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II

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

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)
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计

解决临界区问题的算法：皮特森算法

- flag[] 对应位为TRUE, 表示该线程申请进入临界区
- turn 裁决谁可以进入临界区，数字1表示线程1进，0表示线程0进

线程0

```
while (TRUE) {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1] == TRUE && turn == 1);
    临界区部分
    flag[0] = FALSE;
    其它代码
}
```

线程1

```
while (TRUE) {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] == TRUE && turn == 0);
    临界区部分
    flag[1] = FALSE;
    其它代码
}
```

LockOne：皮特森算法的前身

- flag[] 对应位为TRUE, 表示该线程申请进入临界区

- 多核环境下能够互斥访问吗？

线程0

```
while (TRUE) {
    flag[0] = TRUE;

    while (flag[1] == TRUE);
    临界区部分
    flag[0] = FALSE;
    其它代码
}
```

线程1

```
while (TRUE) {
    flag[1] = TRUE;

    while (flag[0] == TRUE);
    临界区部分
    flag[1] = FALSE;
    其它代码
}
```

LockOne：在现实硬件中能够保证互斥访问吗？

- LockOne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

线程0

```
while (TRUE) {
    flag[0] = TRUE;

    while (flag[1] == TRUE);
    临界区部分
    flag[0] = FALSE;
    其它代码
}
```

线程1

```
while (TRUE) {
    flag[1] = TRUE;

    while (flag[0] == TRUE);
    临界区部分
    flag[1] = FALSE;
    其它代码
}
```

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？

- LockOne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

线程0

```
while (TRUE) {
    while (flag[1] == TRUE);
    flag[0] = TRUE;
    临界区部分
    flag[0] = FALSE;
    其它代码
}
```

顺序颠倒

线程1

```
while (TRUE) {
    flag[1] = TRUE;
    while (flag[0] == TRUE);
    临界区部分
    flag[1] = FALSE;
    其它代码
}
```

LockOne：在现实硬件中能够保证互斥访问吗？

- LockOne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

线程0

```
while (TRUE) {
    while (flag[1] == TRUE);
    flag[0] = TRUE;
    临界区部分
    flag[0] = FALSE;
    其它代码
}
```

线程1

```
while (TRUE) {
    while (flag[0] == TRUE);
    flag[1] = TRUE;
    临界区部分
    flag[1] = FALSE;
    其它代码
}
```

T1 T2 T3 T4

开始时都读到了FALSE，同时进入临界区

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)
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计

几种内存模型

- 严格一致性模型
- 顺序一致性模型
- TSO一致性模型
- 弱序一致性模型

强保证

弱保证

内存模型：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 = 1, B = 1

内存模型：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
- 执行结果必须与其中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];
                                                         }
```

内存模型：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
- 执行结果必须与其中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];
                                                                     }
```

多种可能的结果

内存模型：顺序一致性模型

- 不要求操作按照真实发生的时间顺序（全局时钟）全局可见
- 执行结果必须与其中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

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

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

实际发生顺序（全局时钟，上帝视角）

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

内存模型：顺序一致性模型

- 不要求操作按照真实发生的时间顺序（全局时钟）全局可见
- 执行结果必须与其中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

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

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

实际发生顺序（全局时钟，上帝视角）

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

A = 0, B = 1

内存模型：顺序一致性模型

- 不要求操作按照真实发生的时间顺序（全局时钟）全局可见
- 执行结果必须与其中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

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

实际发生顺序（全局时钟，上帝视角）

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

内存模型：顺序一致性模型

- 不要求操作按照真实发生的时间顺序（全局时钟）全局可见
- 执行结果必须与其中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

```
flag[0] = 1;

                flag[1] = 1;

A = flag[1];

                B = flag[0];
```

实际发生顺序（全局时钟，上帝视角）

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

A = 1, B = 1

内存模型：顺序一致性模型

- 不要求操作按照真实发生的时间顺序（全局时钟）全局可见
- 执行结果必须与其中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

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

实际发生顺序（全局时钟，上帝视角）

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

内存模型：顺序一致性模型

- 不要求操作按照真实发生的时间顺序（全局时钟）全局可见
- 执行结果必须与其中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

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

实际发生顺序（全局时钟，上帝视角）

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

A = 1, B = 0

内存模型：顺序一致性模型

- 不要求操作按照真实发生的时间顺序（全局时钟）全局可见
- 执行结果必须与其中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：

全局顺序

```
                    proc_B的程序顺序被打破了
                    B = flag[0];
flag[0] = 1;
A = flag[1];
                    flag[1] = 1;
```

实际发生顺序（全局时钟，上帝视角）

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

内存模型：顺序一致性模型

- 不要求操作按照真实发生的时间顺序（全局时钟）全局可见
- 执行结果必须与其中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：

全局顺序

```
                    proc_B的程序顺序被打破了
                    B = flag[0];
flag[0] = 1;
A = flag[1];
                    flag[1] = 1;
```

实际发生顺序（全局时钟，上帝视角）

```
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

内存模型：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

- 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      A = flag[1];     读
T2                                                           B = flag[0]; 读
T3      flag[0] = 1;     写
T4  }                                                        flag[1] = 1; 写
                                                         }
```

实际执行的结果有可能是这样颠倒的顺序!

内存模型：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

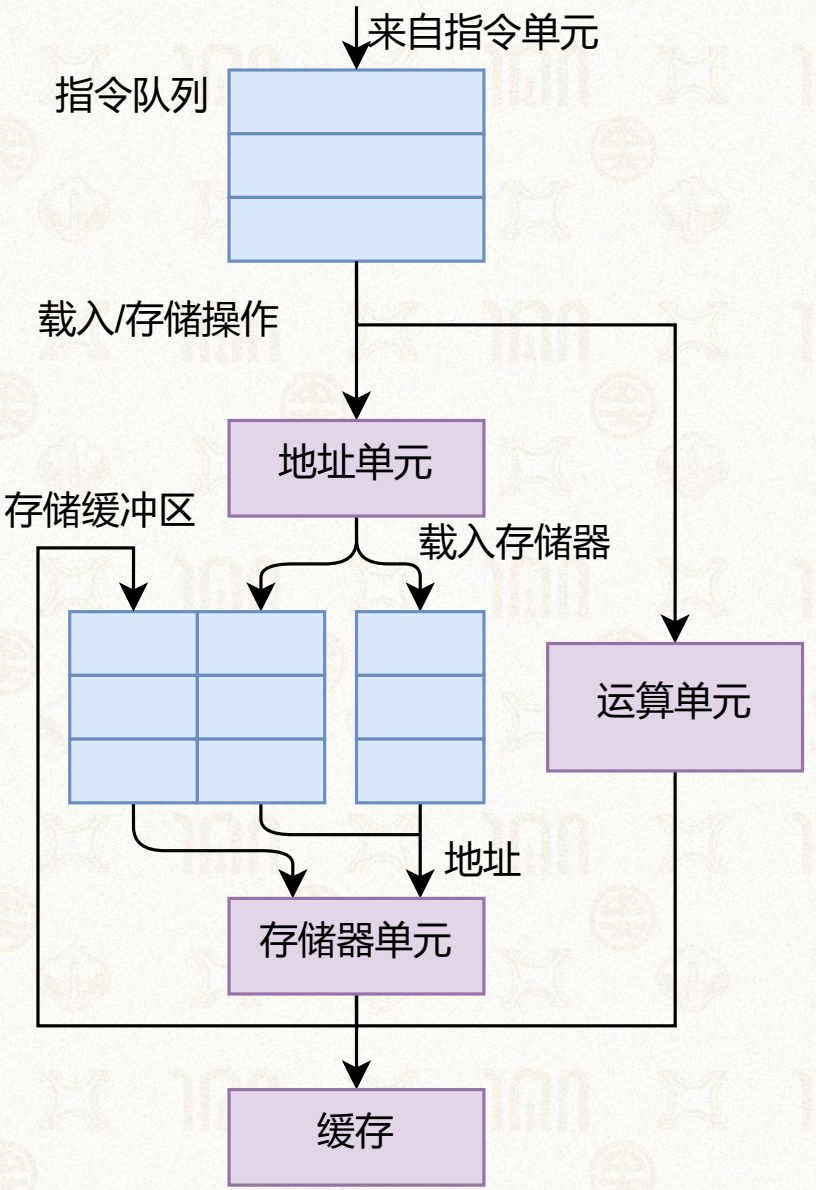
初始值：　flag[0] = 0;　　flag[1] = 0;

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

最终允许的结果：(A, B) = (0, 0)

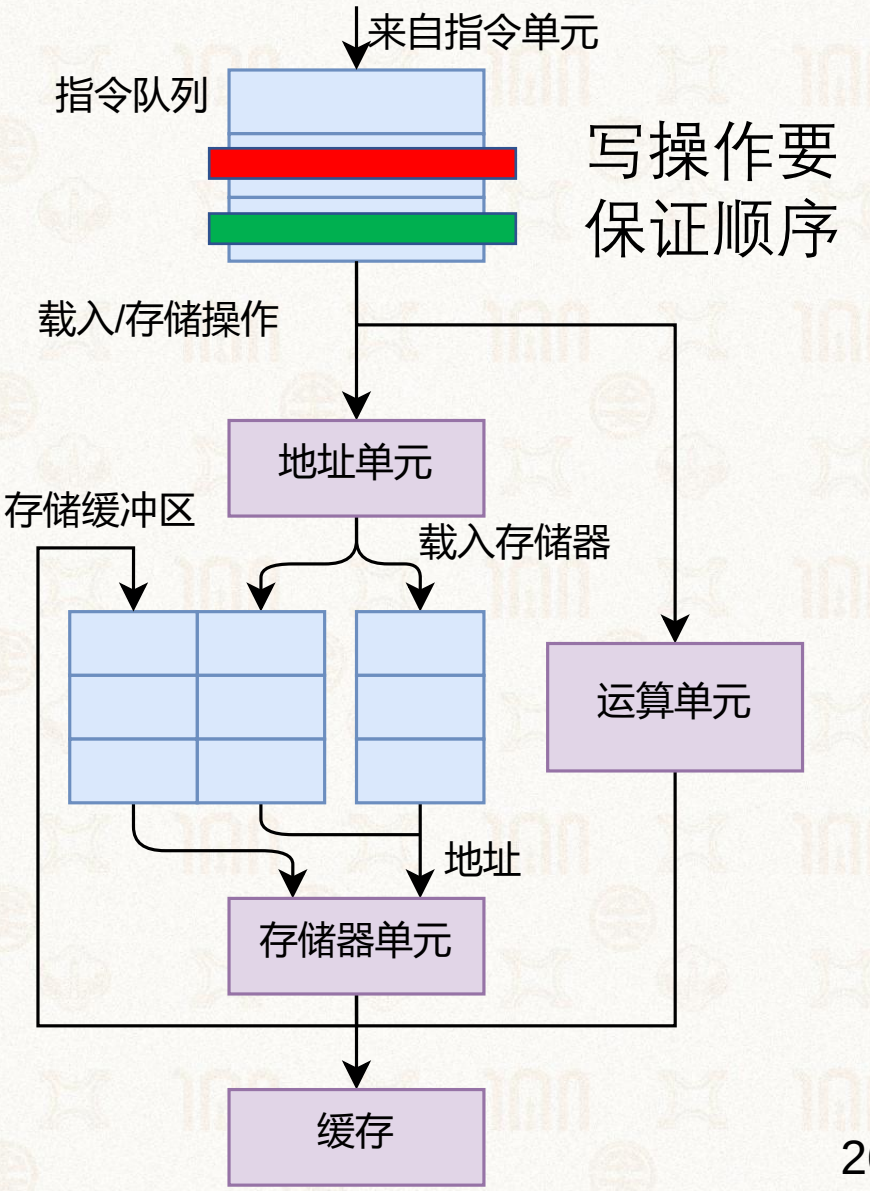
内存模型：TSO一致性模型

- 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）



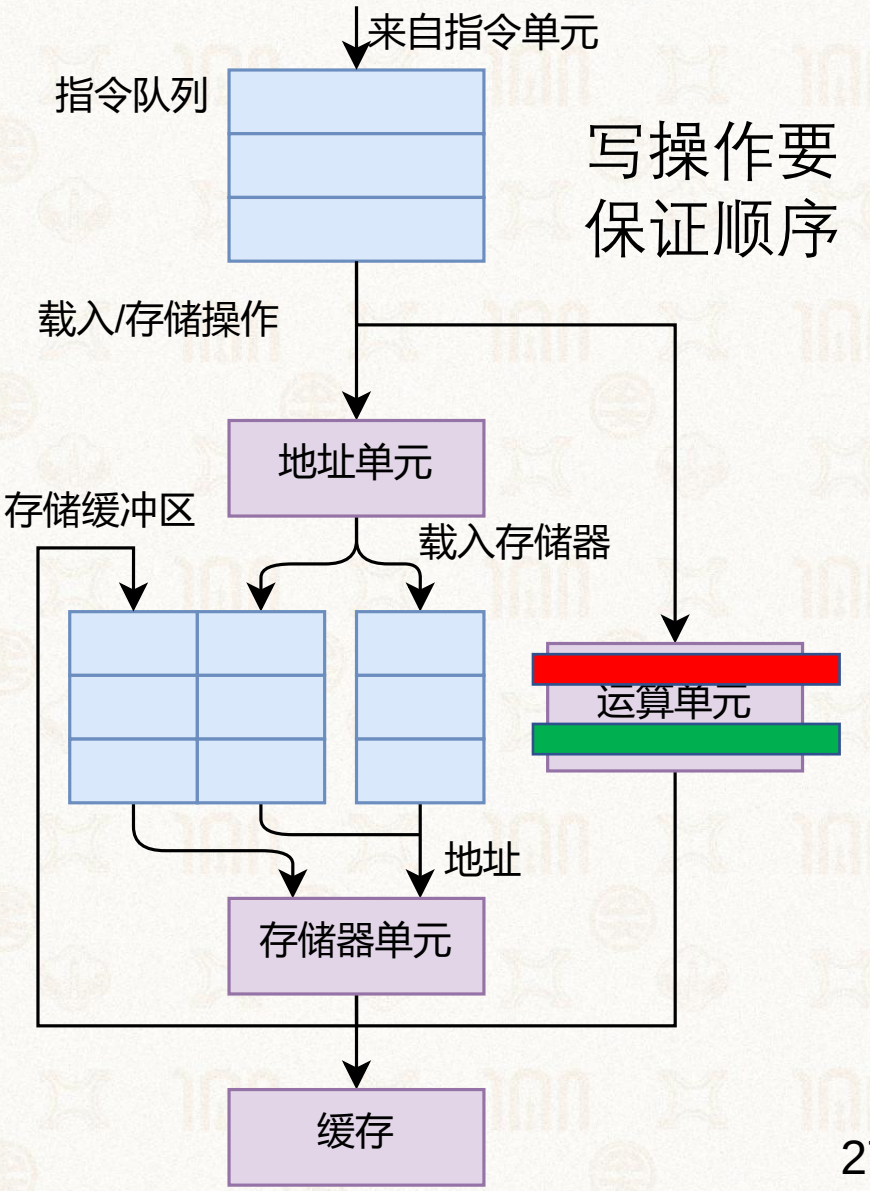
内存模型：TSO一致性模型

- 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）



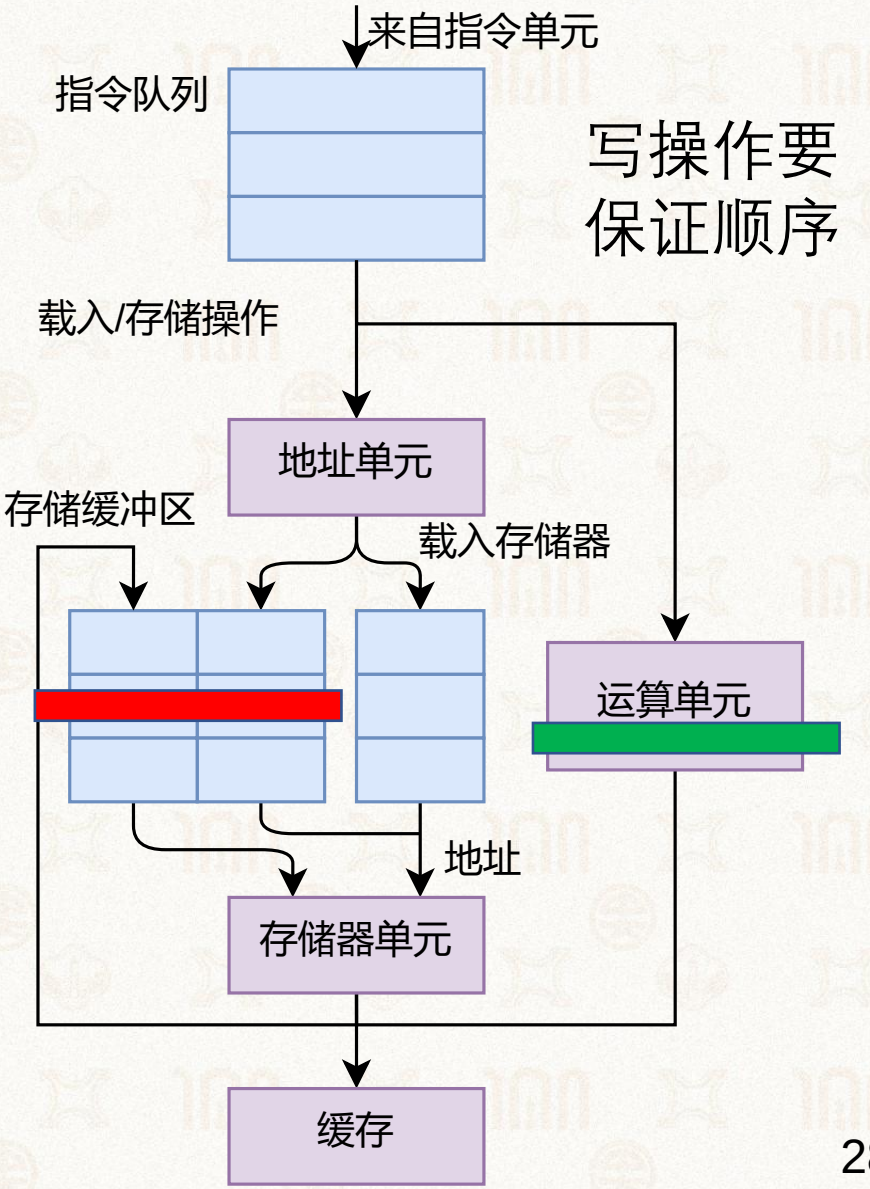
内存模型：TSO一致性模型

- 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）



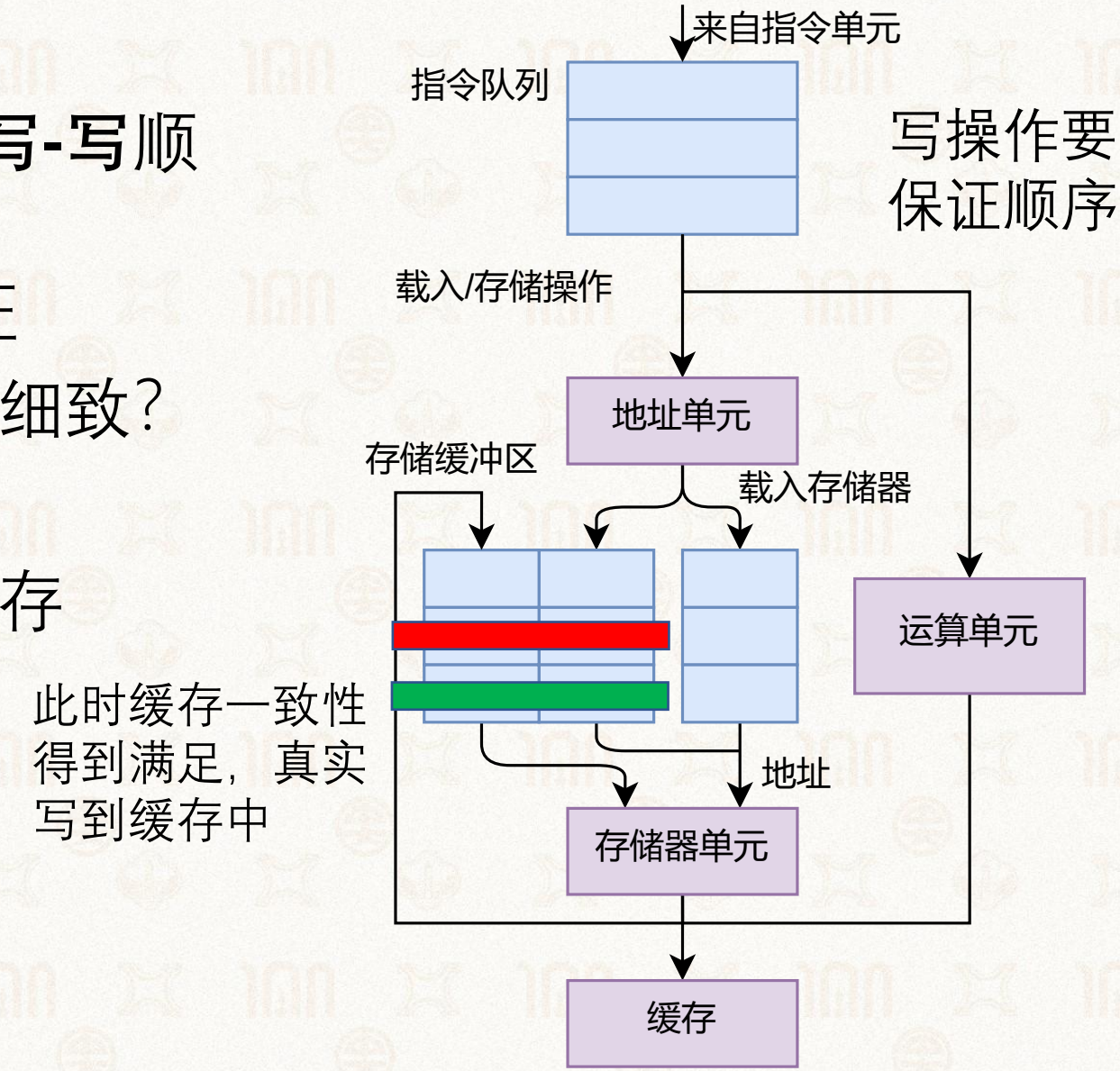
内存模型：TSO一致性模型

- 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）



内存模型：TSO一致性模型

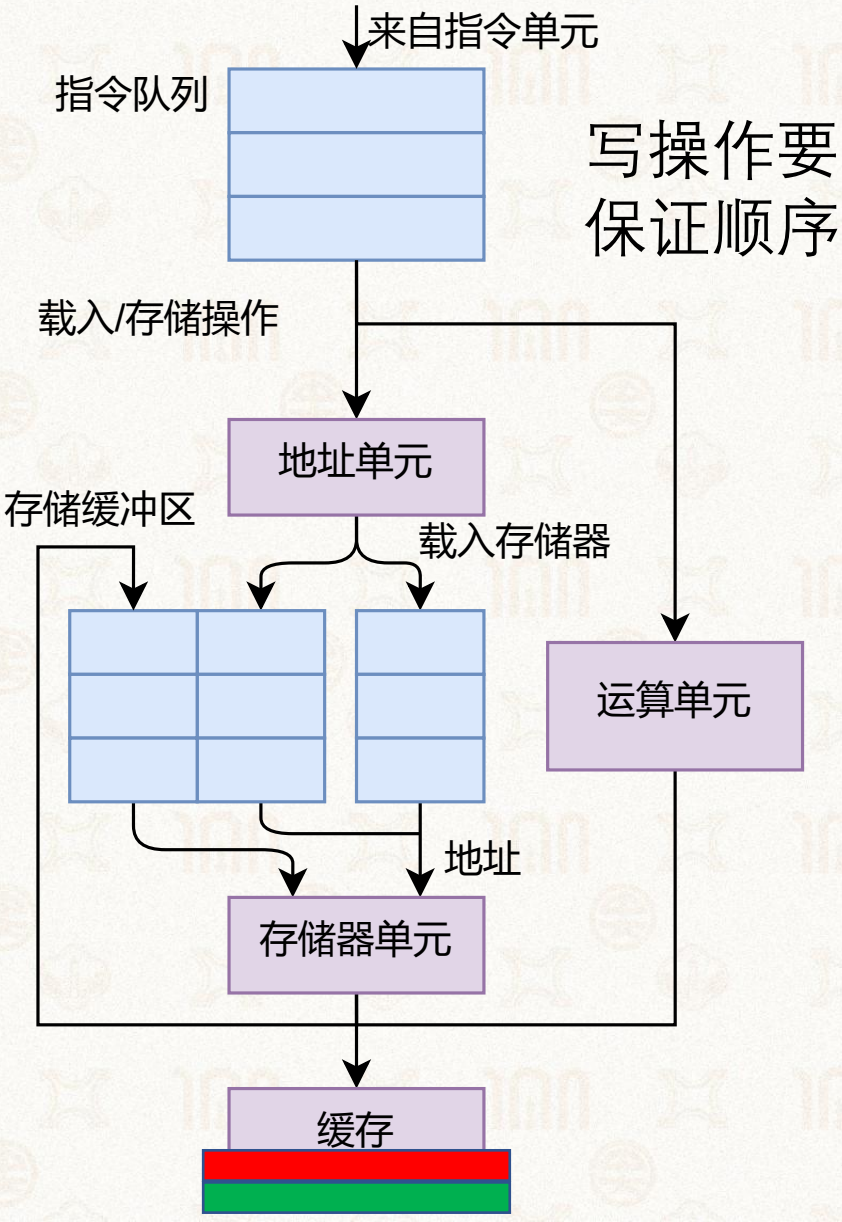
- Total Store Ordering
- 针对**不同地址**的**读-读**、**读-写**、**写-写**顺序都能得到保证
- 只有写-读的顺序不能够得到保证
- 为什么看起来很奇怪？规定这么细致？
- 需要允许乱序执行来提升性能！
- 要保证顺序：使用了内存保序缓存（Memory Ordering Buffer）



此时缓存一致性得到满足，真实写到缓存中

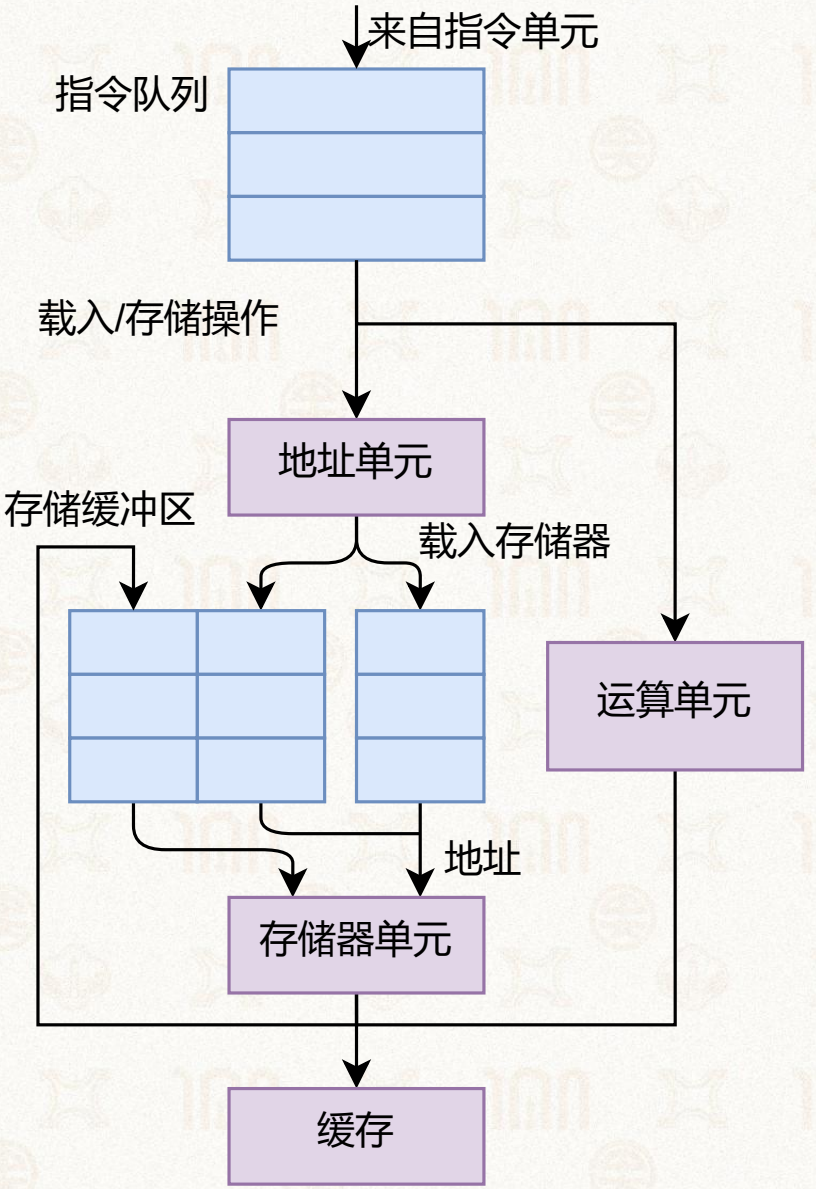
内存模型：TSO一致性模型

- 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）



内存模型：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：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

- 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);
}
```

单选题 1分

设置

SUN YAT-SEN UNIVERSITY

TSO模型的电脑是否会出现以下实际的执行流程？为什么？

这是程序员写的代码：

```
void proc_A(void) {
    data = 666;

    flag = READY;
}
```

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

实际执行流程是否可以这样：

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

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

T1

```
    data = 666;
}
```

T1时刻，proc_B读到错误的值

TSO下程序不会出错

TSO下程序会出错

提交

内存模型：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

- 弱序一致性模型
  - 移动场景，成本控制
  - 考虑CPU复杂度
  - 考虑功耗表现
  - ARM架构
  - PowerPC架构

- TSO一致性模型
  - “高性能”
  - CPU更复杂
  - x86/64架构
    - Intel / AMD CPU

如何在弱的内存模型中保证顺序

- 通常的做法：添加硬件内存屏障 （barrier/fence）
  - 任何访存操作不会逾越内存屏障

```
void proc_A(void) {
    flag[0] = 1;
        ↓ 保证写-读顺序
    while(flag[1]);
}
```

```
void proc_B(void) {
    flag[1] = 1;
        ↓ 保证写-读顺序
    while(flag[0]);
}
```

```
void proc_A(void) {
    data = 666;
        ↓ 保证写-写顺序
    flag = READY;
}
```

```
void proc_B(void) {
    while (flag != READY) ;
        ↓ 保证读-读顺序
    handle(data);
}
```

如何在弱的内存模型中保证顺序

- 通常的做法：添加硬件内存屏障 （barrier/fence）
  - 任何访存操作不会逾越内存屏障

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

↓ 保证写-读顺序

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

↓ 保证写-读顺序

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

↓ 保证写-写顺序

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

↓ 保证读-读顺序

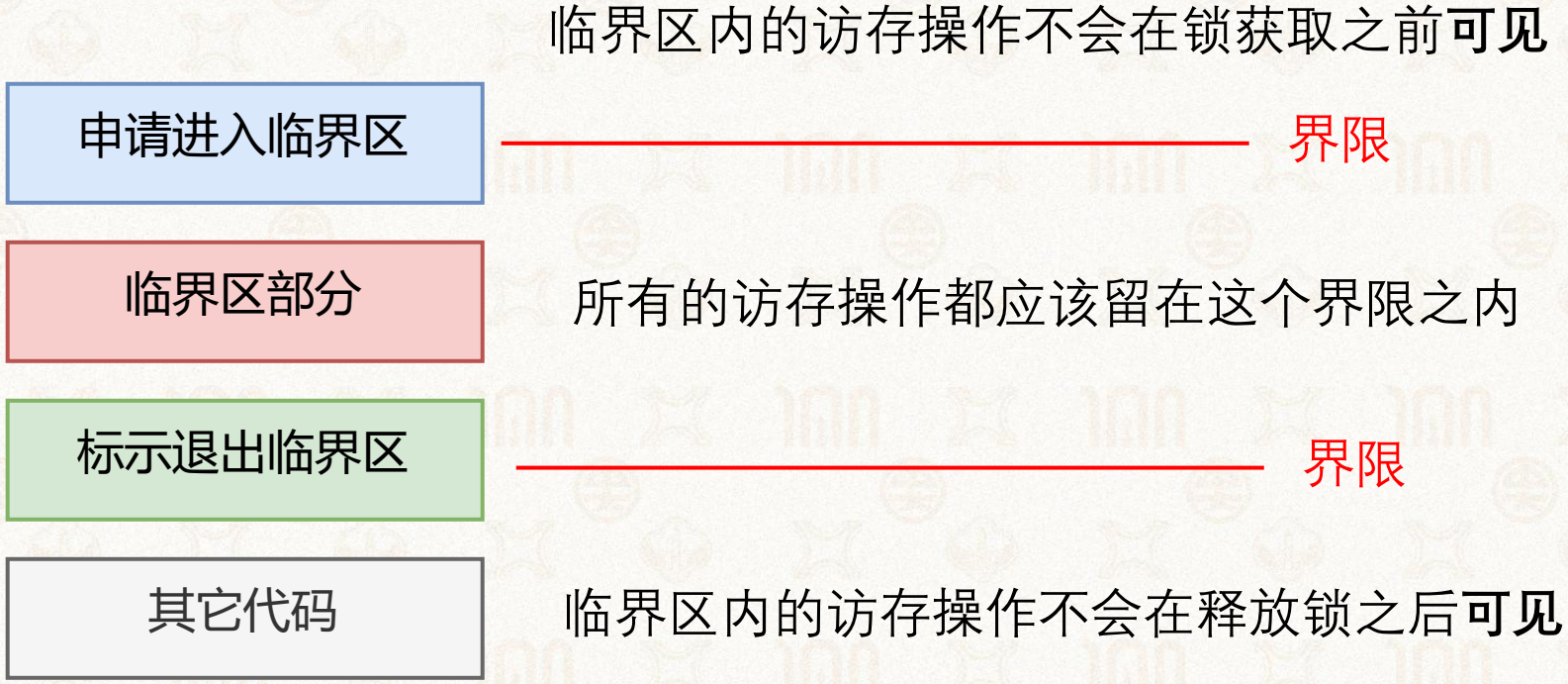
如何在弱的内存模型中保证顺序

- 这些讨论都只针对没有显式依赖关系的访存操作

- 关于有依赖关系的访存操作，想想托马斯洛算法

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

大纲

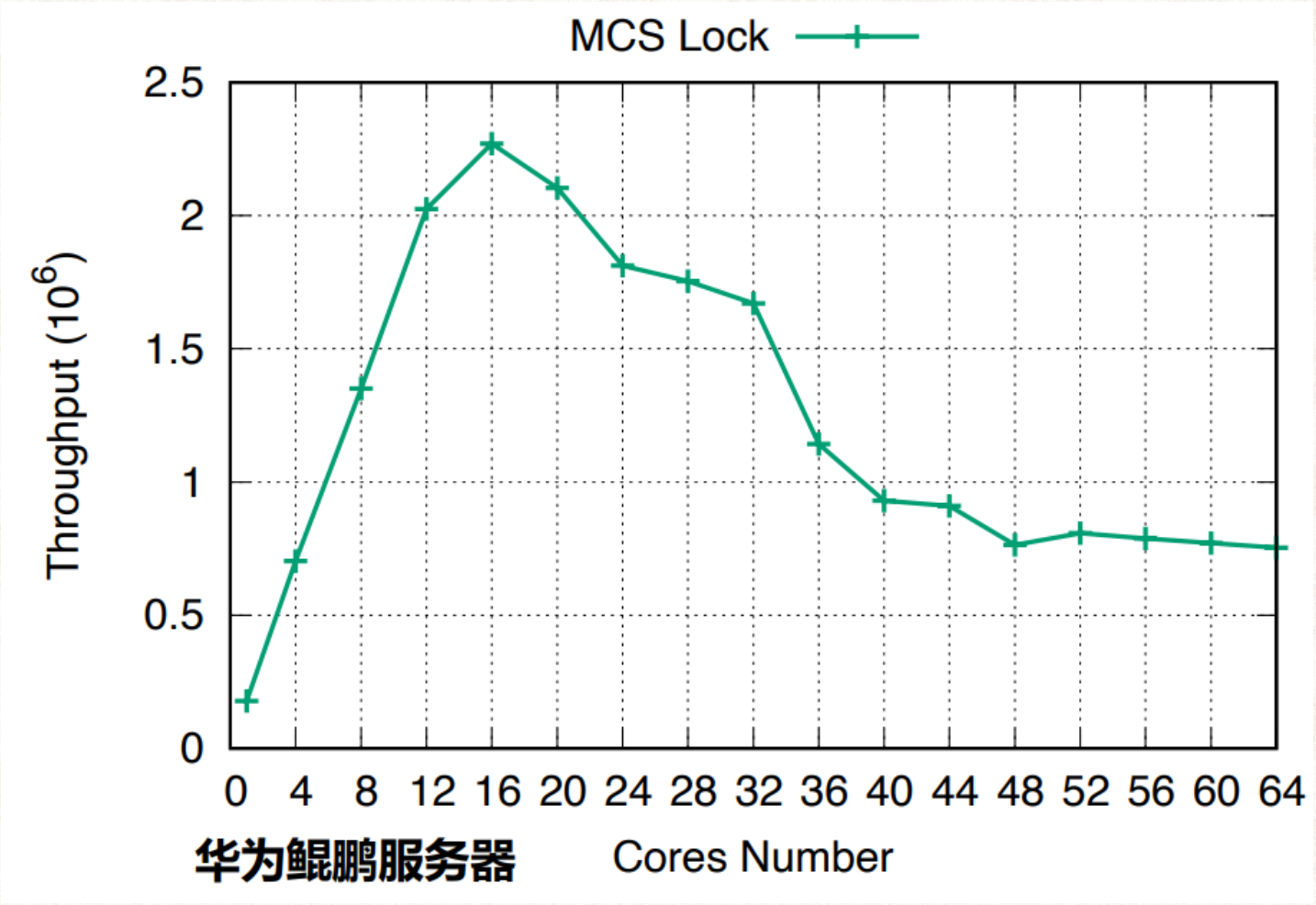
- 多核性能问题
- 缓存一致性
  - 状态迁移
  - 基于目录项的缓存一致性
- 多核性能可扩展性
  - 性能低下原因
  - 回退锁
  - MCS锁
  - 对程序员的启发
- 内存一致性铺垫
  - 死锁预防(复习)
  - 乱序执行(补课+超纲)
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计

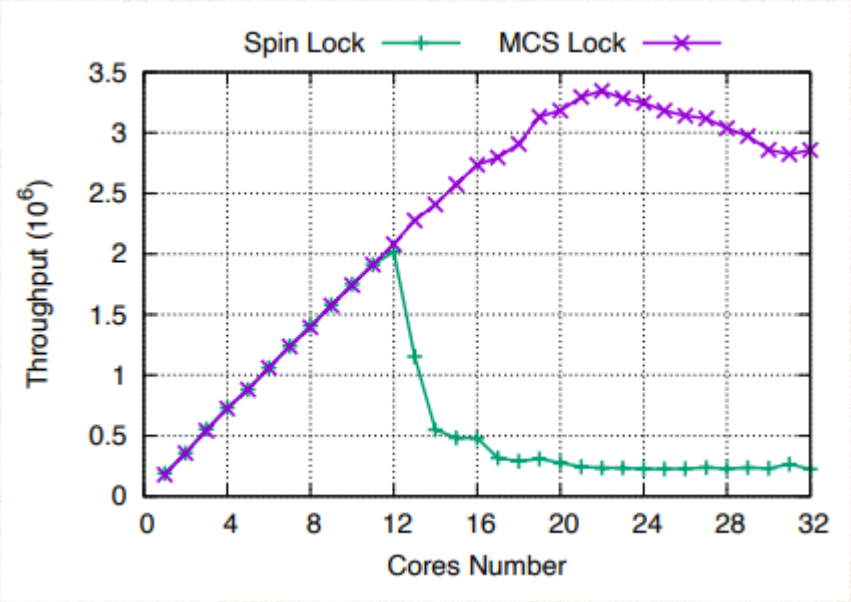
互斥锁微基准测试

```
struct lock *glock;
unsigned long gcnt = 0;
char shared_data[CACHE_LINE_SIZE * 10];
void *thread_routine(void *arg) {
    while (1) {
        lock(glock);
        // 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, 10);
        unlock(glock);
        interval();
    }
}
```

在临界区访问更多缓存行，是否会影响 mcs 锁的可扩展性？

MCS锁在更多缓存内容上的表现

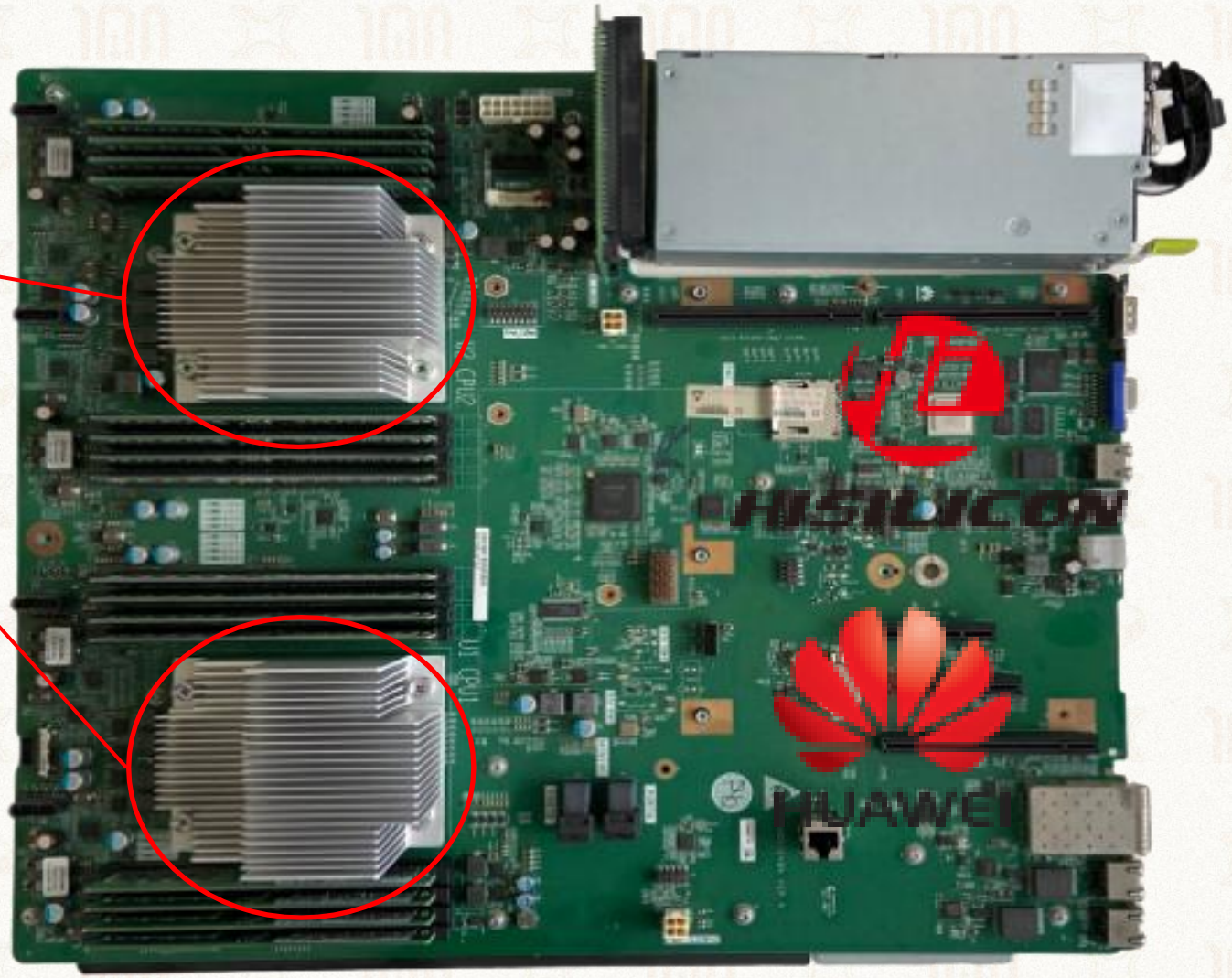




- 与之前测试有两点不同:
  - 1. 临界区访问共享缓存行数量 1 =>10
  - 2. 测试核心数扩大到64个核心

鲲鹏服务器

2*处理器



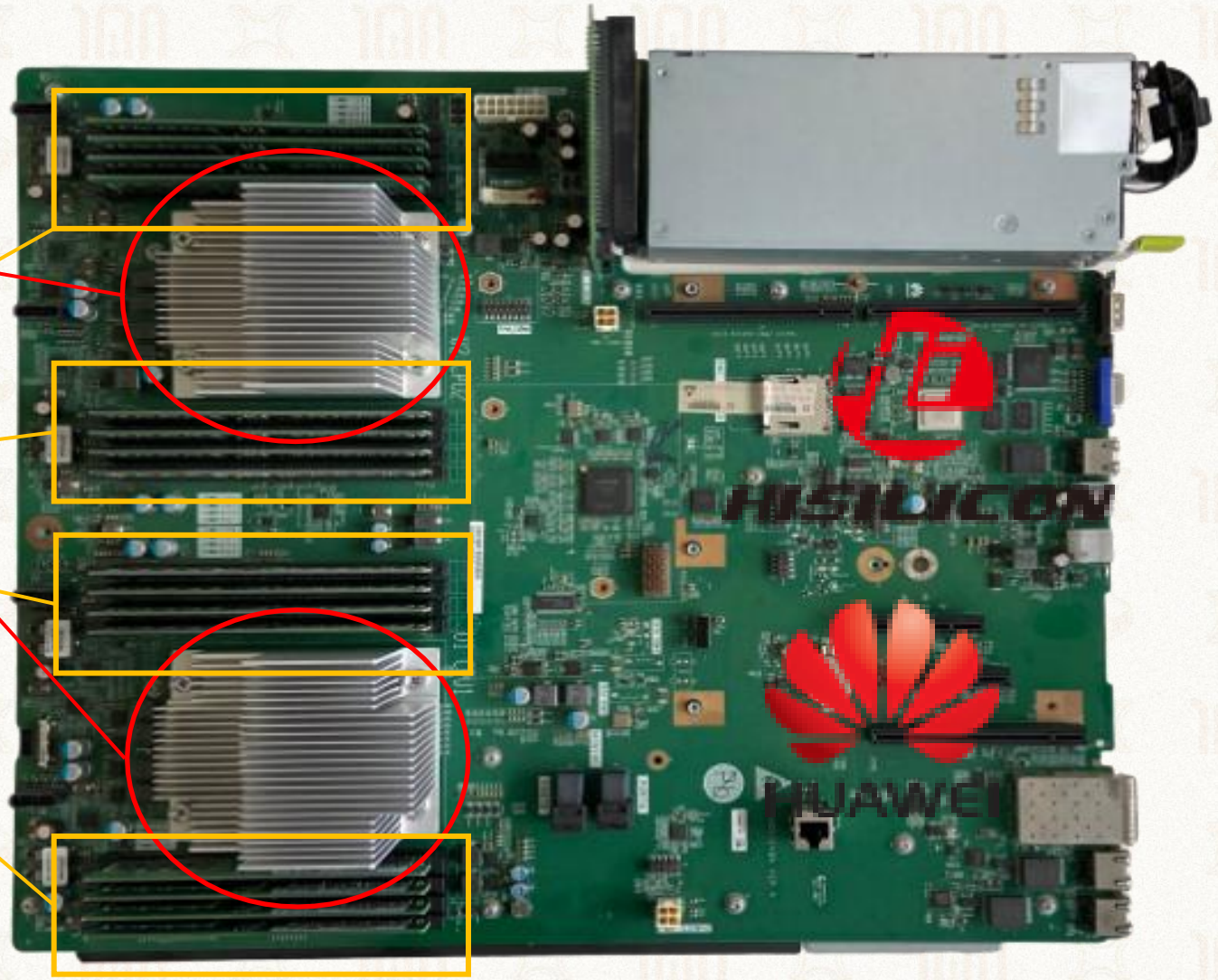
华为鲲鹏服务器（ARM架构）

常见于多处理器（多插槽）机器

鲲鹏服务器

2*处理器

内存条

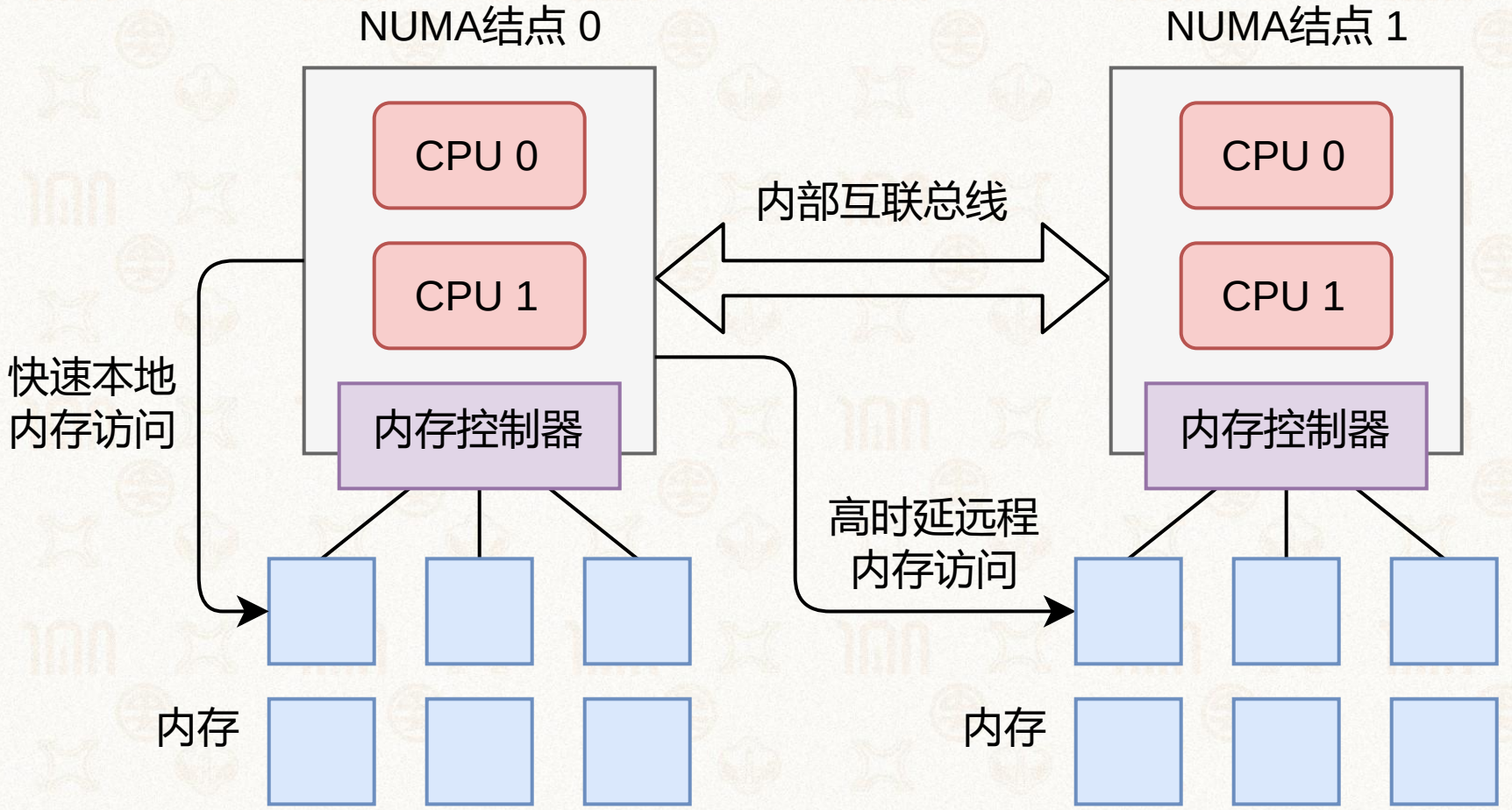


常见于多处理器（多插槽）机器

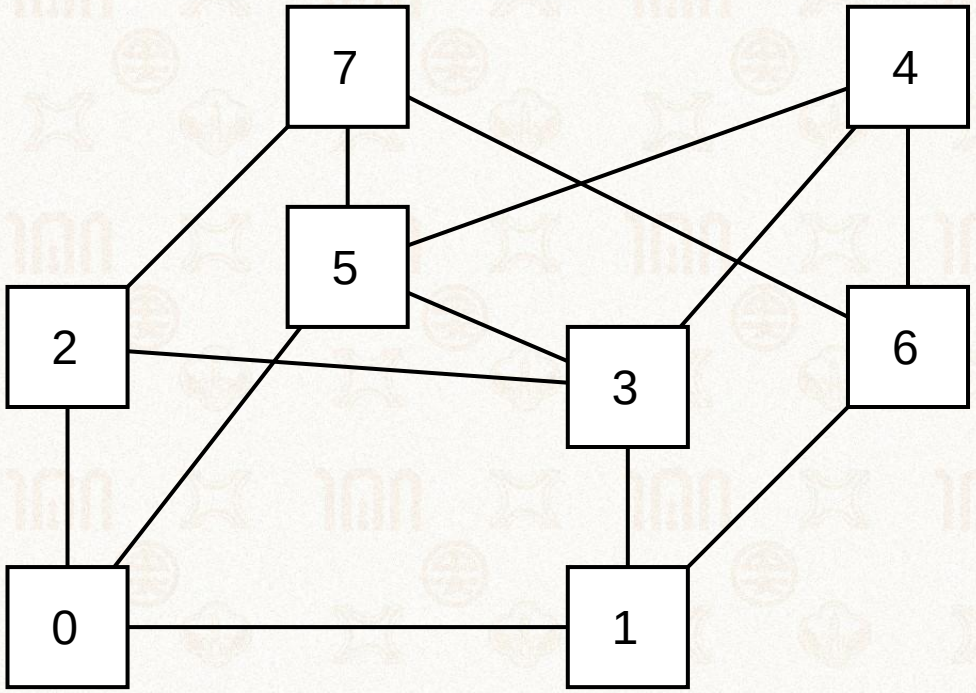
华为鲲鹏服务器（ARM架构）

非一致内存访问(NUMA)

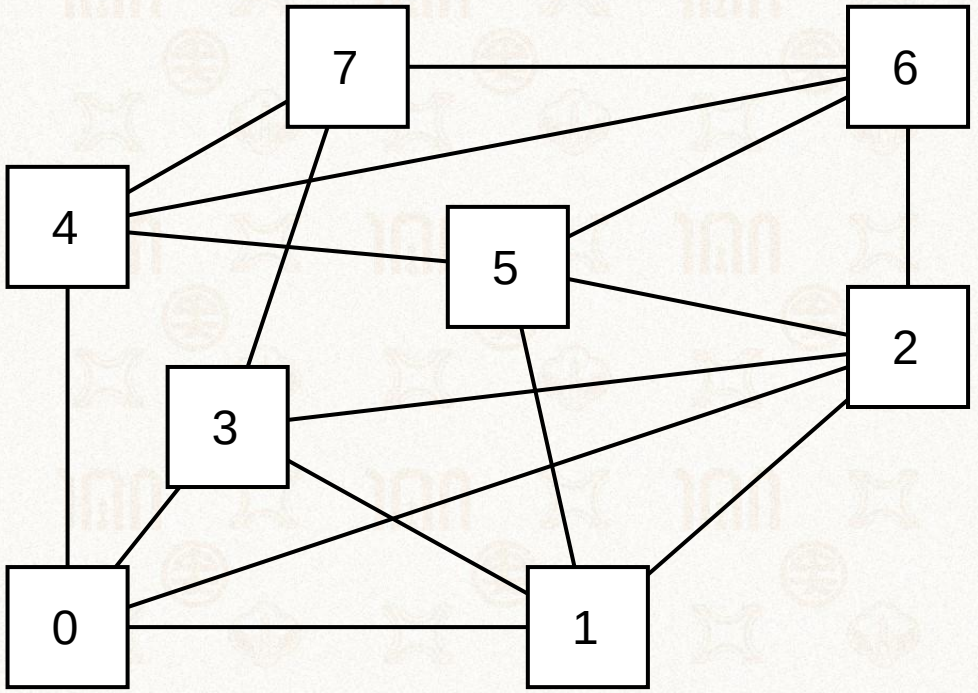
- Non-Uniform Memory Access
- 避免单内存控制器成为瓶颈，减少内存访问距离
- 常见于多处理器（多插槽）机器
- 单处理器众核系统也有可能使用，如Intel Xeon Phi



Intel与AMD的NUMA系统架构与特性



Intel Xeon



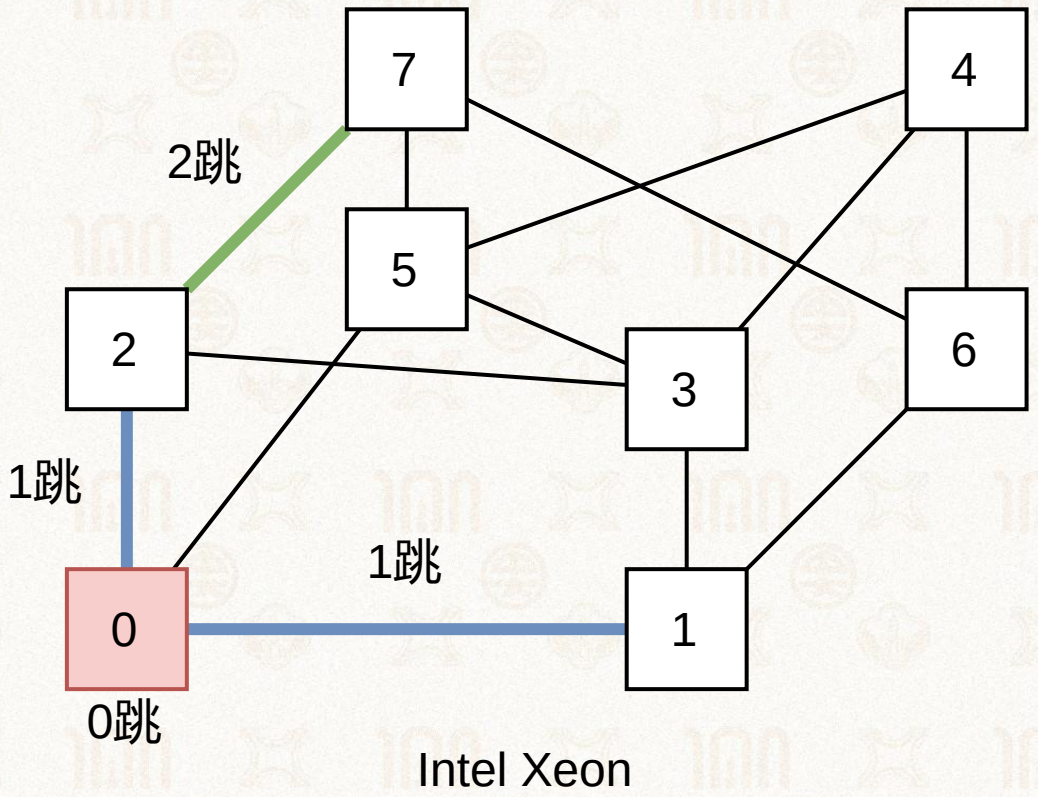
AMD Opteron

Intel与AMD多插槽NUMA架构

结构复杂

查看NUMA结构的命令： numactl --hardware

Intel与AMD的NUMA系统架构与特性



Intel Xeon

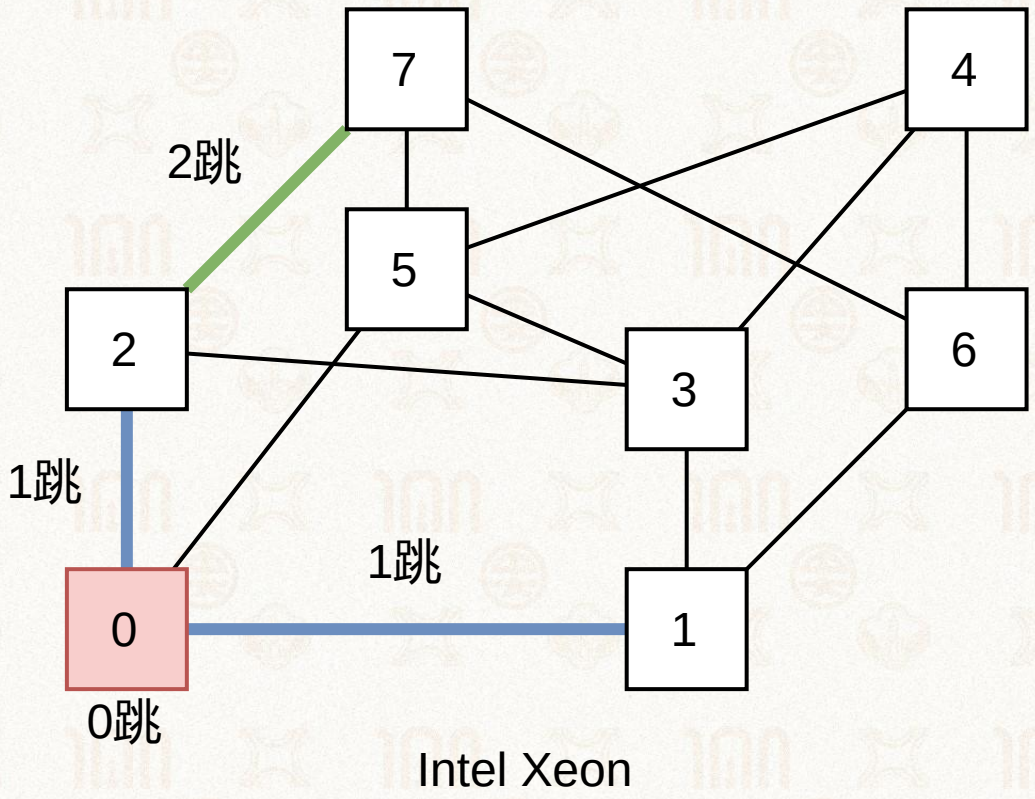
Inst.	0-hop	1-hop	2-hop
80 核心 Intel Xeon 服务器			
Load	117	271	372
Store	108	304	409
64 核心 AMD Opteron 服务器			
Load	228	419	498
Store	256	463	544

远程内存访问（cycles）时延特性

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



Intel Xeon

表 12.5: 内存访问带宽 (MB/s)

Access	0-hop	1-hop	2-hop	Interleaved
80 核 Intel Xeon 服务器				
Sequential	3207	2455	2101	2333
Random	720	348	307	344
64 核 AMD Opteron 服务器				
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

除了锁的元数据，主要是
临界区中访问的共享数据

```
while (TRUE) {
```

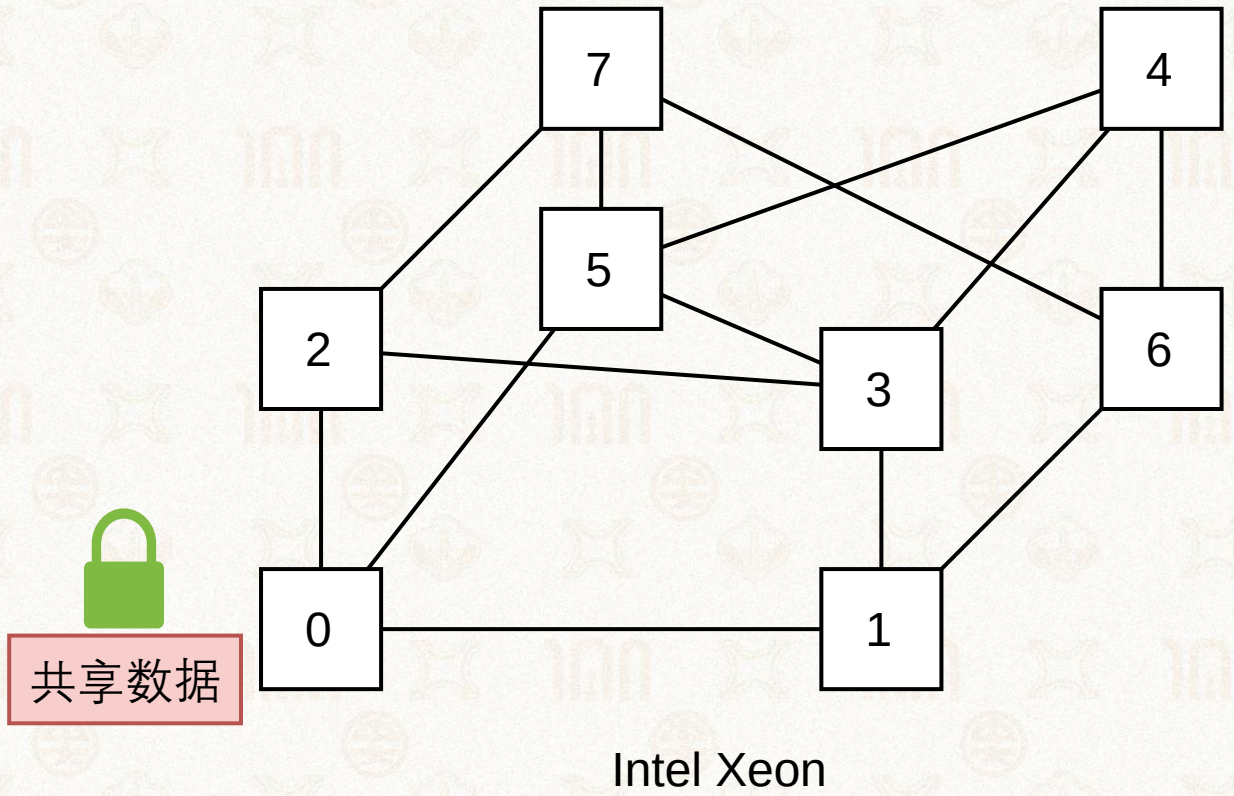
申请进入临界区

临界区部分

标示退出临界区

其它代码

```
}
```



Intel Xeon

NUMA环境中新的挑战

除了锁的元数据，主要是
临界区中访问的共享数据

```
while (TRUE) {
```

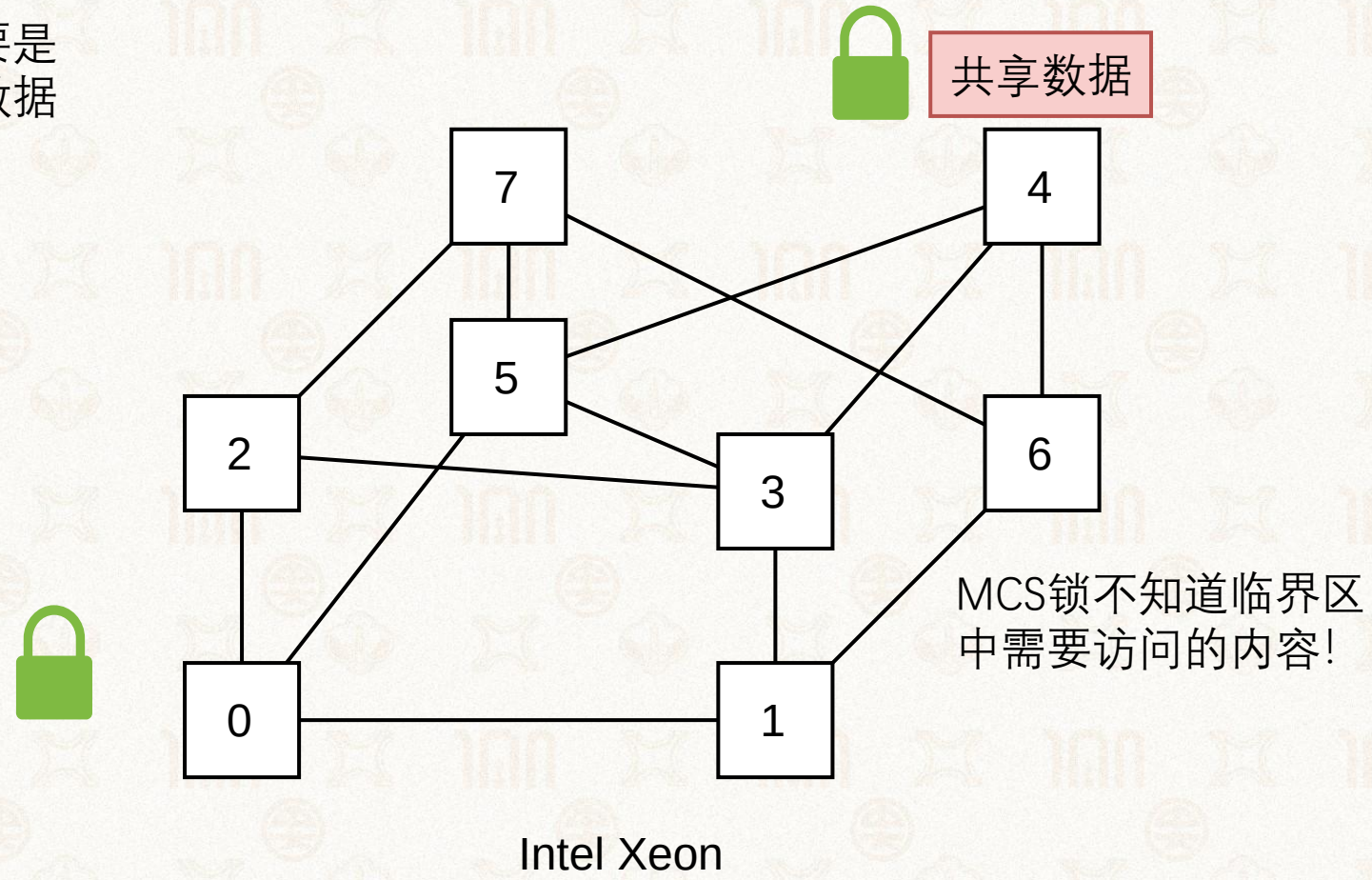
申请进入临界区

临界区部分

标示退出临界区

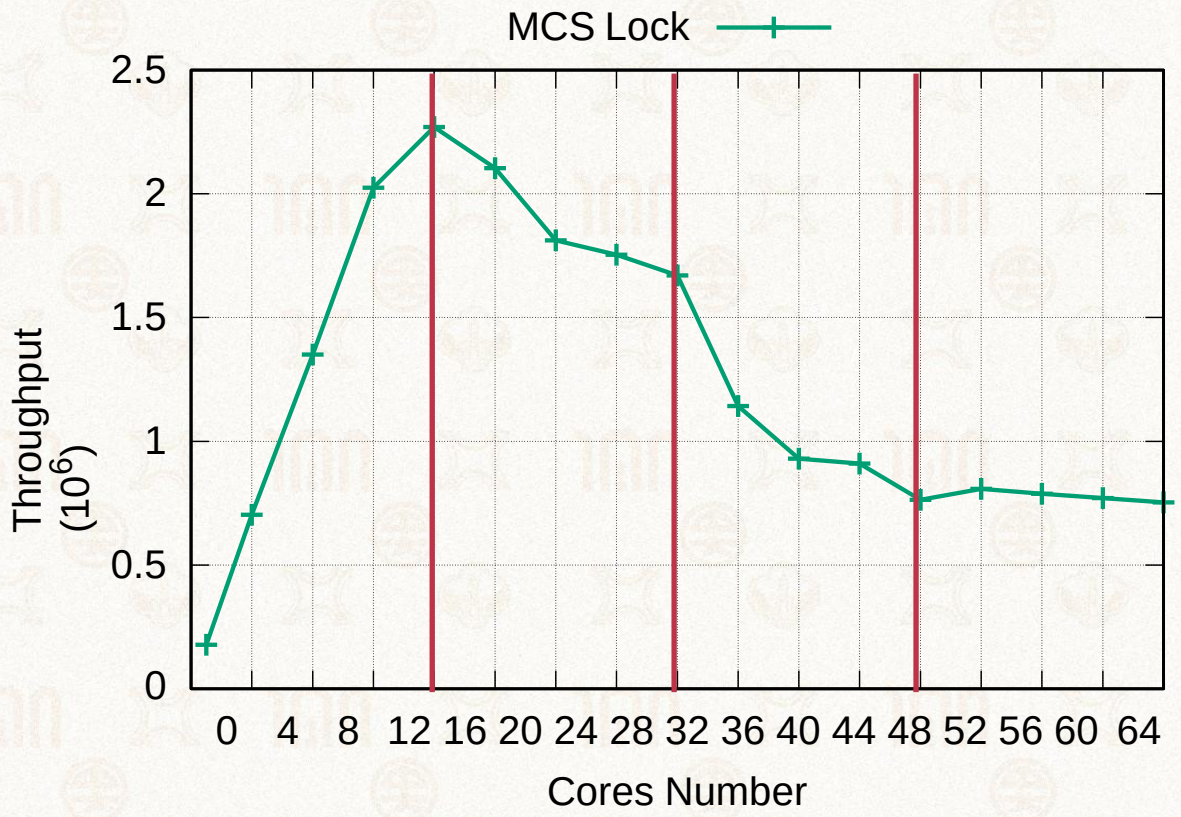
其它代码

```
}
```



跨结点的缓存一致性协议开销巨大

MCS锁可扩展性



鲲鹏服务器

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

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)
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计

NUMA感知设计：以cohort锁为例

- 核心思路：在一段时间内将访存限制在本地

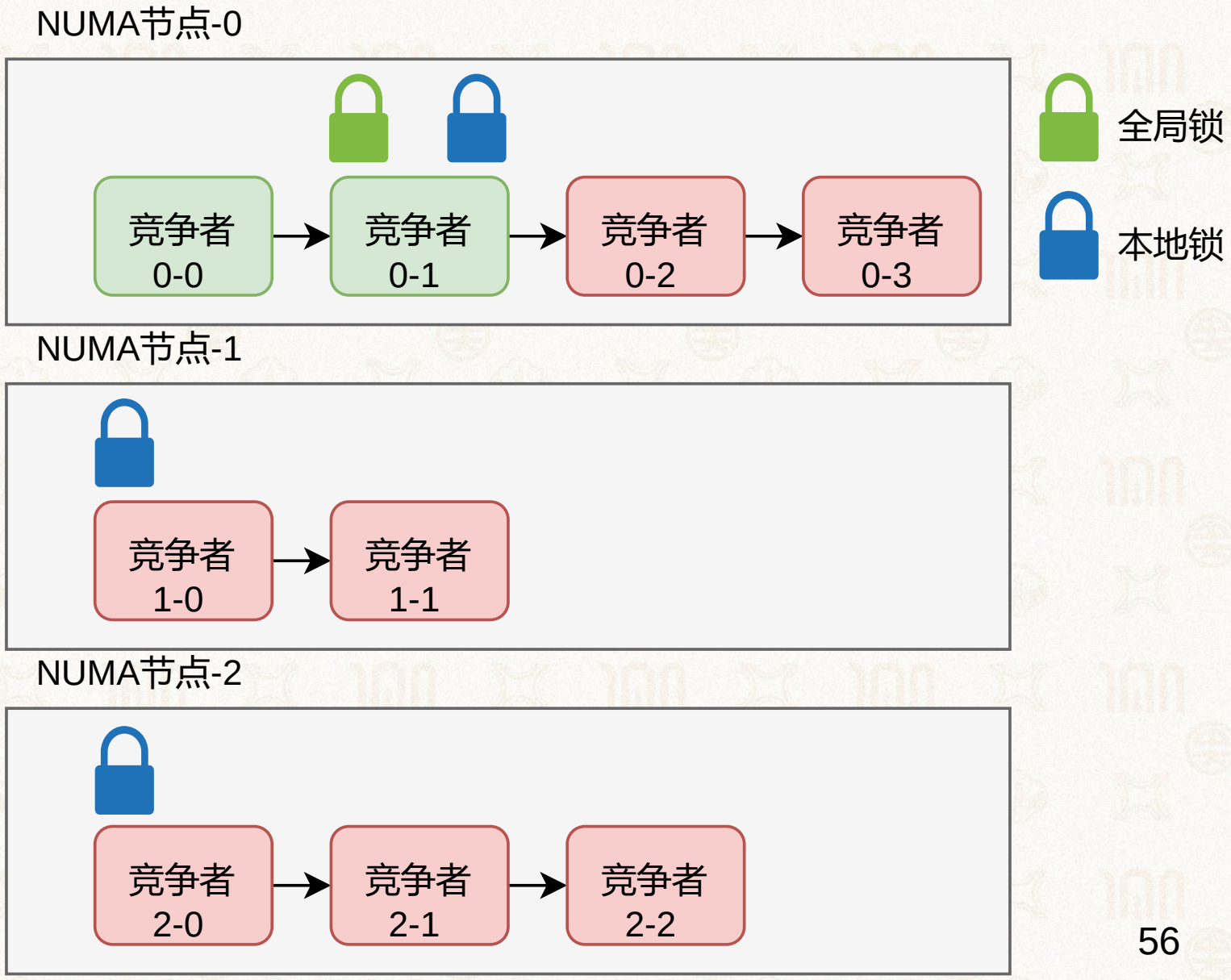
- 先获取每结点本地锁
- 再获取全局锁

- 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

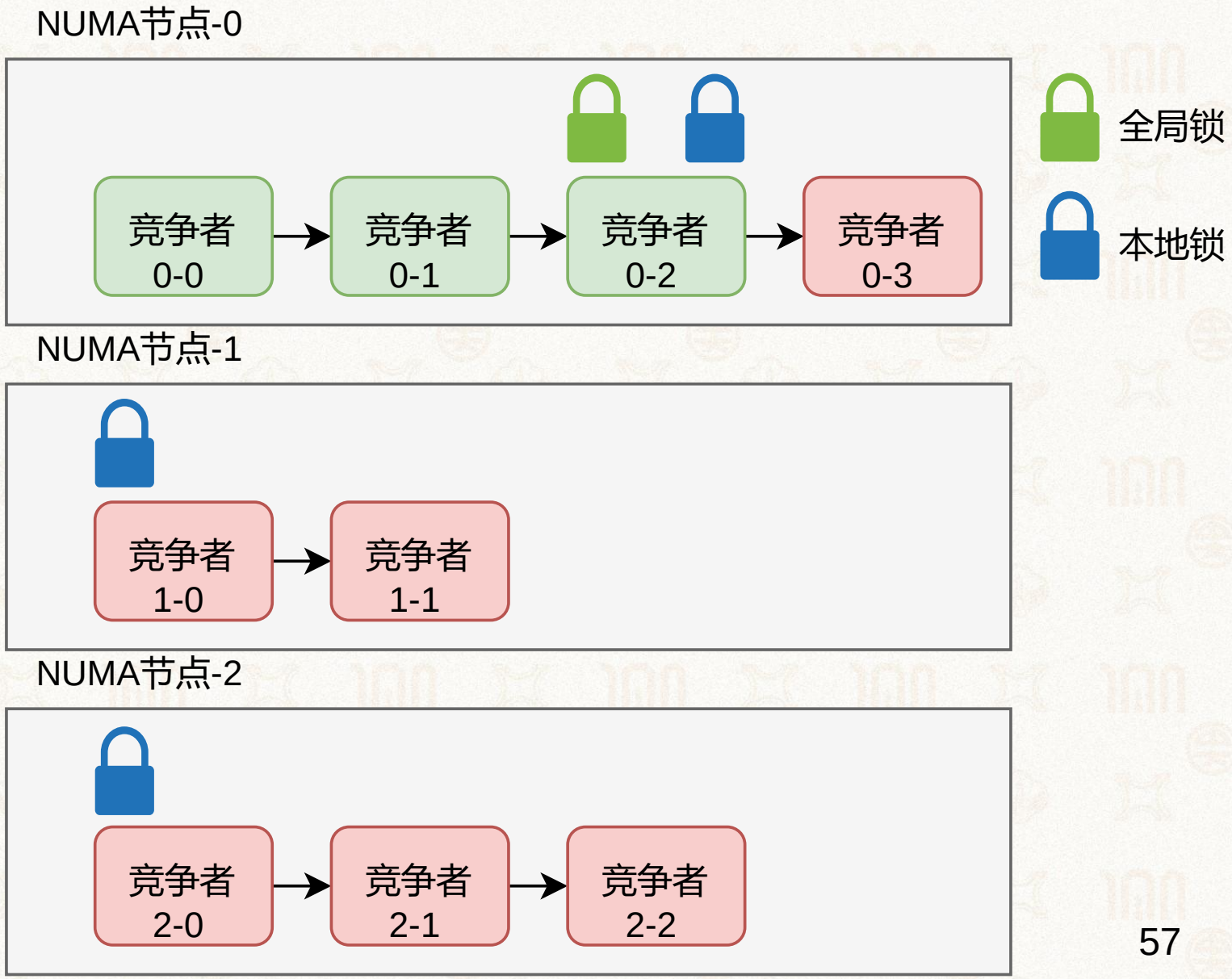
NUMA感知设计：以cohort锁为例

- 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



NUMA感知设计：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



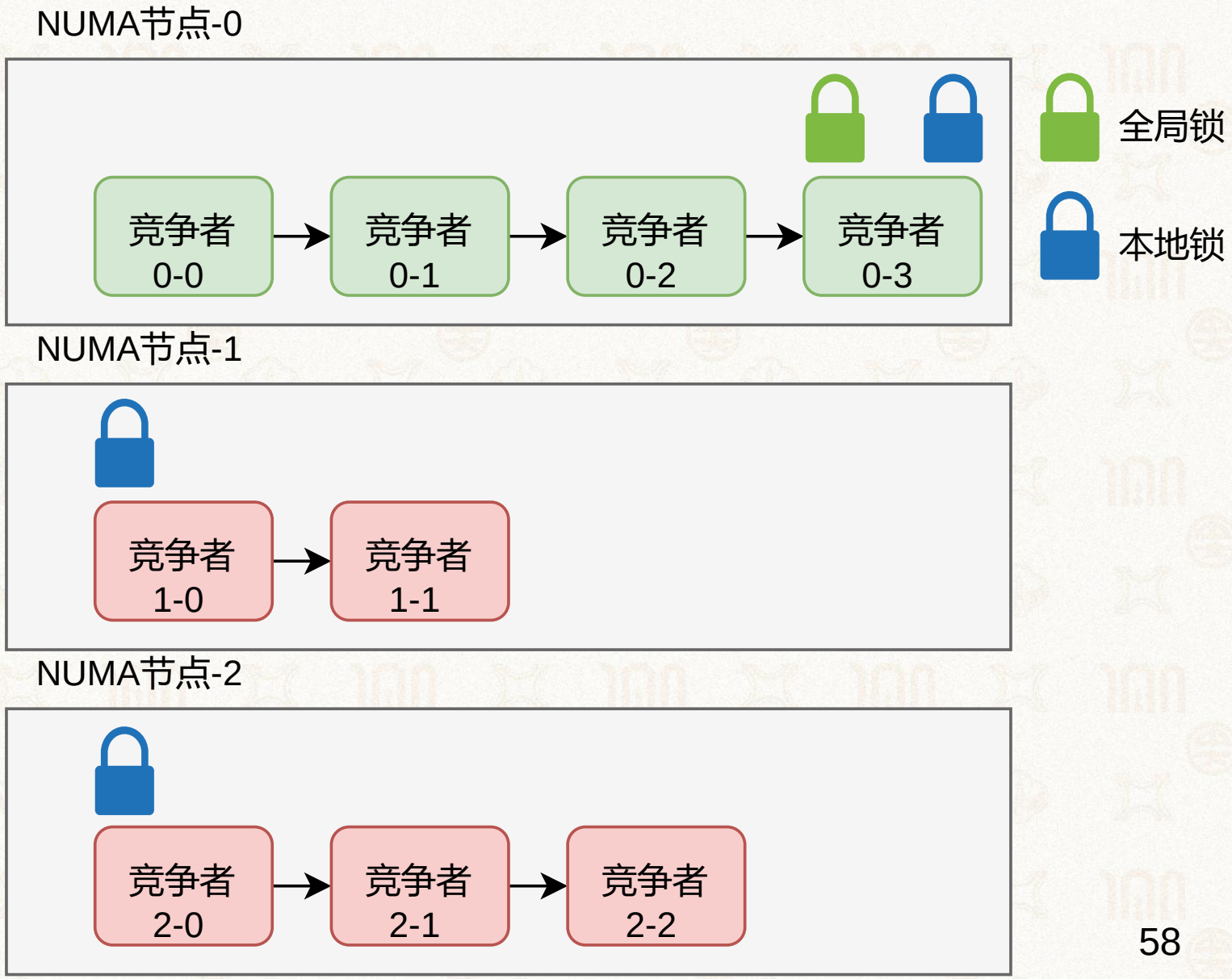
NUMA感知设计：以cohort锁为例

- 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

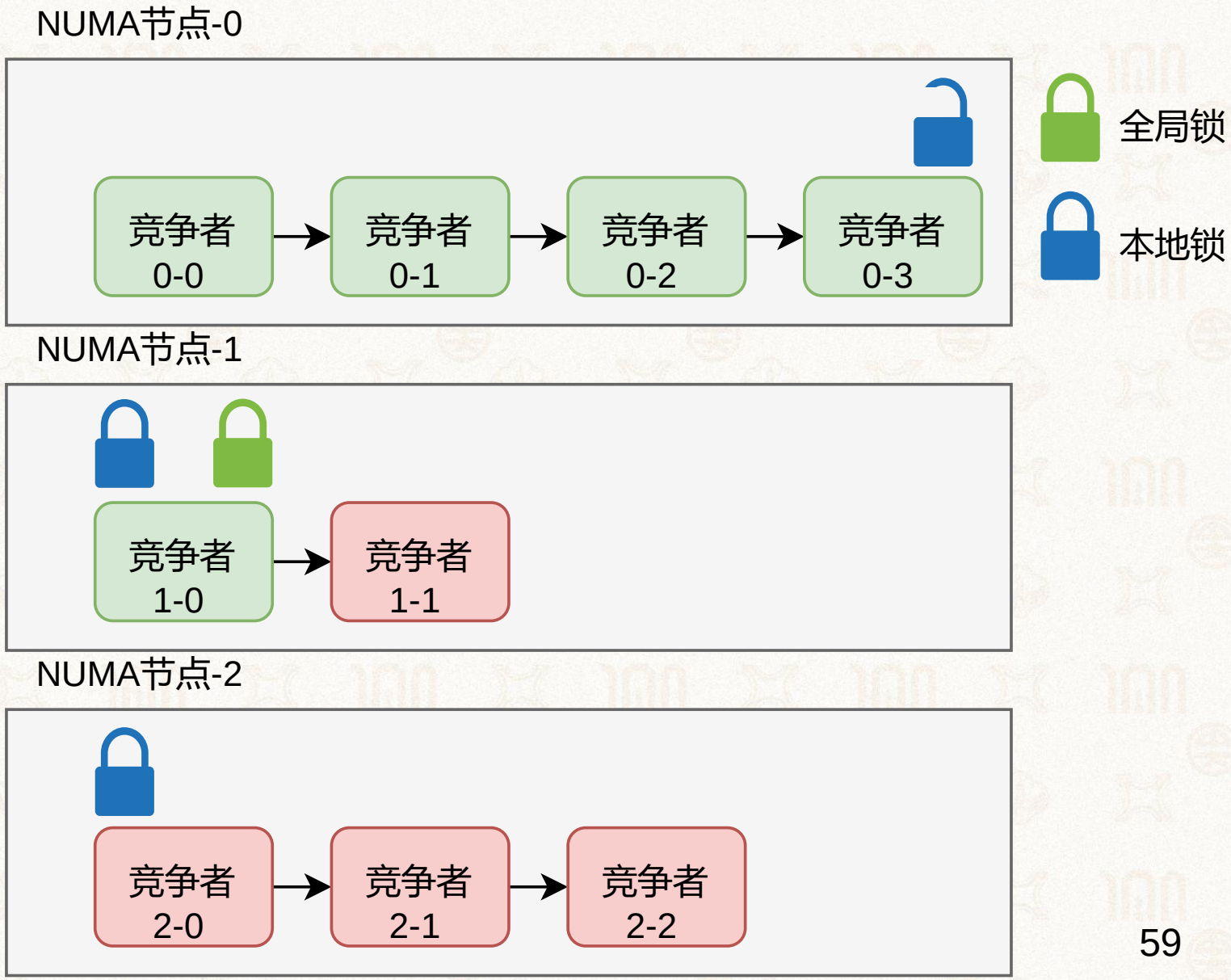
- 成功获取全局锁
- 释放时将其传递给本地等待队列的下一位

- 全局锁在一段时间内只在一个结点内部传递



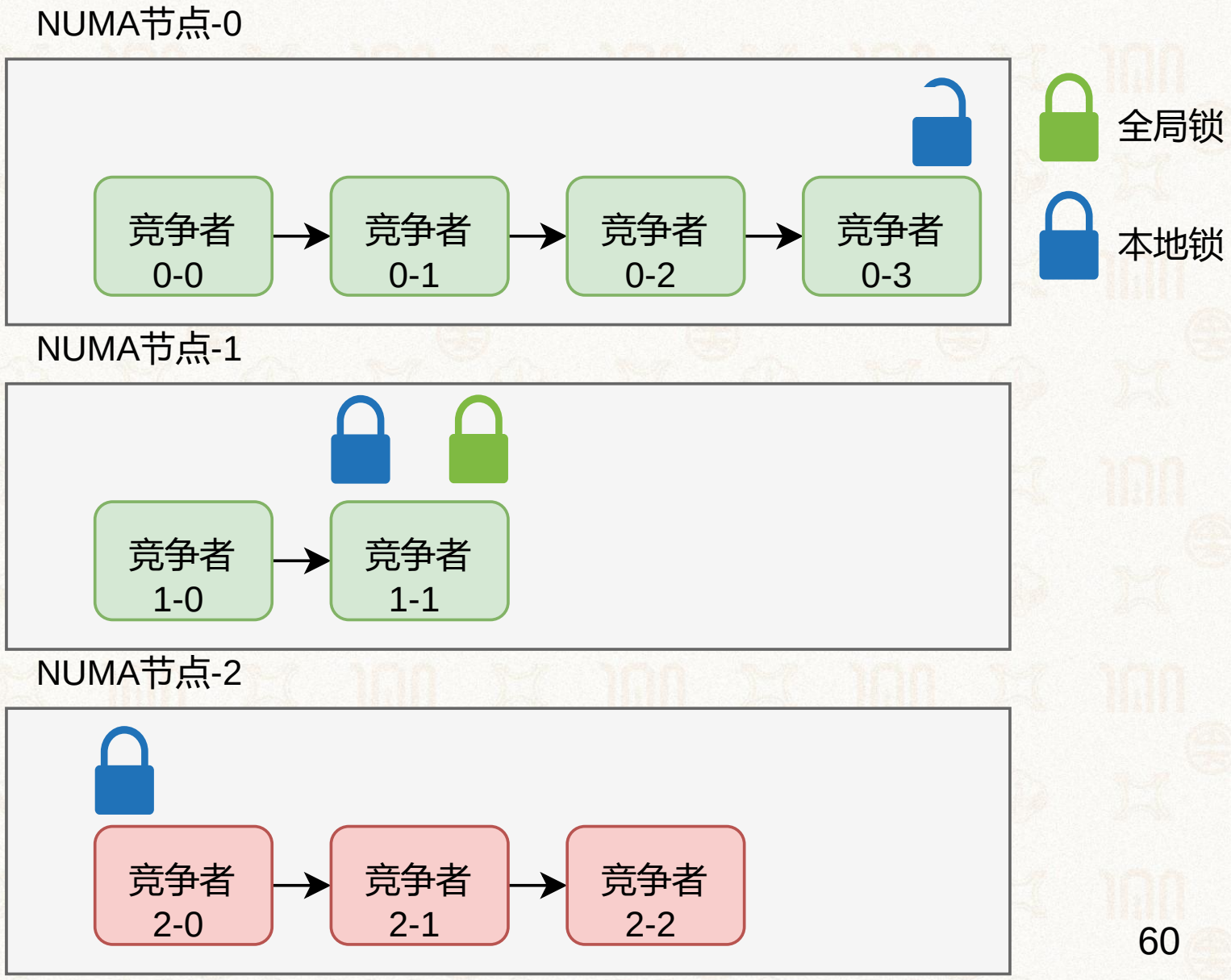
NUMA感知设计：以cohort锁为例

- 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



NUMA感知设计：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



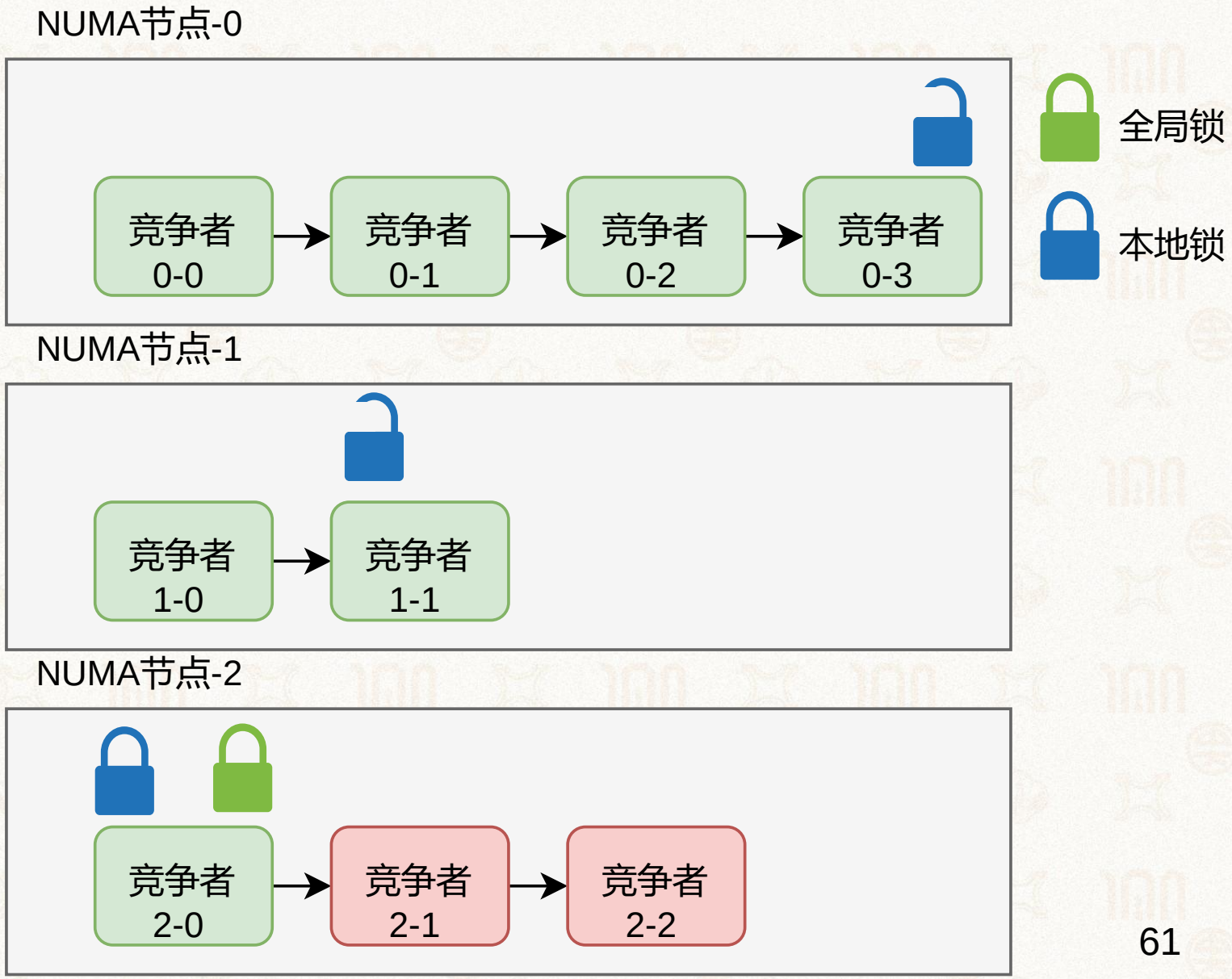
NUMA感知设计：以cohort锁为例

- 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



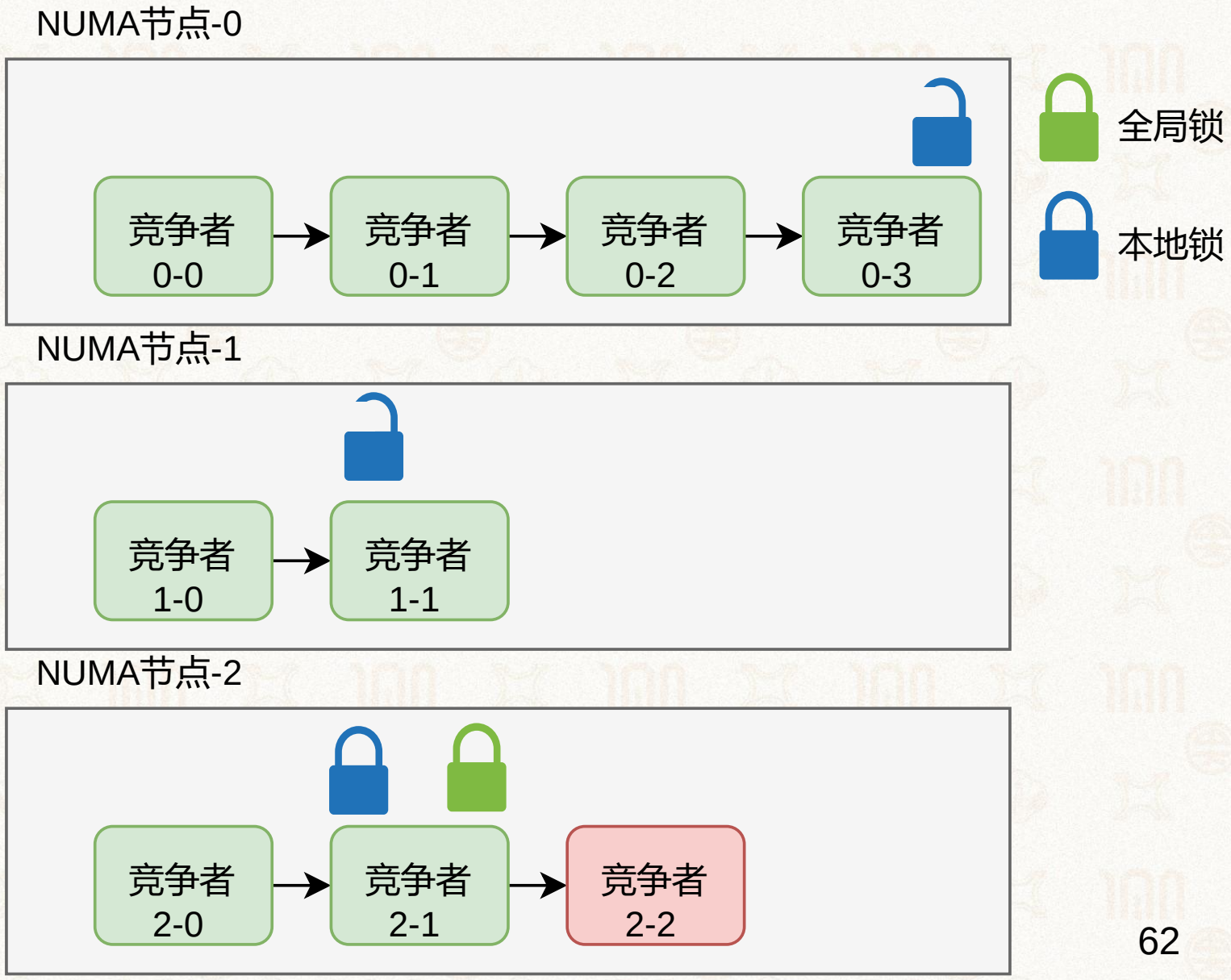
NUMA感知设计：以cohort锁为例

- 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



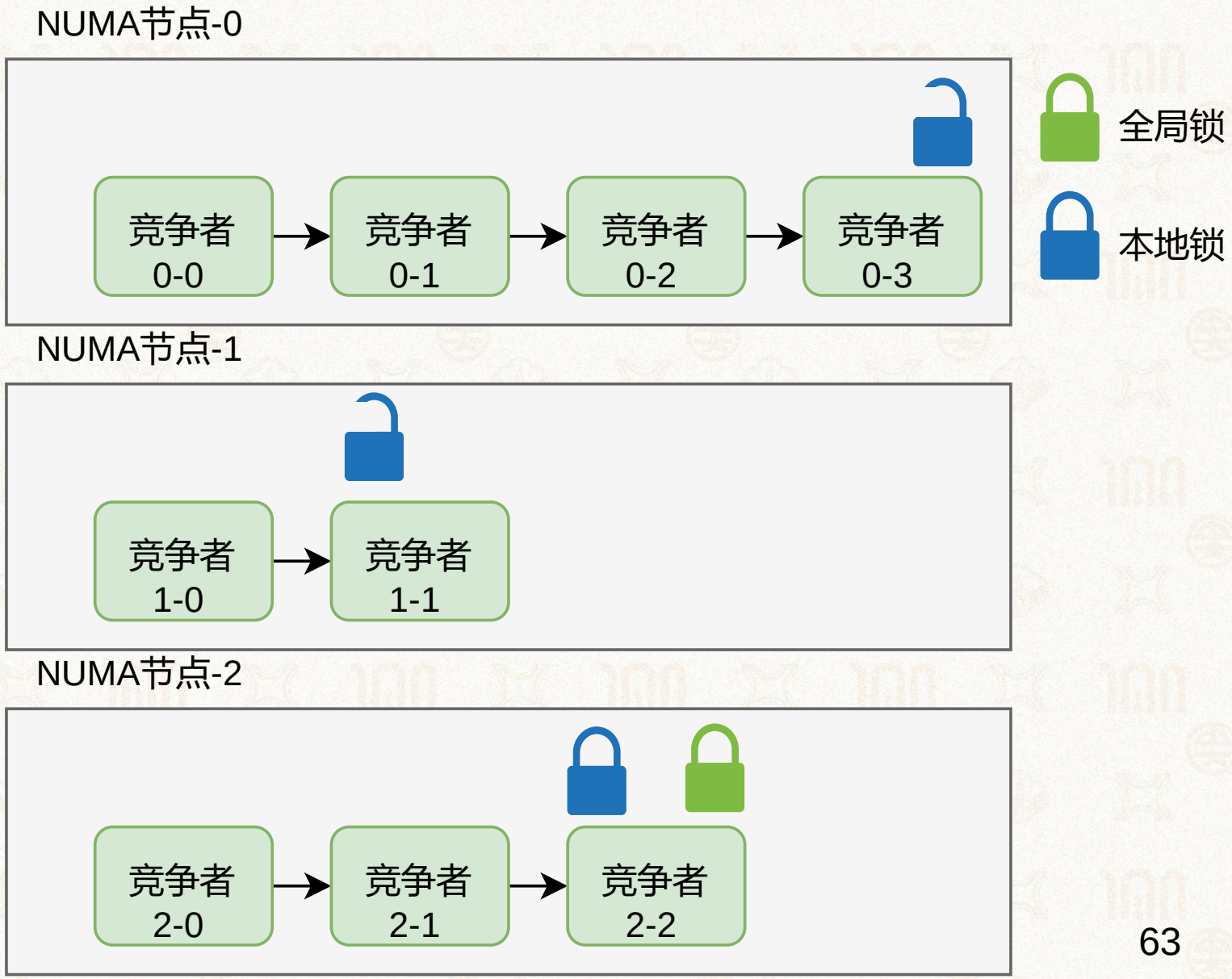
NUMA感知设计：以cohort锁为例

- 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

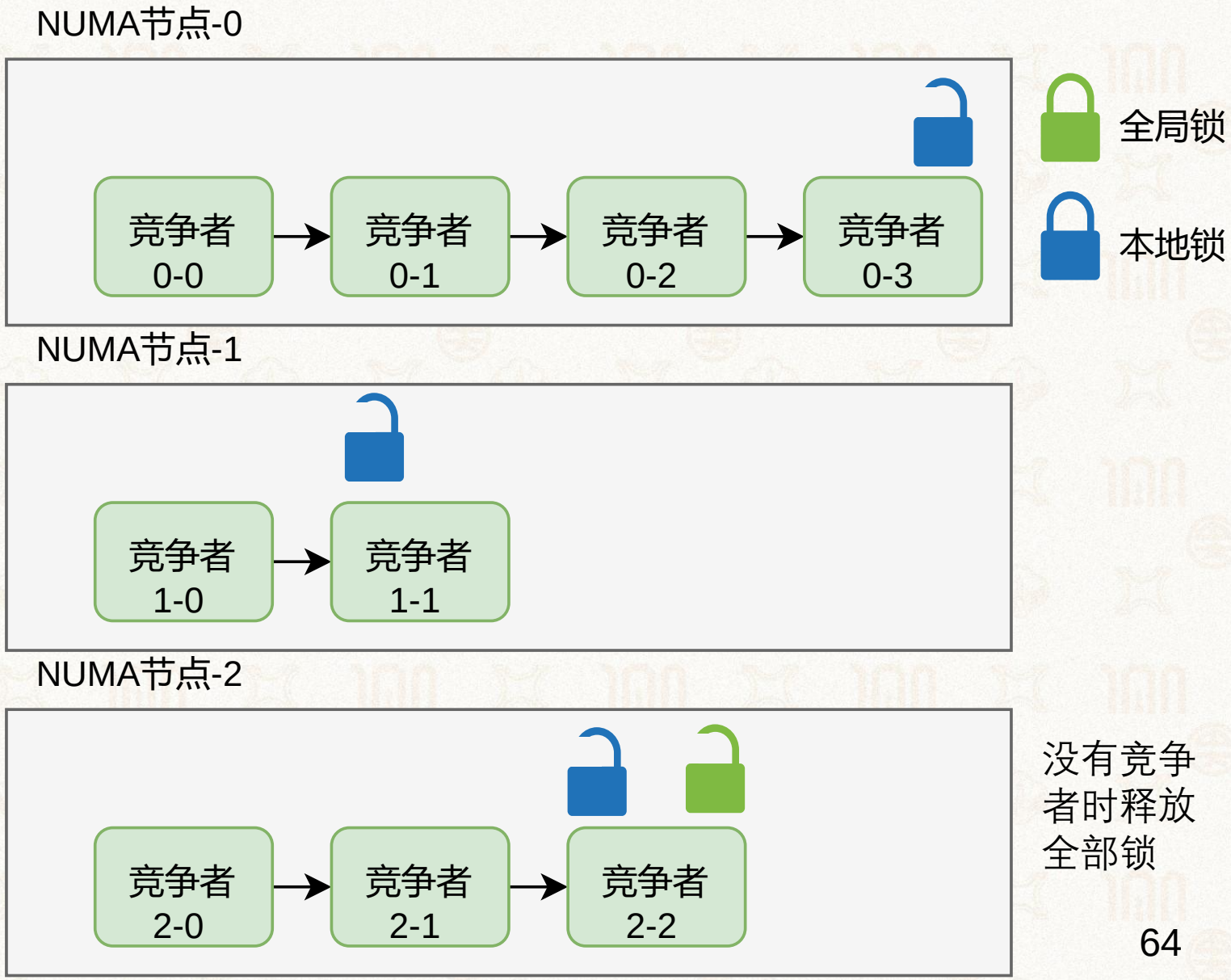
- 成功获取全局锁
- 释放时将其传递给本地等待队列的下一位

- 全局锁在一段时间内只在一个结点内部传递

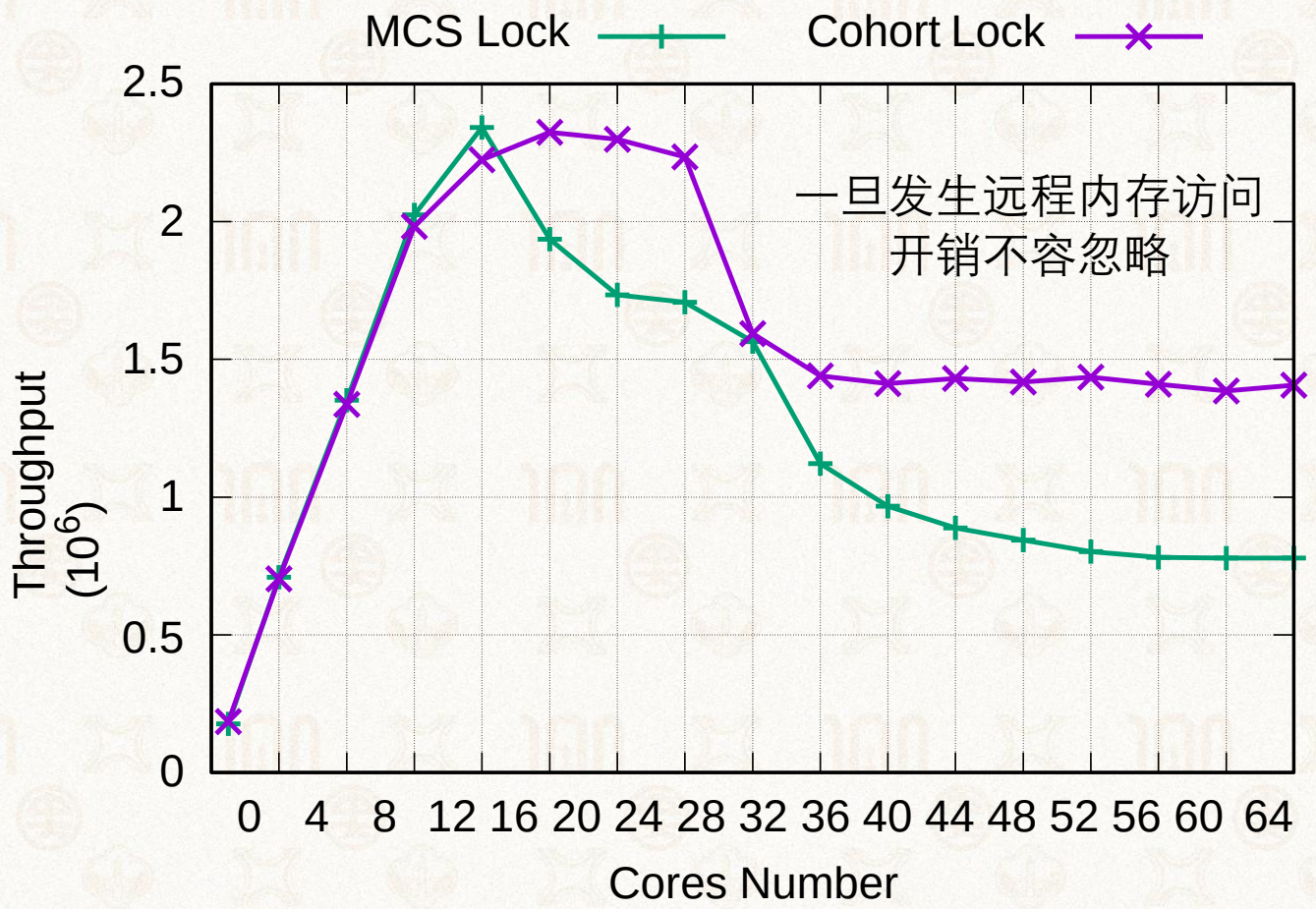


NUMA感知设计：以cohort锁为例

- 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



NUMA感知设计：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感知的应用：尽可能保证其分配的内存的本地性

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)
- 内存一致性模型
  - 不一致现象
  - 四种一致性模型
- 非一致内存访问
  - NUMA系统架构
  - NUMA感知设计

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