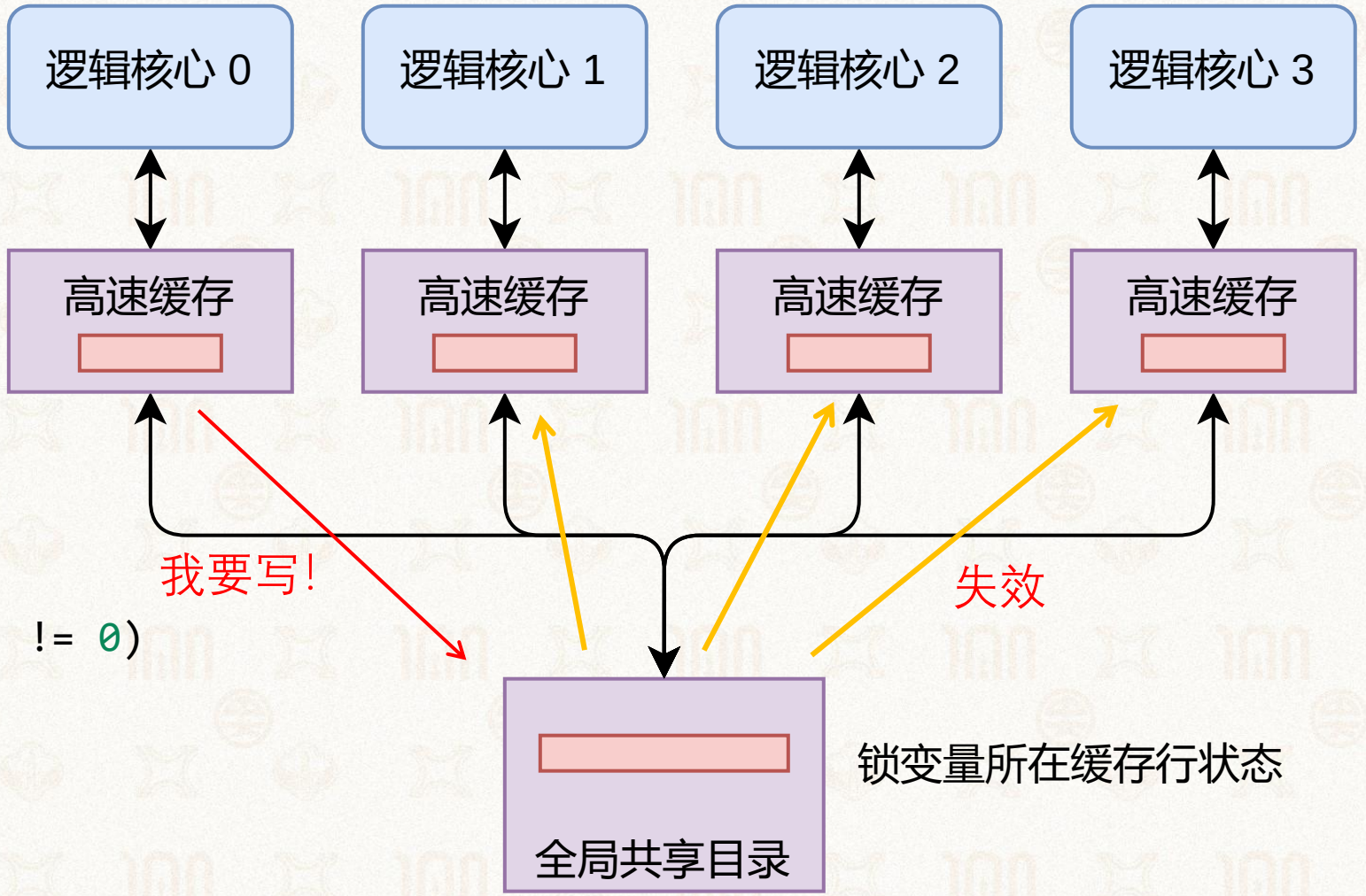
- 自旋锁实现：

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

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

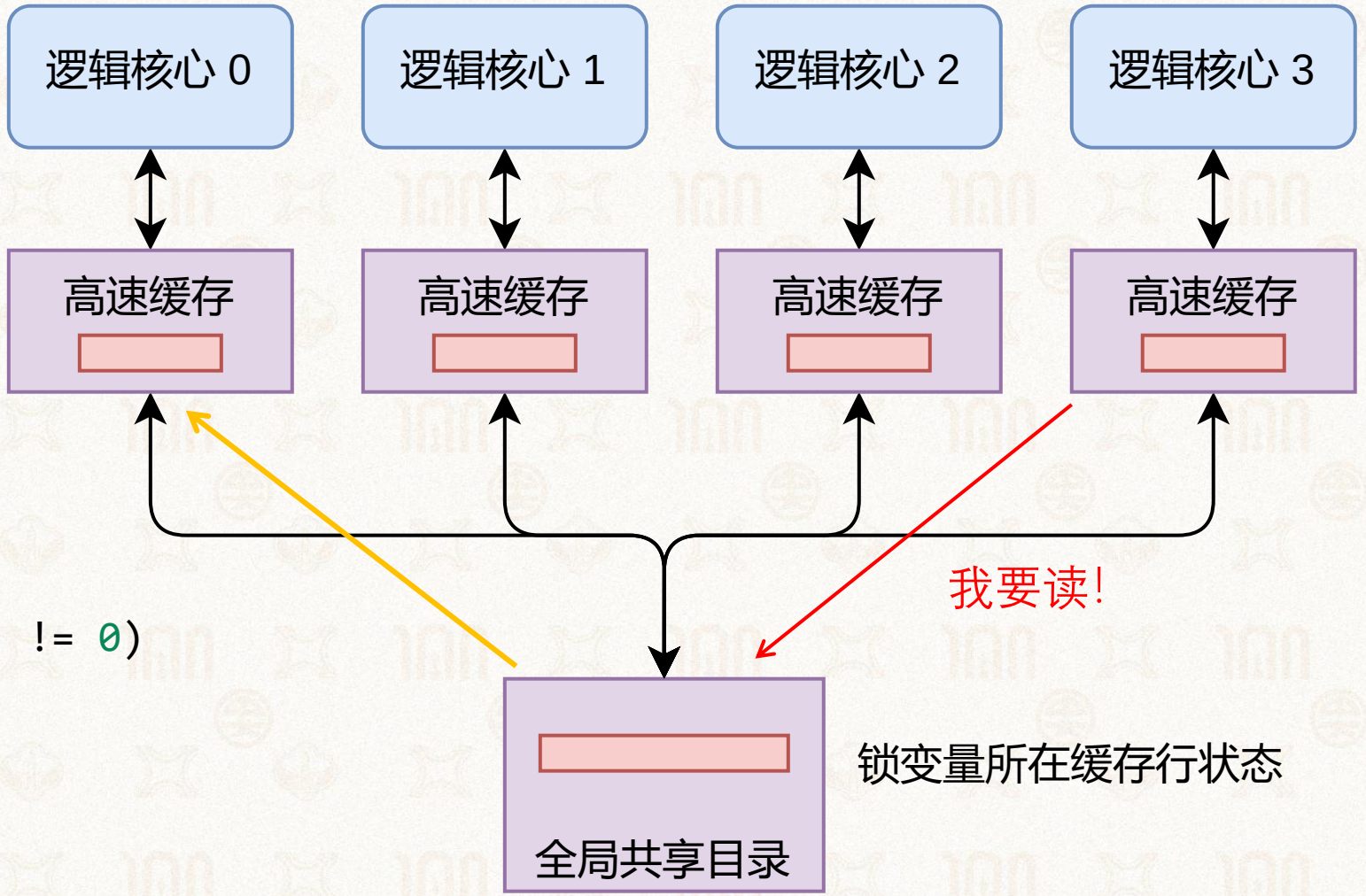
可扩展性断崖背后的原因

- 自旋锁实现：

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

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



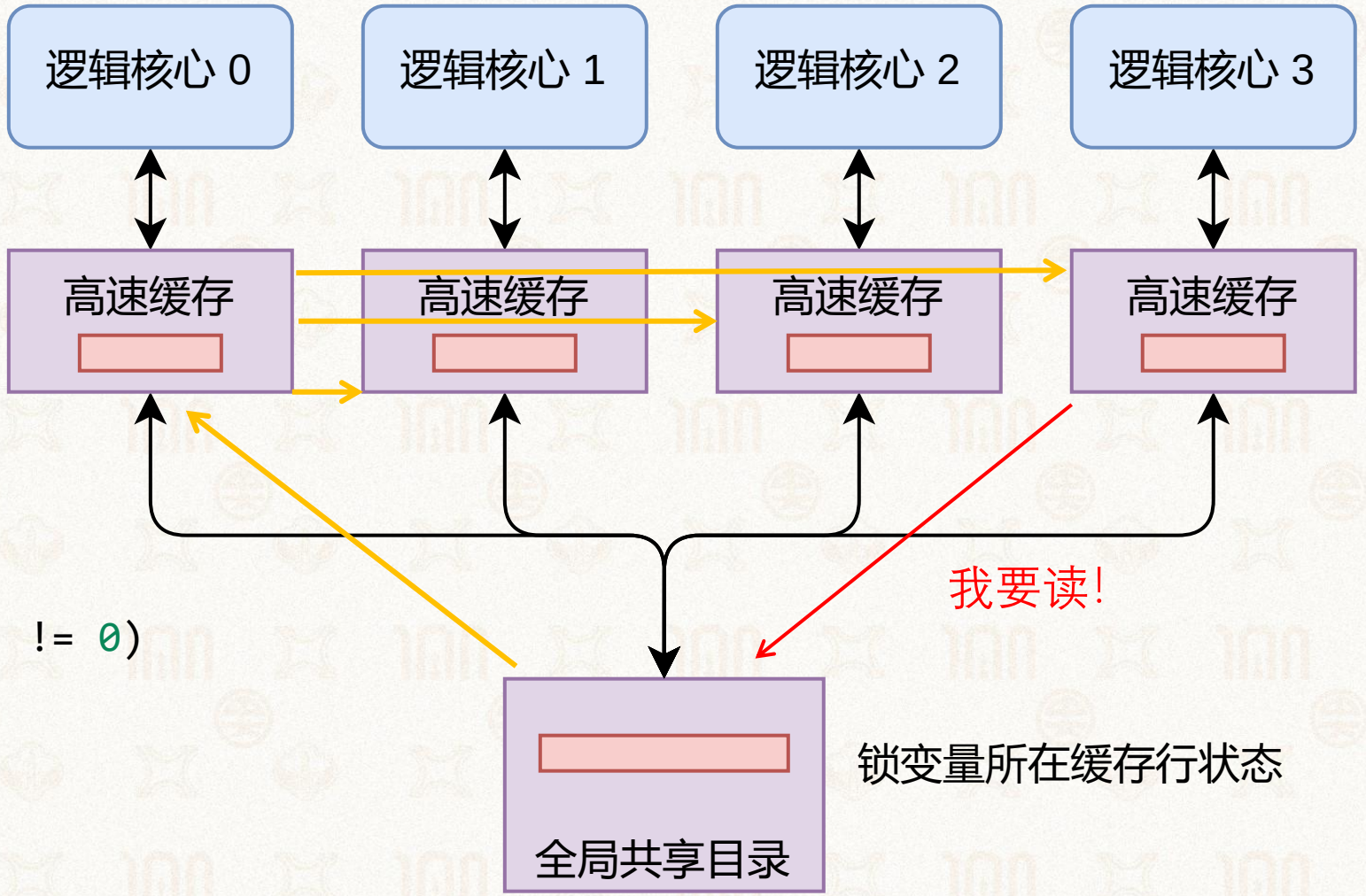
可扩展性断崖背后的原因



- 自旋锁实现：

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

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

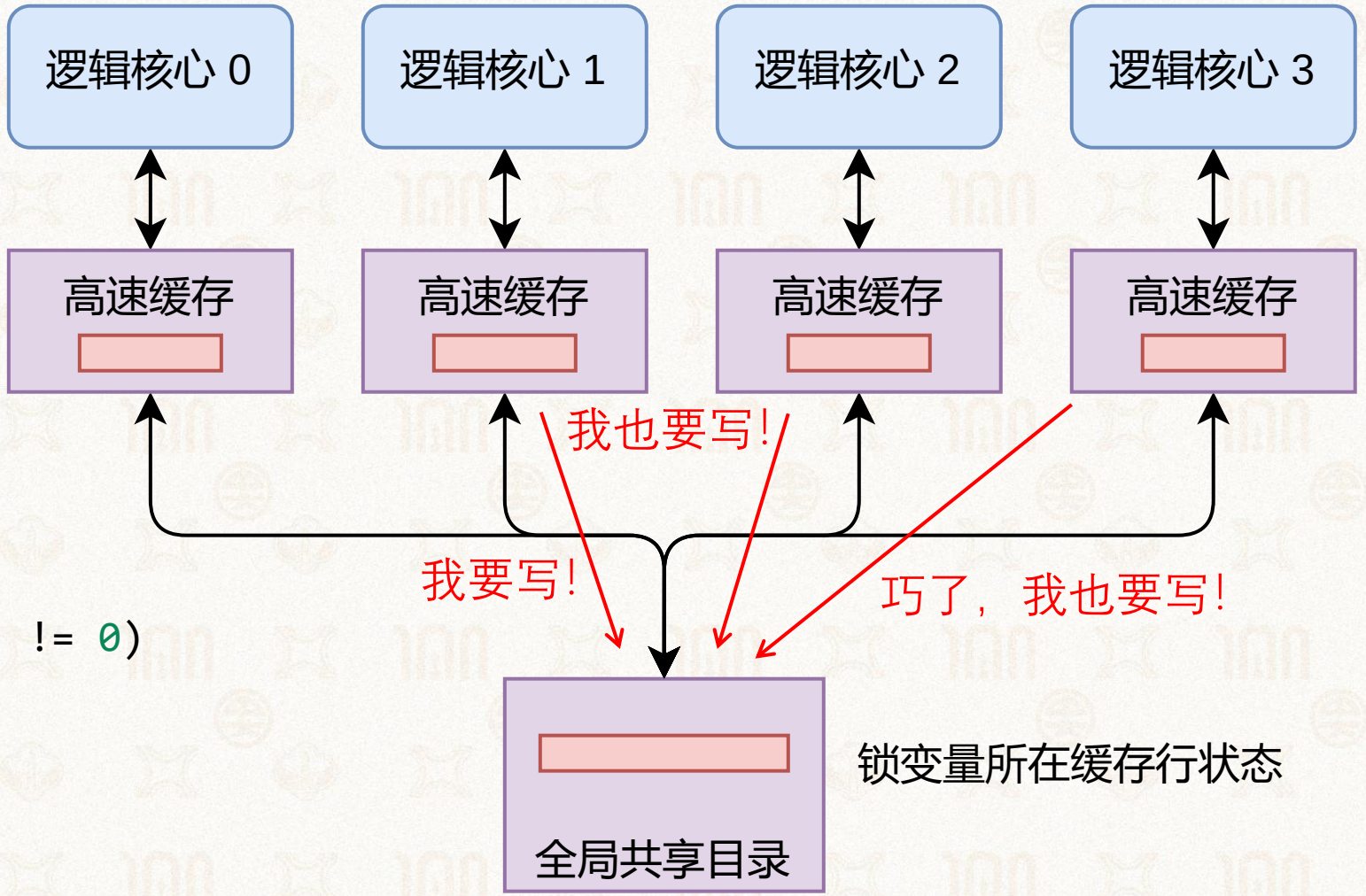
可扩展性断崖背后的原因



- 自旋锁实现：

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

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

可扩展性断崖背后的原因

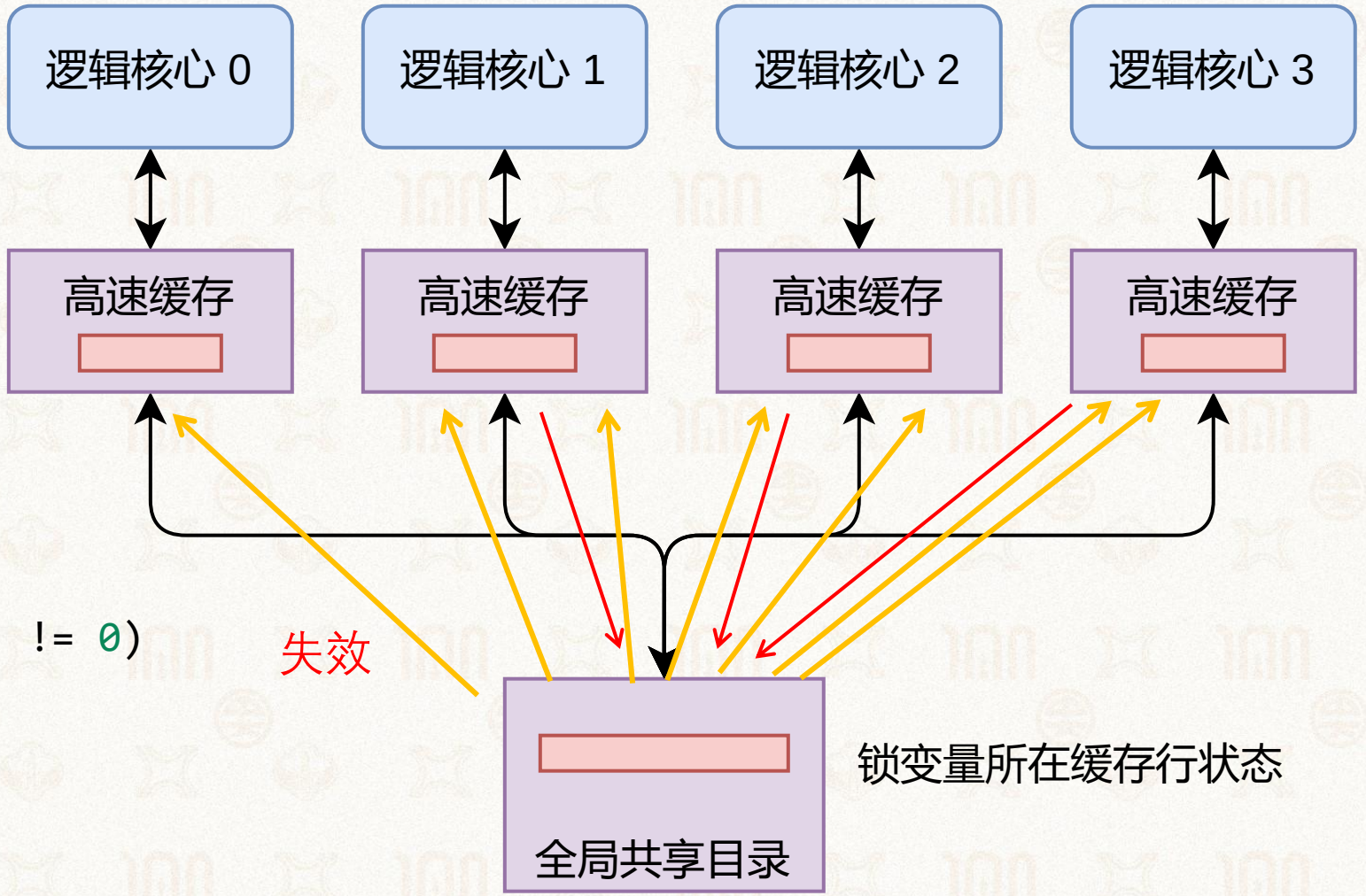


- 自旋锁实现：

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

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

可扩展性断崖背后的原因

- 对单一缓存行的竞争导致严重的性能开销
  - 产生大量的数据同步通讯



- 自旋锁实现：

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

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

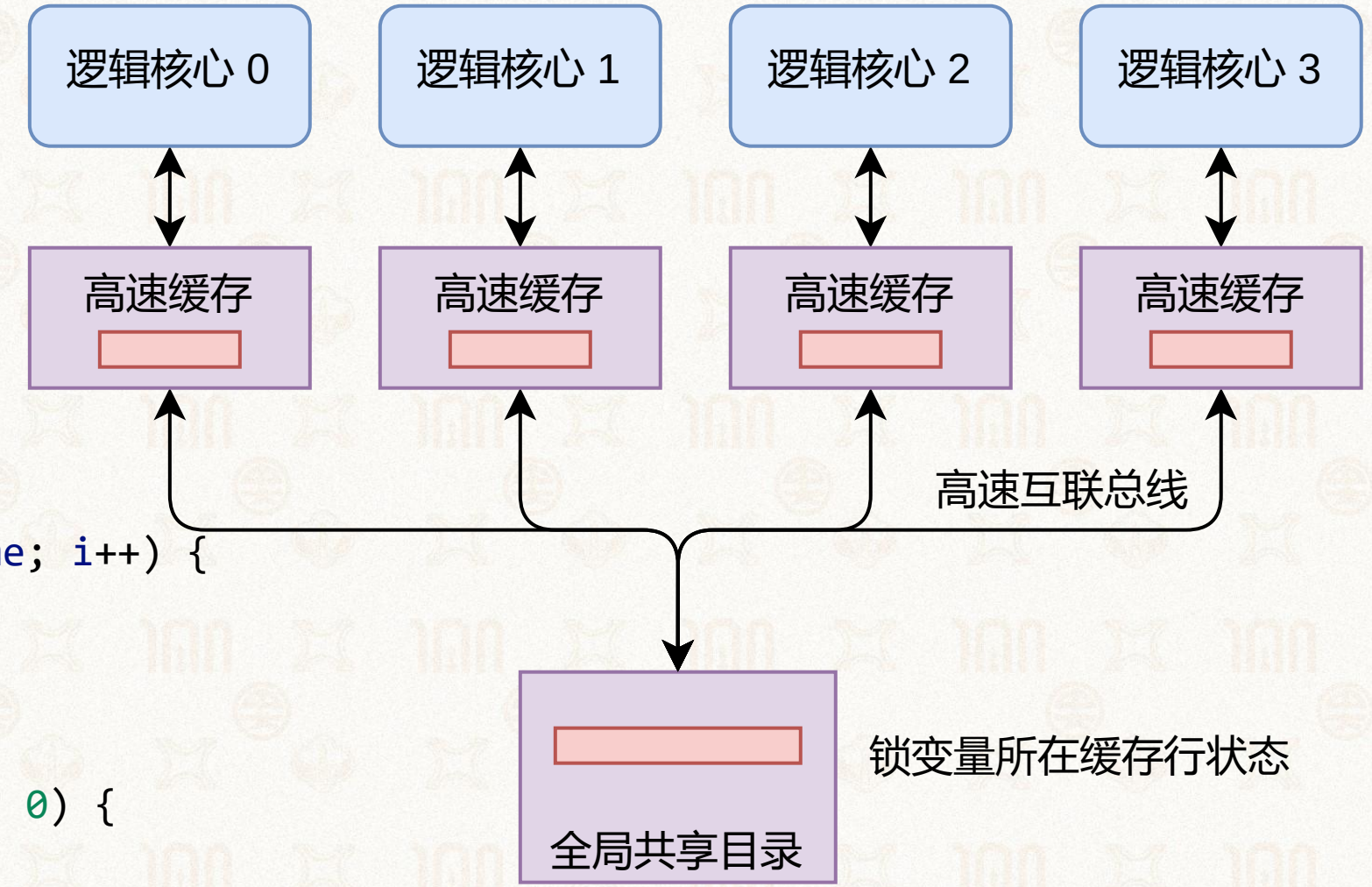
大纲

- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

如何解决可扩展性问题

- 避免对单一缓存行的高度竞争:
  - 回退(Back-off)策略

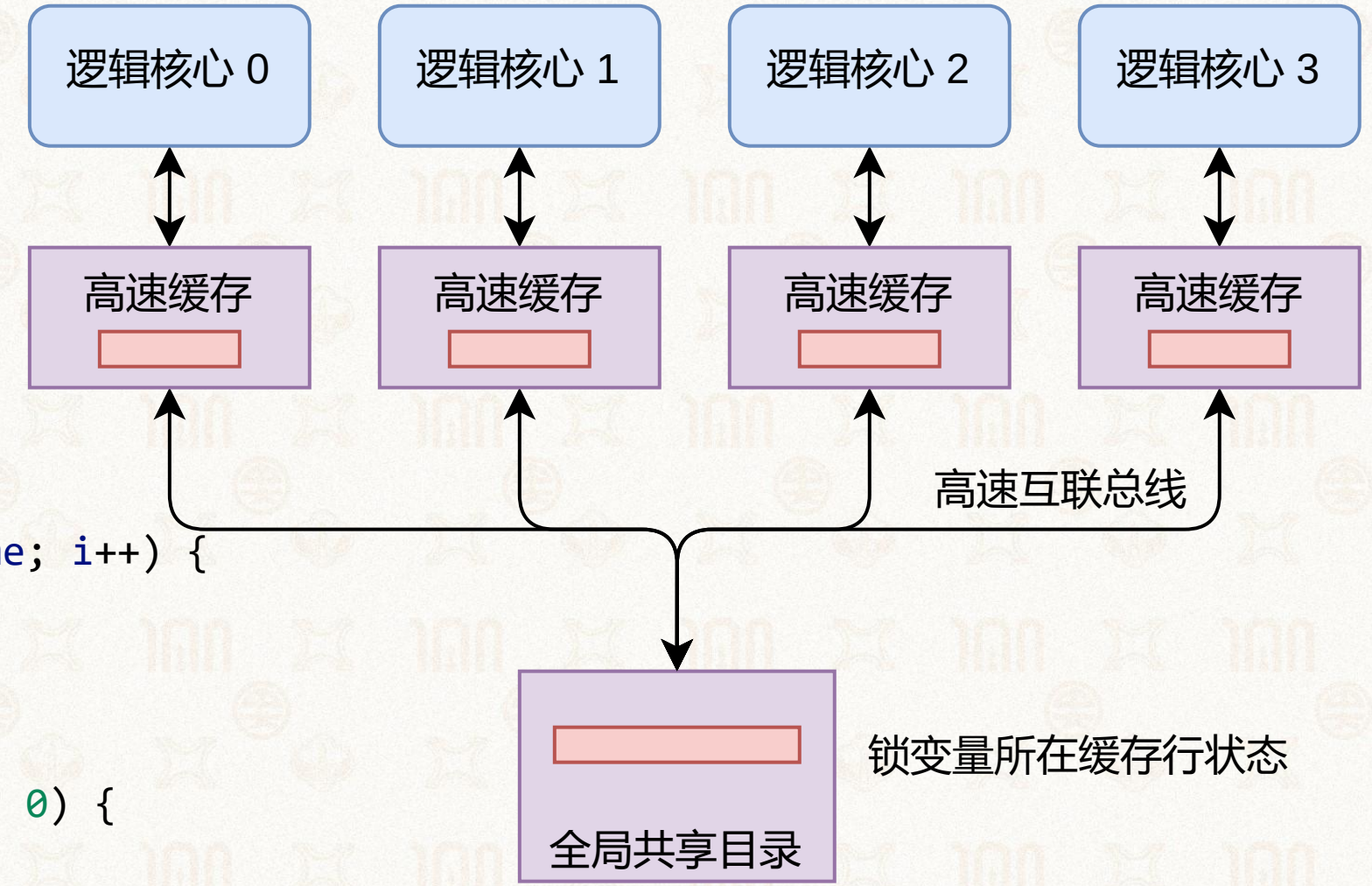
- 这样写能解决问题吗?
- 会有什么样的问题?




```
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);
    }
}
```

如何解决可扩展性问题

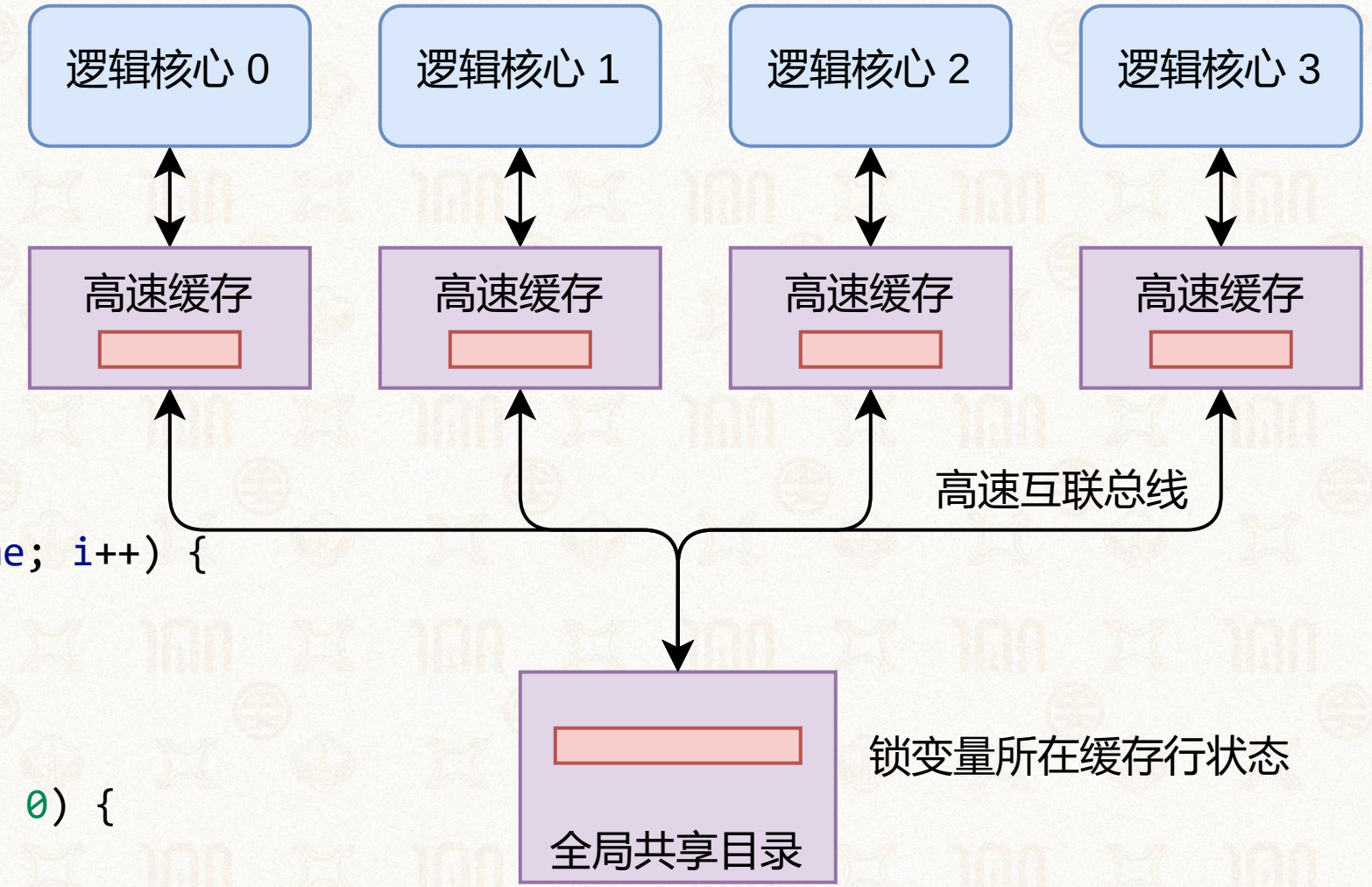
- 避免对单一缓存行的高度竞争:
  - 回退(Back-off)策略
- 等待相同时间
- 同时停止等待
- 同时开始下一轮竞争!




```
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);
    }
}
```

如何解决可扩展性问题

- 避免对单一缓存行的高度竞争:
  - 回退(Back-off)策略
- 随机时间
- 指数回退



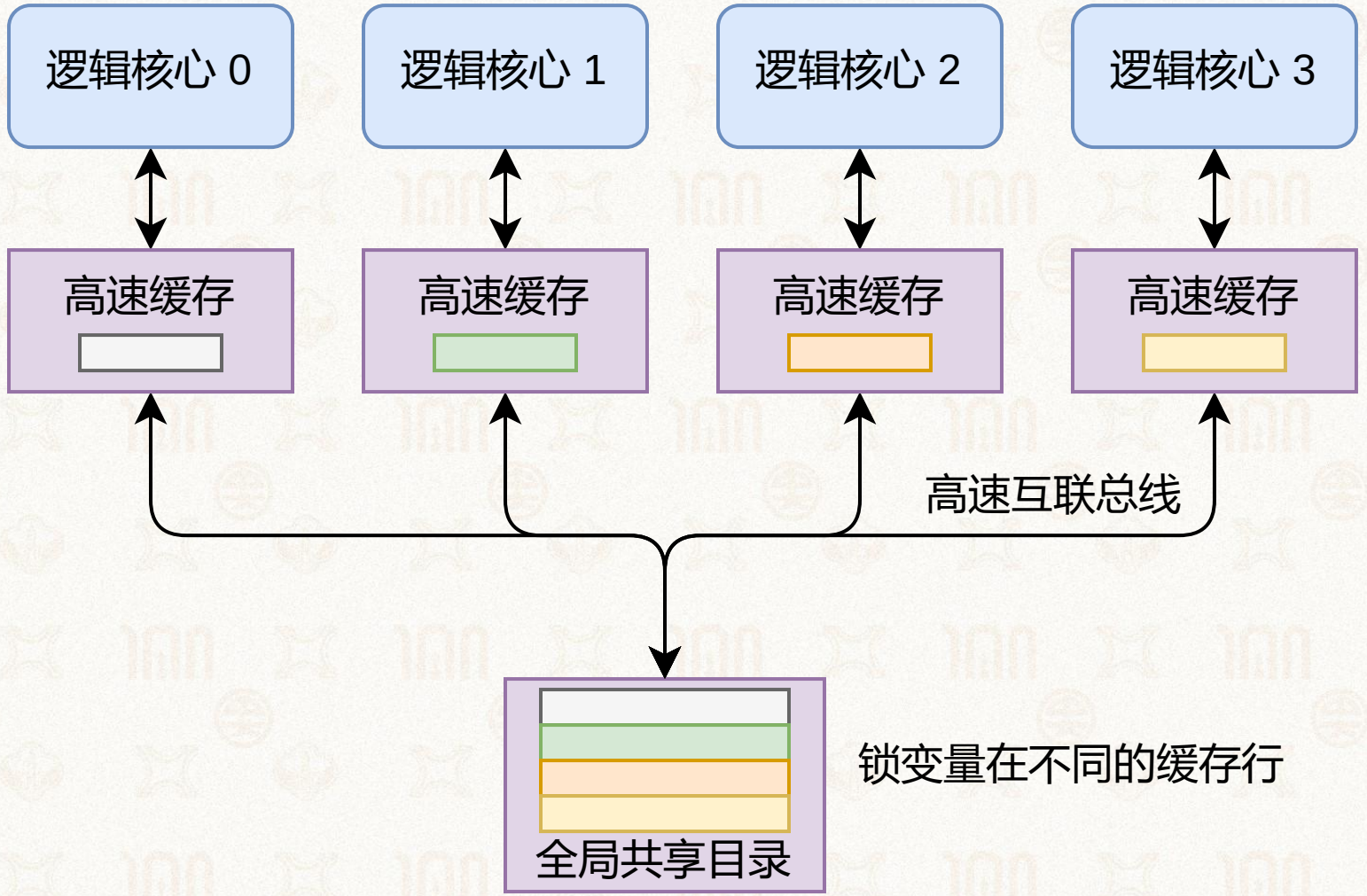

```
void back_off(int time) {
    for (volatile int i = 0; i < time; i++) {
        cpu_relax();
    }
}
void lock(int *lock) {
    while (atomic_CAS(lock, 0, 1) != 0) {
        back_off(DEFAULT_TIME);
    }
}
```

大纲

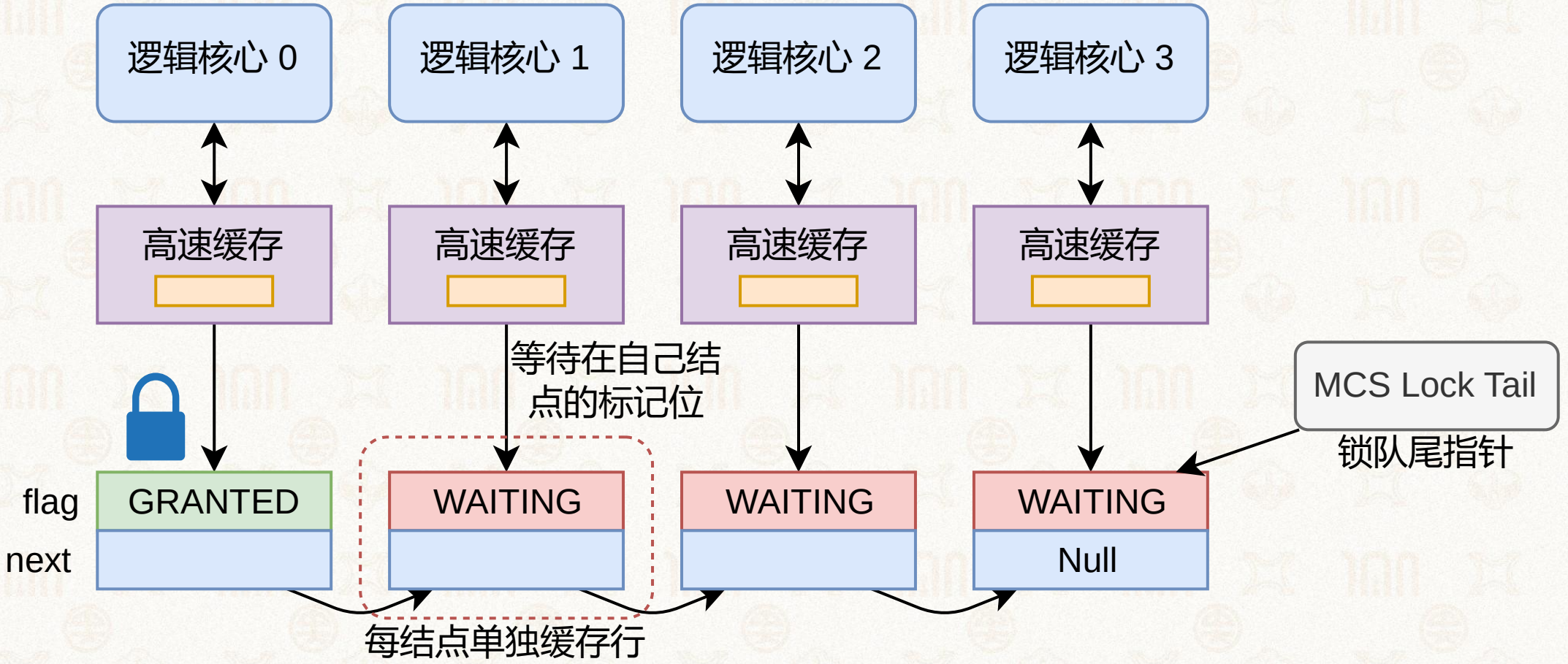
- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

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



MCS锁



Mellor-Crummey, John M., and Michael L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors." ACM Transactions on Computer Systems (TOCS) 9, no. 1 (1991): 21-65.

MCS锁：实现

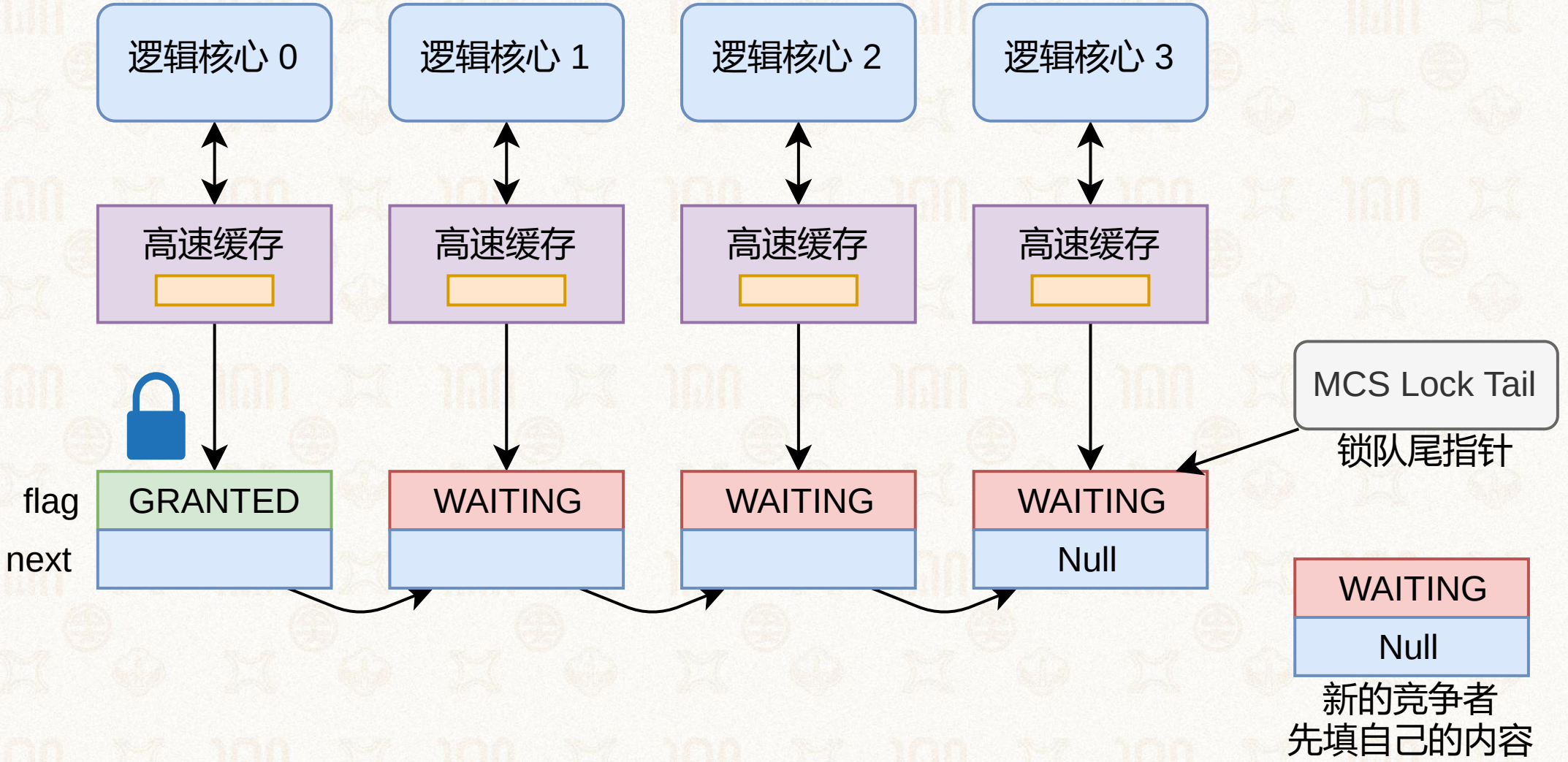
```
struct MCS_node {
    volatile int flag;
    volatile struct MCS_node *next;
} __attribute__((aligned(CACHELINE_SZ)));

struct MCS_lock {
    struct MCS_node *tail;
};
```

```
void unlock(struct MCS_lock *lock,
            struct MCS_node *me) {
    if (!me->next) {
        if (atomic_CAS(&lock->tail, me, 0) == me)
            return;
        while (!me->next)
            ;
    }
    me->next->flag = GRANTED;
}
```

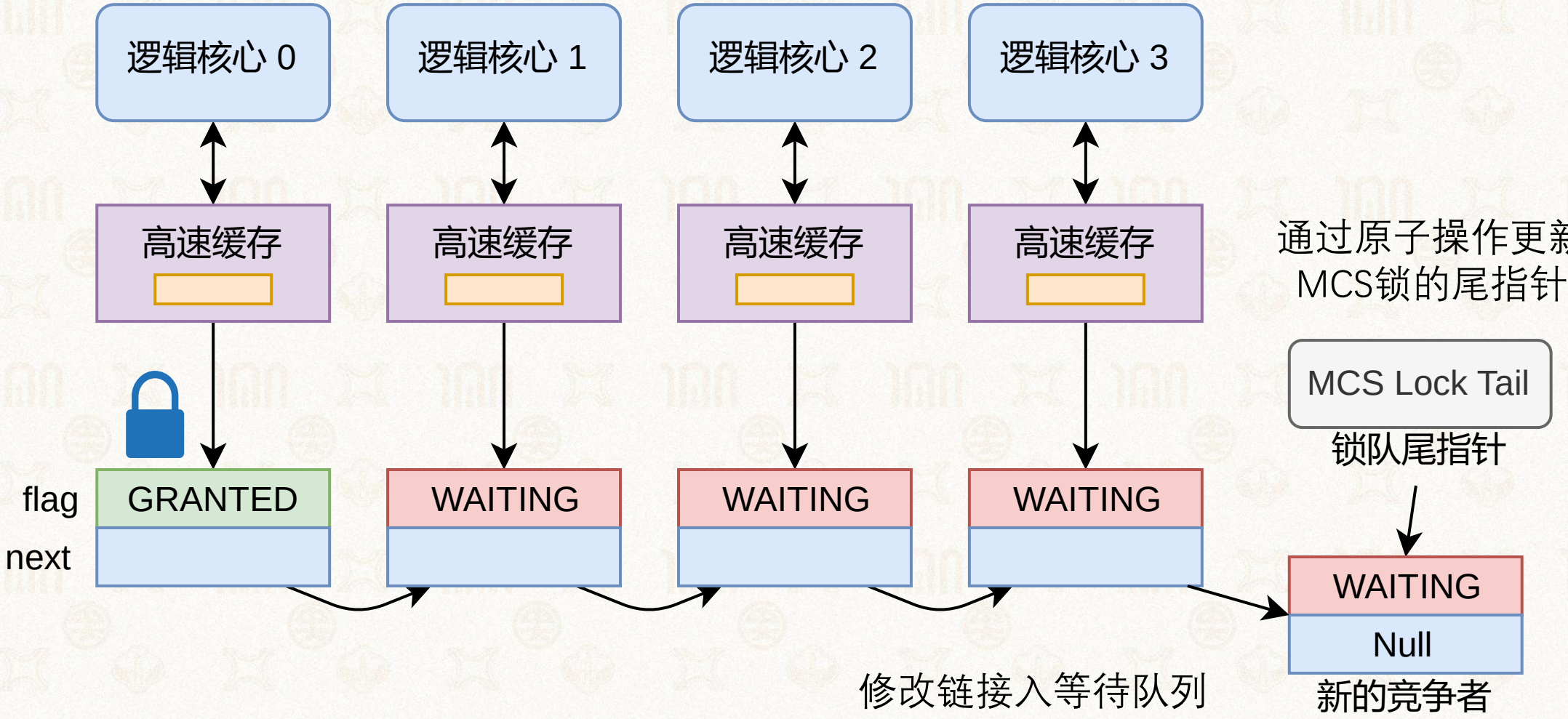
```
void lock(struct MCS_lock *lock, struct
MCS_node *me) {
    struct MCS_node *tail = 0;
    me->next = NULL;
    me->flag = WAITING;
    tail = atomic_XCHG(&lock->tail, me);
    if (tail) {
        tail->next = me;
        while (me->flag != GRANTED)
            ;
    }
}
```

- 新的原子操作:
  - atomic_XCHG: 交换数据

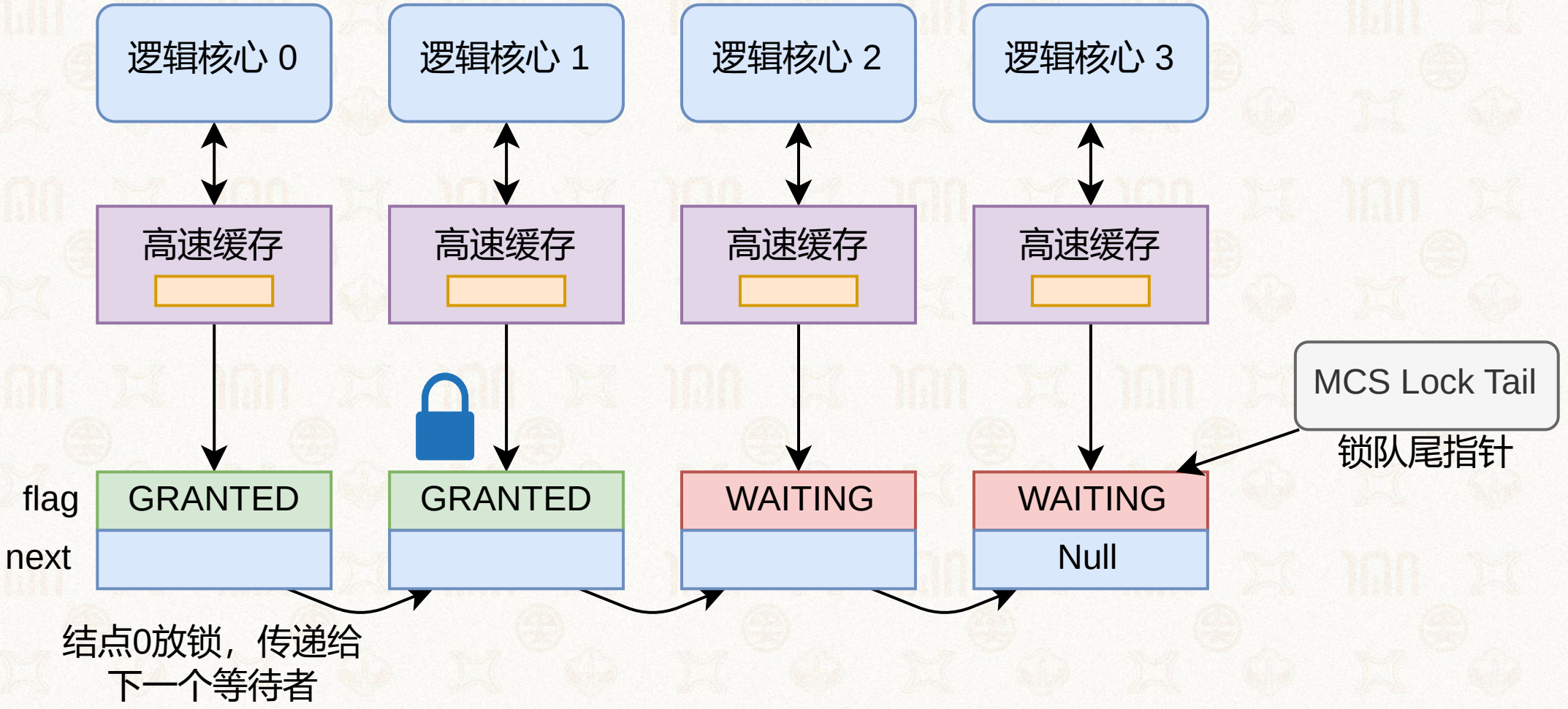
MCS锁：新的竞争者加入等待队列



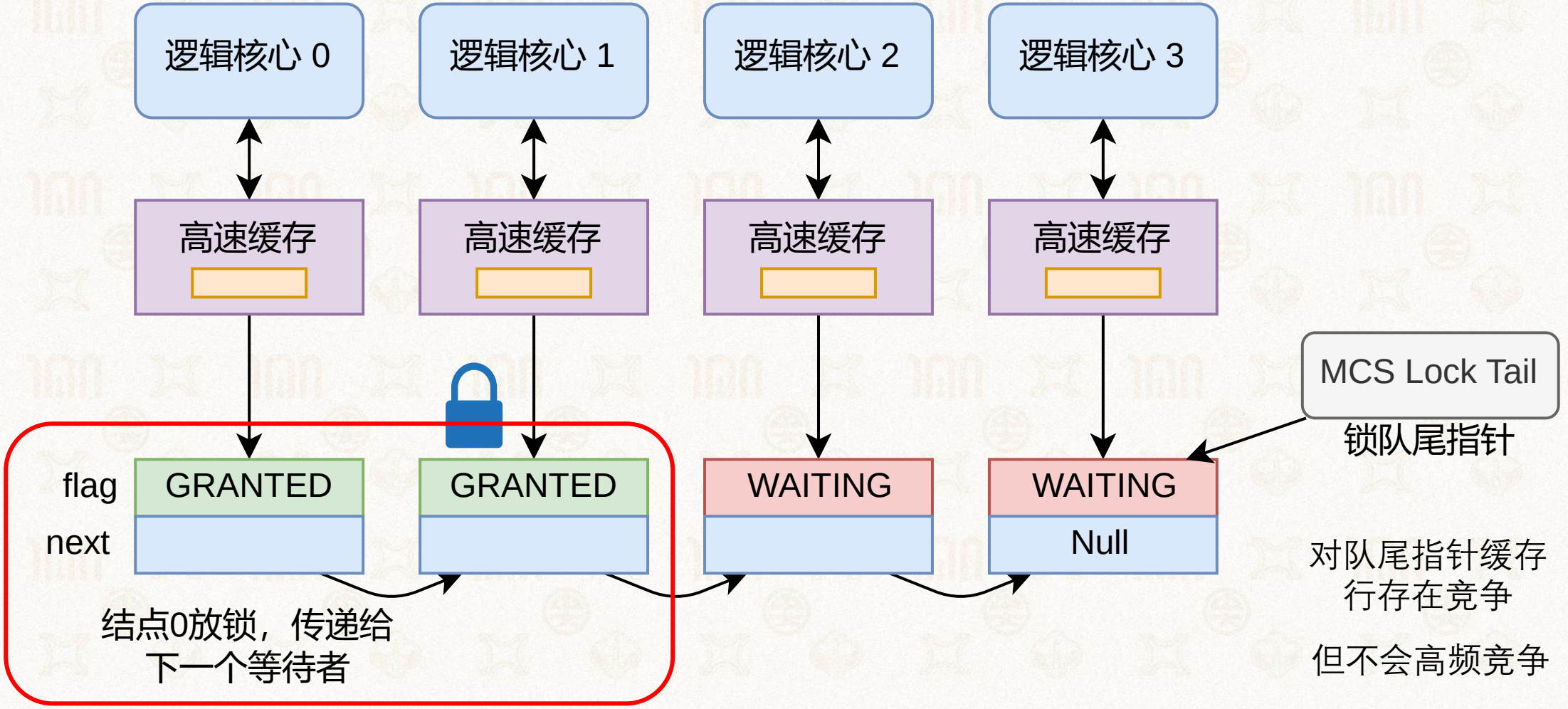
MCS锁：新的竞争者加入等待队列



MCS锁：锁持有者的传递

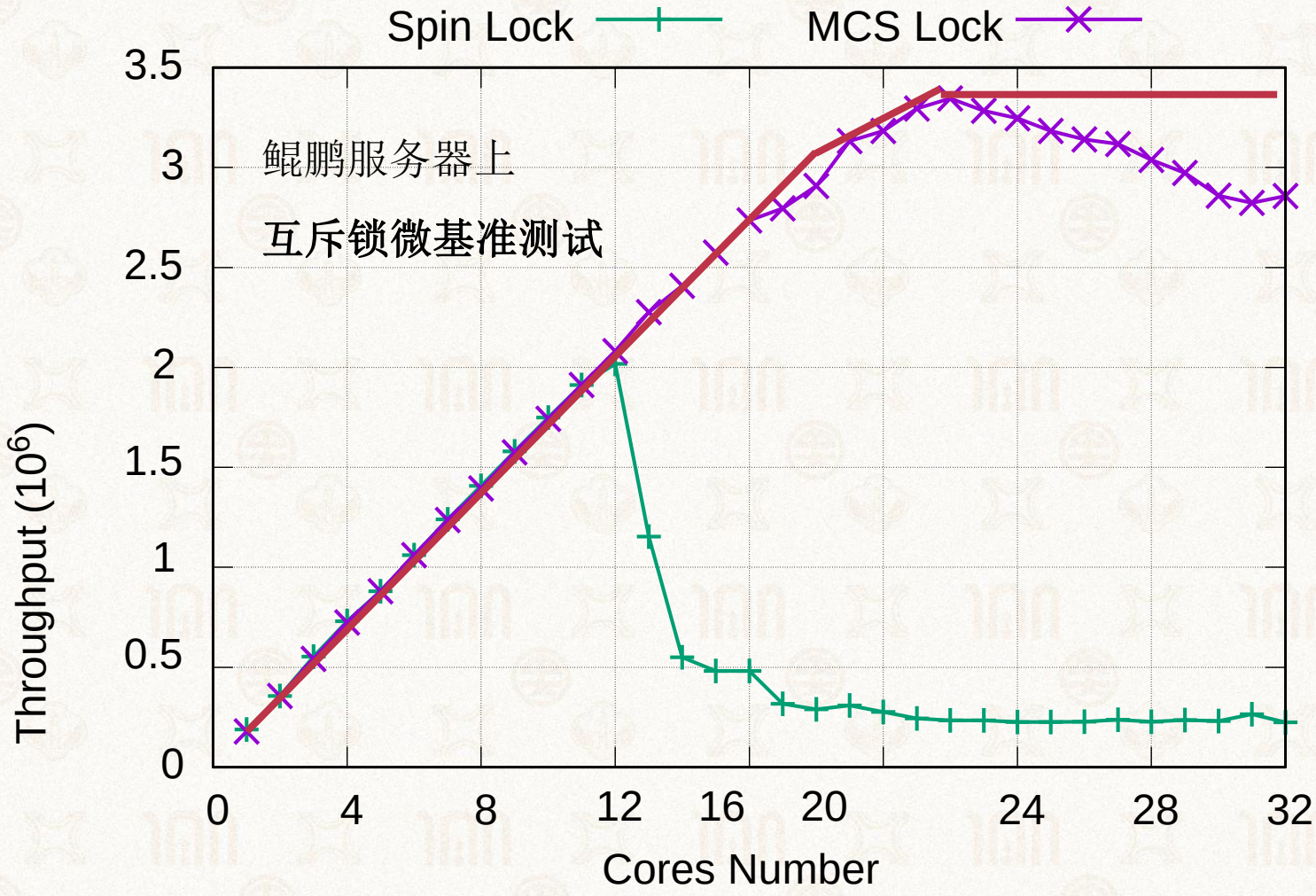


MCS锁：性能分析


逻辑核心 0
逻辑核心 1
逻辑核心 2
逻辑核心 3
高速缓存
高速缓存
高速缓存
高速缓存
MCS Lock Tail
锁队尾指针
flag
next
GRANTED
GRANTED
WAITING
WAITING
Null
结点0放锁，传递给
下一个等待者
对队尾指针缓存
行存在竞争
但不会高频竞争
写私有缓存行，不再会高频
竞争全局缓存行

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



Linux Kernel中的可扩展锁：QSpinlock

- 快速路径：
  - 竞争程度低
  - 使用类似自旋锁设计
  - 加锁/放锁流程简单

- 慢速路径
  - 使用类似MCS锁设计
  - 可扩展性好

https://elixir.bootlin.com/linux/v5.16.14/source/include/asm-generic/qspinlock_types.h#L14

大纲

- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

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

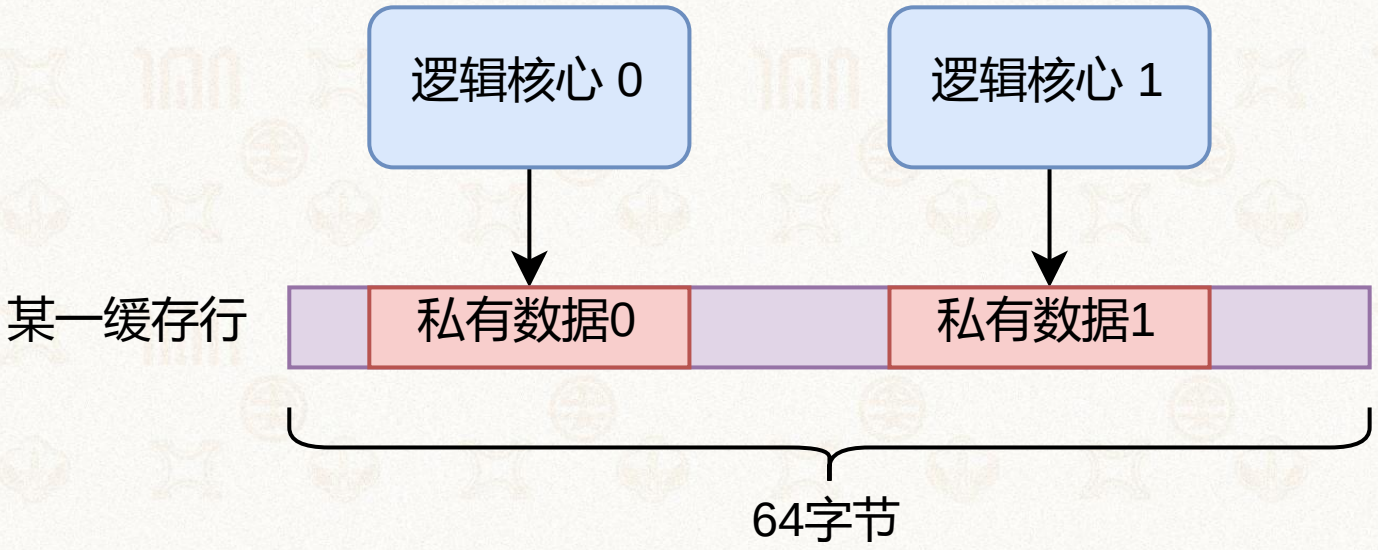
- 对软件开发者透明

- 系统软件开发者视角：
  - 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
  - 虚假共享（False Sharing）在多核情况下是致命的

虚假共享

- 一般缓存行大小为64字节
- 程序在一个页表内，经常连续分配内存空间

```
double num[8];
```




```
int a;
int b;
int c;
```

num[1] 和 num[7] 一定在同一个缓存行中

a b c 也可能在同一个缓存行中

- 虚假共享在多核情况下是致命的
  - 无谓的缓存一致性同步通讯
  - 可能发生写覆盖

大纲

- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

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

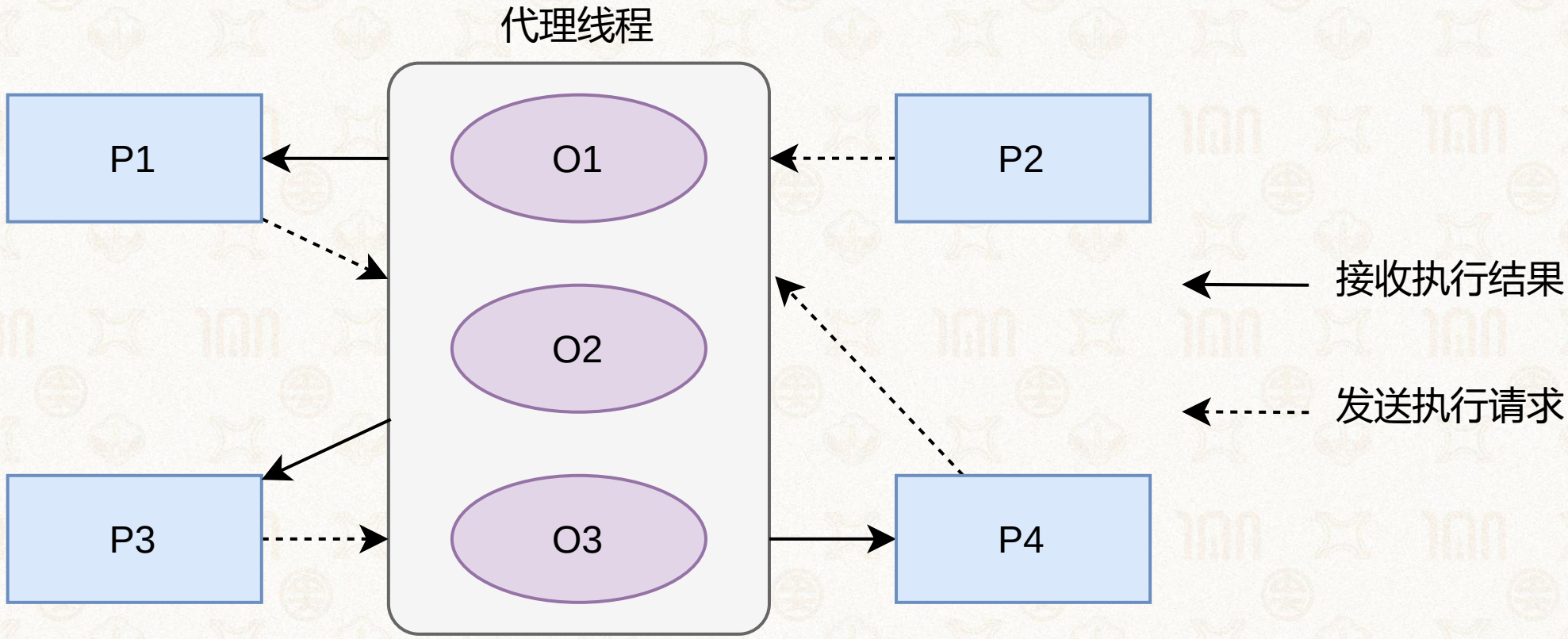
```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

T1时刻的死锁

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）



- 缺点是大部分程序都不太容易修改为这种模式

死锁预防：四个方向

- 2. 不允许持有并等待：一次性申请所有资源

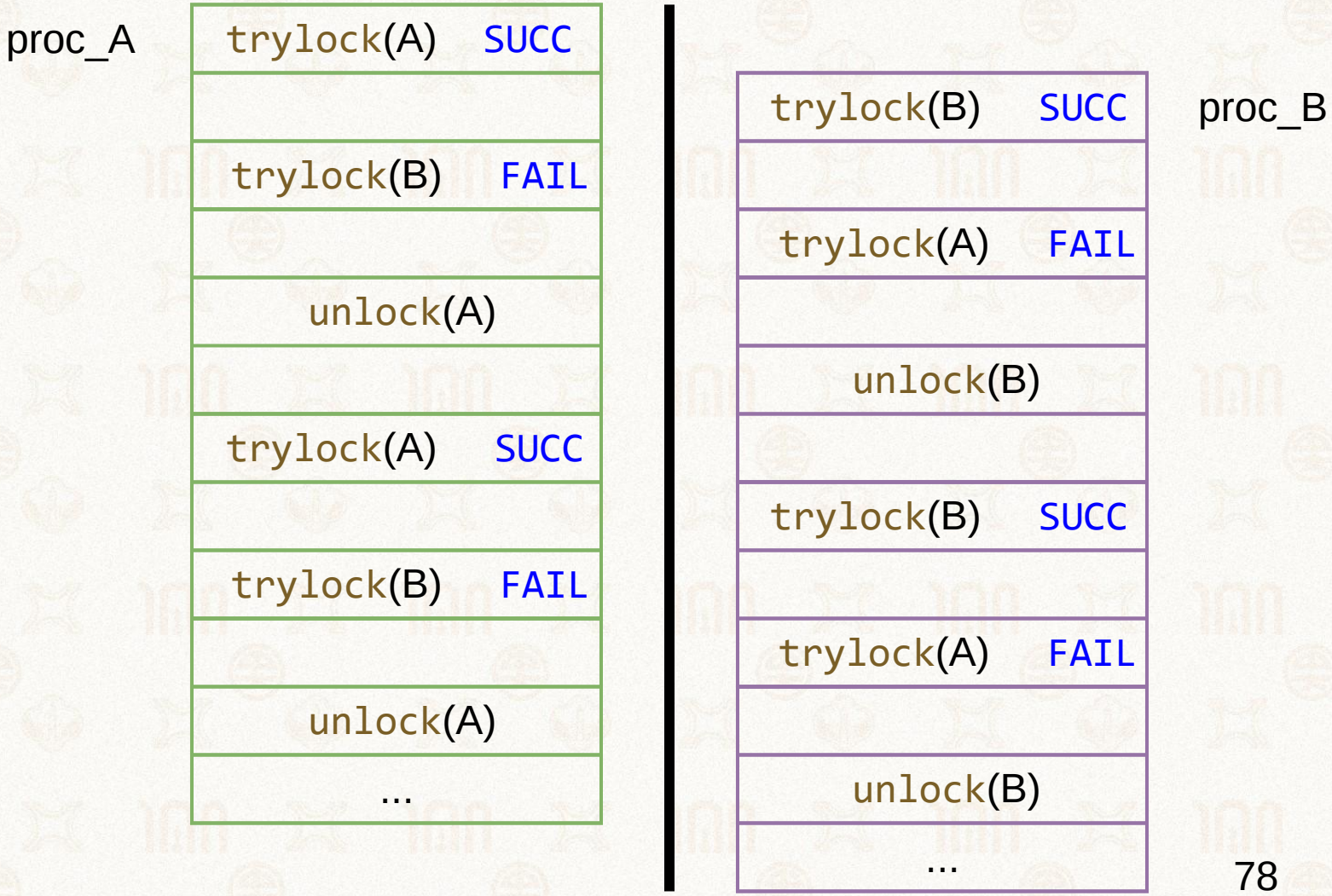
```
while(true) {
    if(trylock(A) == SUCC) {    // trylock非阻塞，立即返回成功或失败
        if(trylock(B) == SUCC) {
            // 临界区代码
            // ...
            unlock(B);
            unlock(A);
            break;
        } else {
            unlock(A); // 无法获取B，那么释放A
        }
    }
}
```

死锁预防：四个方向

- 2. 不允许持有并等待：一次性申请所有资源

```
while(true) {
    if(trylock(A) == SUCC) {
        if(trylock(B) == SUCC) {
            // 临界区代码
            // ...
            unlock(B);
            unlock(A);
            break;
        } else {
            unlock(A);
        }
    }
}
```

运气很差时可能出现如此往复，但运气不会一直这么差



死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源
  - 让proc_B回滚

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源
  - 让proc_B回滚

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
→   unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
→   // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源
  - 让proc_B回滚
  - proc_A结束后再恢复proc_B的执行

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源
  - 让proc_B回滚
  - proc_A结束后再恢复proc_B的执行

- 回滚和恢复只适用于易于保存和恢复的场景

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 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：(A、B、C、D...)

  - 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

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

大纲

- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型

并行程序题

- 有两个并发进程P1和P2，其程序代码如下：

```
P1() {
    x = 1;
    y = 2;
    if(x > 0) {
        z = x + y;
    } else {
        z = x * y;
    }
    printf("%d", z);
}
```

```
P2() {
    x = -1;
    a = x + 3;
    x = a + x;
    b = a + x;
    c = b * b;
    printf("%d", c);
}
```

- 假设每条赋值语句是一个原子操作，其中只有x是P1和P2的共享变量。
  - 1) 可能打印出的z值分别是多少？
  - 2）可能打印出的c值分别是多少？

并行程序题

能对P2的正确性造成伤害的，只有它

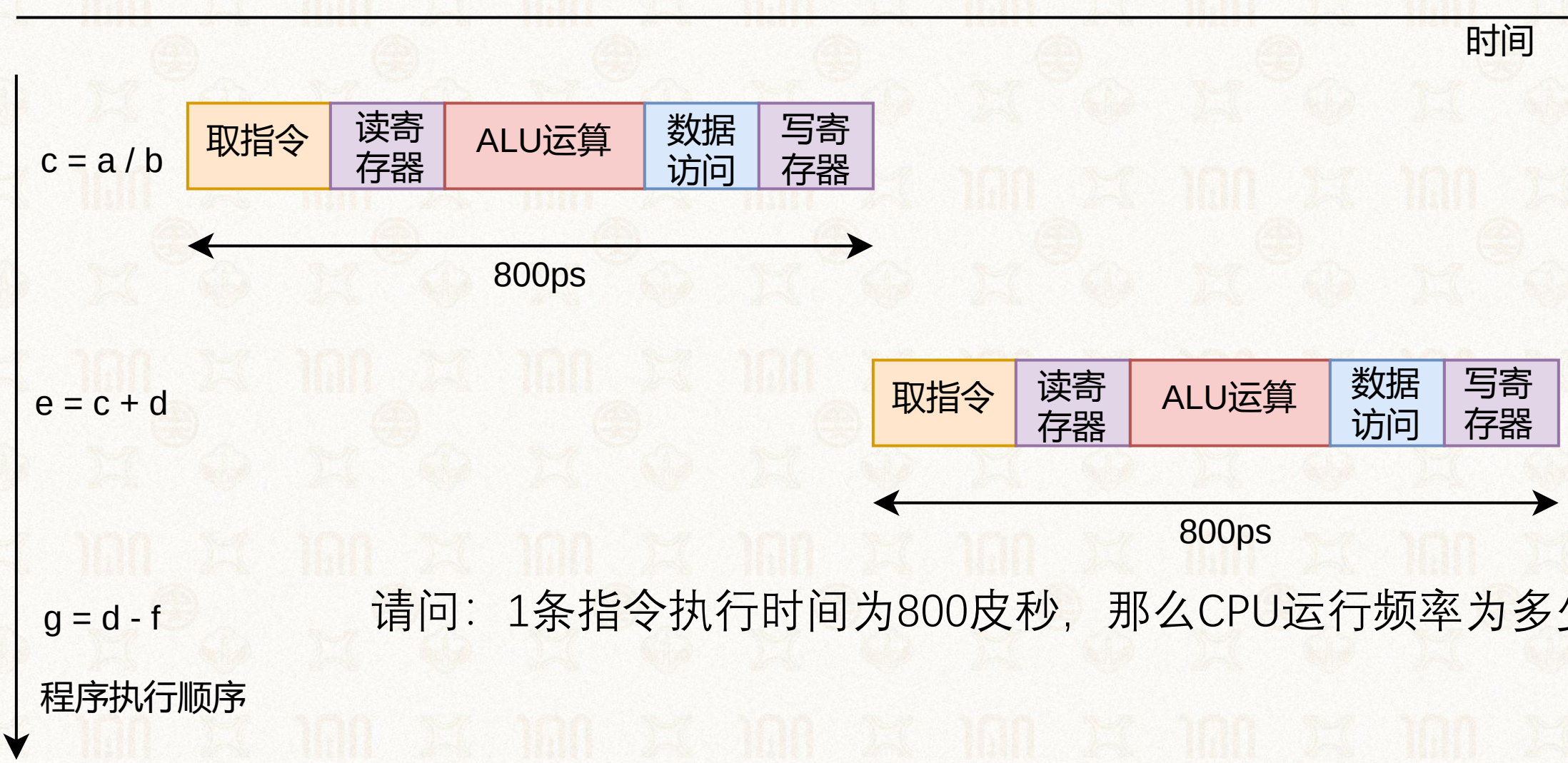
P1:	P2:
x = 1;	x = -1;
y = 2;	a = x + 3;
if(x > 0) {	x = a + x;
z = x + y;	b = a + x;
} else {	c = b * b;
z = x * y;	printf("%d", c);
}	
printf("%d", z);	

理想状态

P1:	P2:
x = 1;	x = -1;
	a = x + 3;
y = 2;	x = a + x;
if(x > 0) {	b = a + x;
z = x + y;	
} else {	c = b * b;
z = x * y;	printf("%d", c);
}	
printf("%d", z);	

实际：任何指令都可能有任意长度的延时

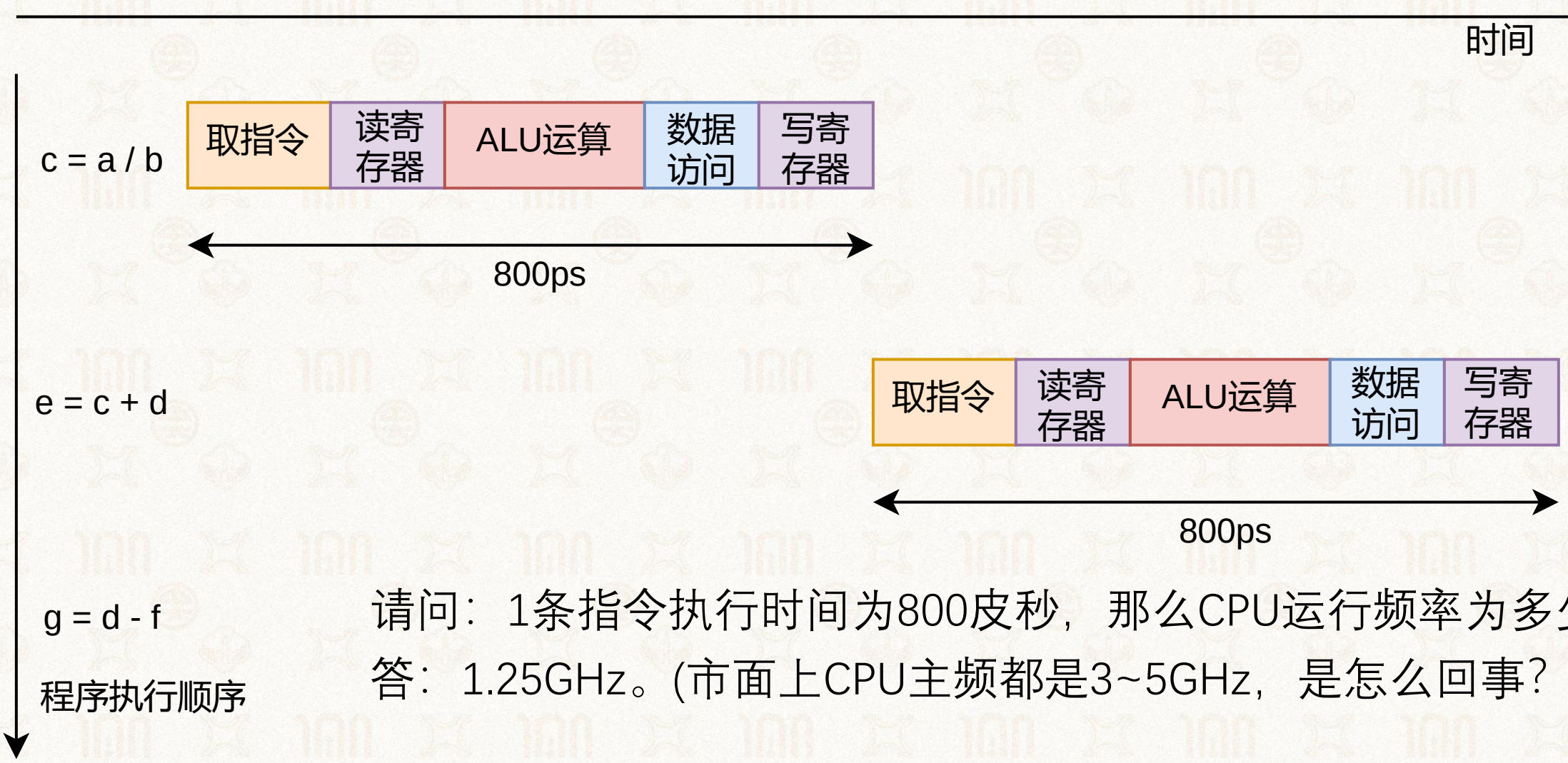
理想的指令执行过程



请问：1条指令执行时间为800皮秒，那么CPU运行频率为多少？

1000皮秒=1纳秒，1000纳秒=1微秒，1000微秒=1毫秒，1000毫秒=1秒

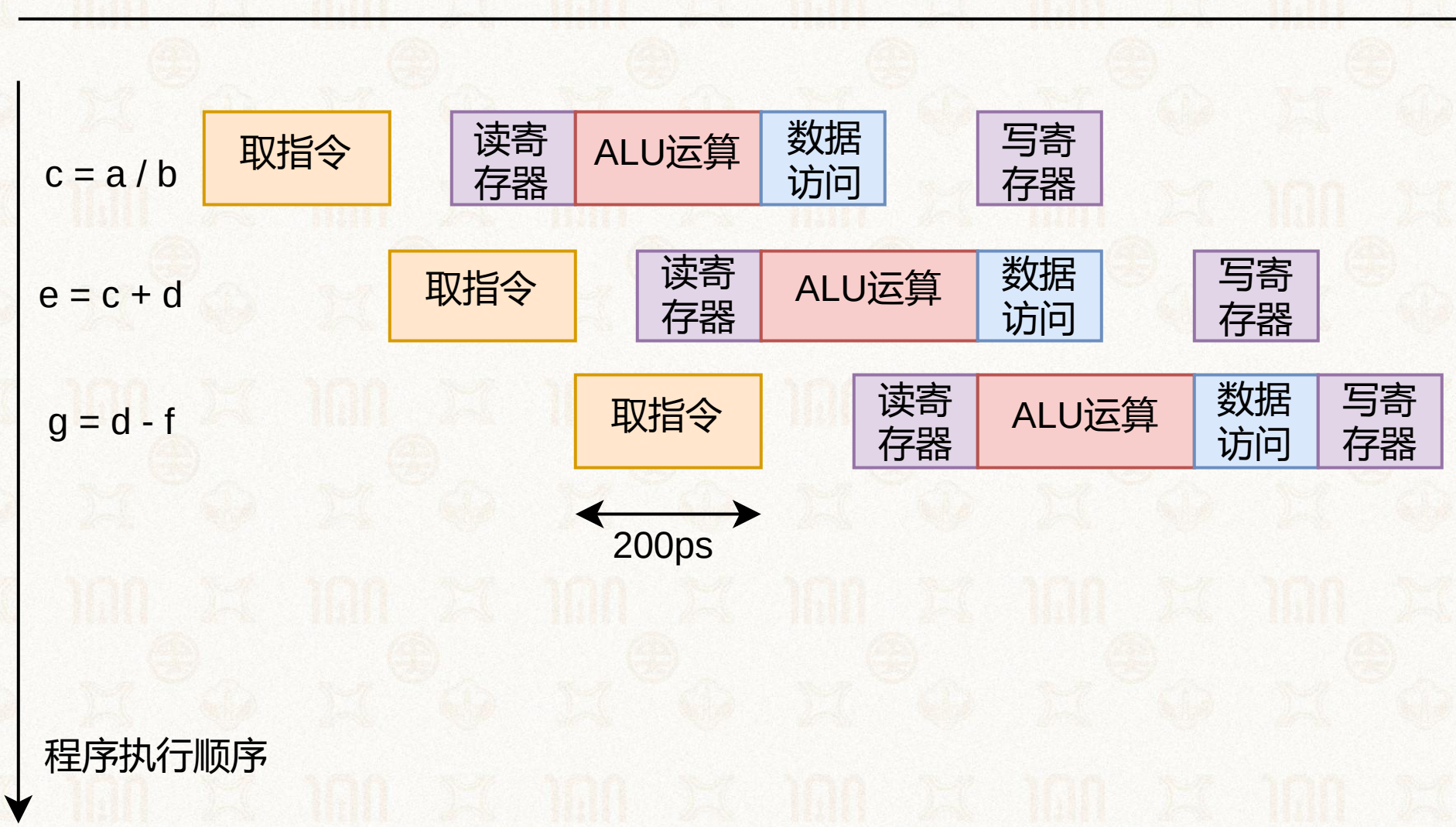
理想的指令执行过程



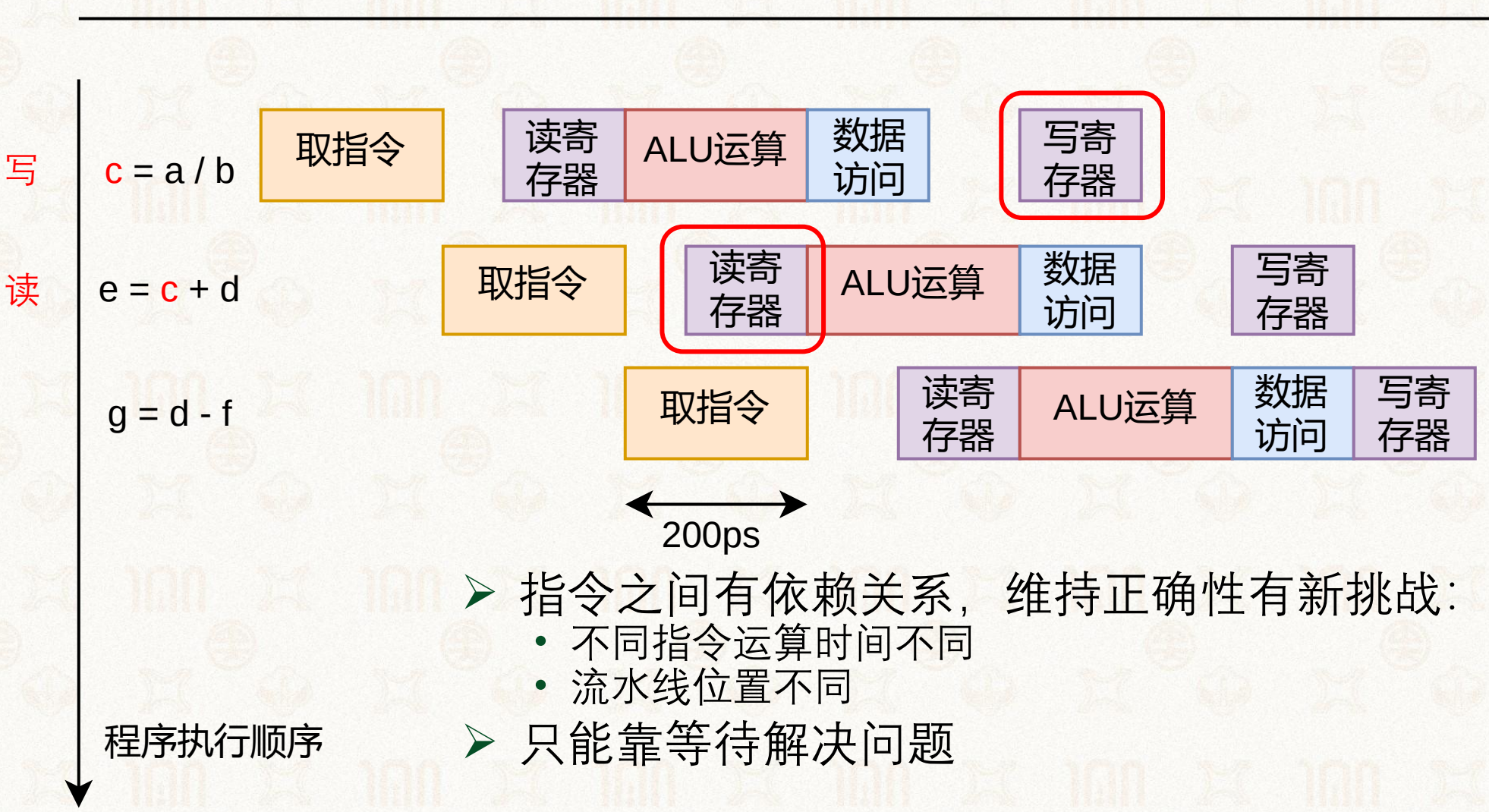
请问：1条指令执行时间为800皮秒，那么CPU运行频率为多少？

答：1.25GHz。(市面上CPU主频都是3~5GHz，是怎么回事？)

流水线执行

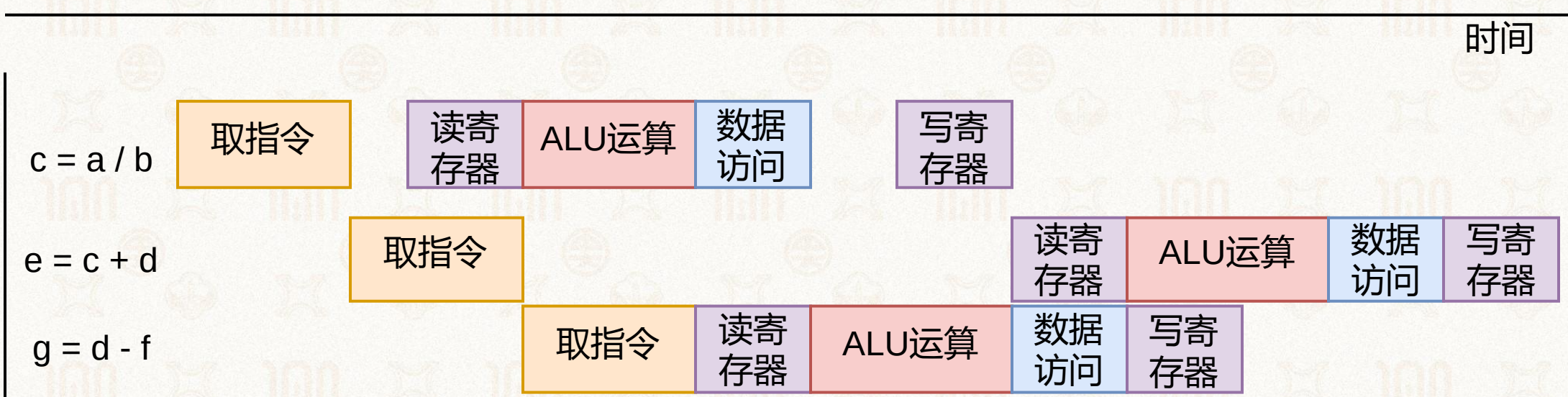

时间
c = a / b
取指令
读寄存器
ALU运算
数据访问
写寄存器
e = c + d
取指令
读寄存器
ALU运算
数据访问
写寄存器
g = d - f
取指令
读寄存器
ALU运算
数据访问
写寄存器
200ps
程序执行顺序

流水线执行



- 指令之间有依赖关系，维持正确性有新挑战：
  - 不同指令运算时间不同
  - 流水线位置不同
- 只能靠等待解决问题

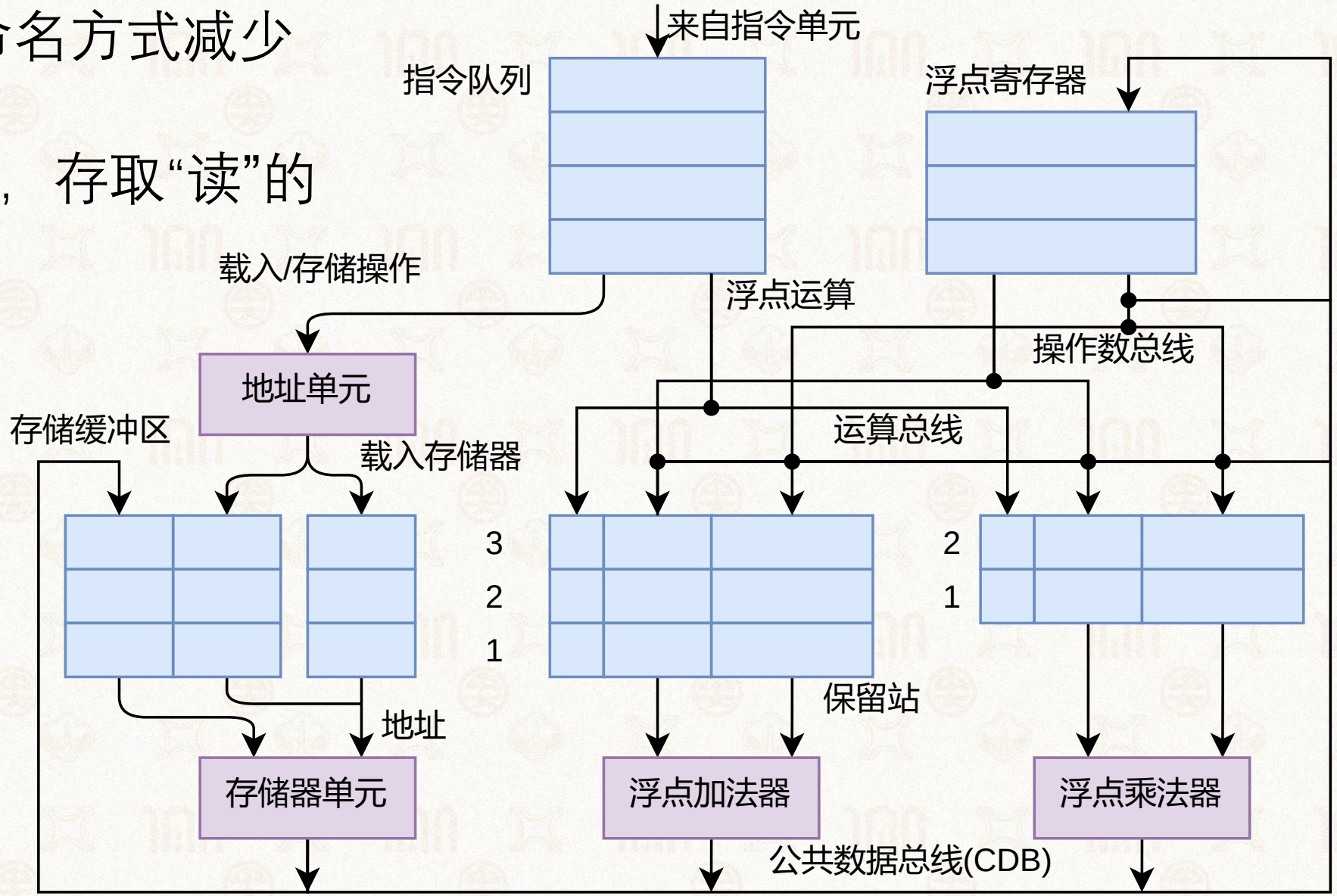
乱序执行



- 乱序执行：指令完成顺序与指令加载顺序并不一致
- 读写顺序不一致，带来很多一致性问题

指令级动态调度：Tomasulo(托马斯洛)算法

- 利用缓存、重命名方式减少“写后读”冲突
- 未完成的“写读”，存取“读”的目标地址



谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫