

同步原语

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

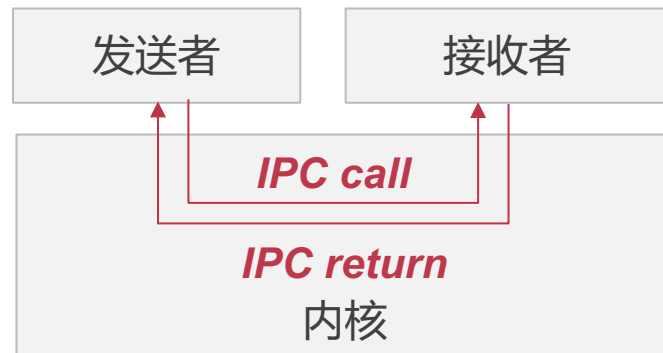
<https://ipads.se.sjtu.edu.cn>

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：进程间通讯

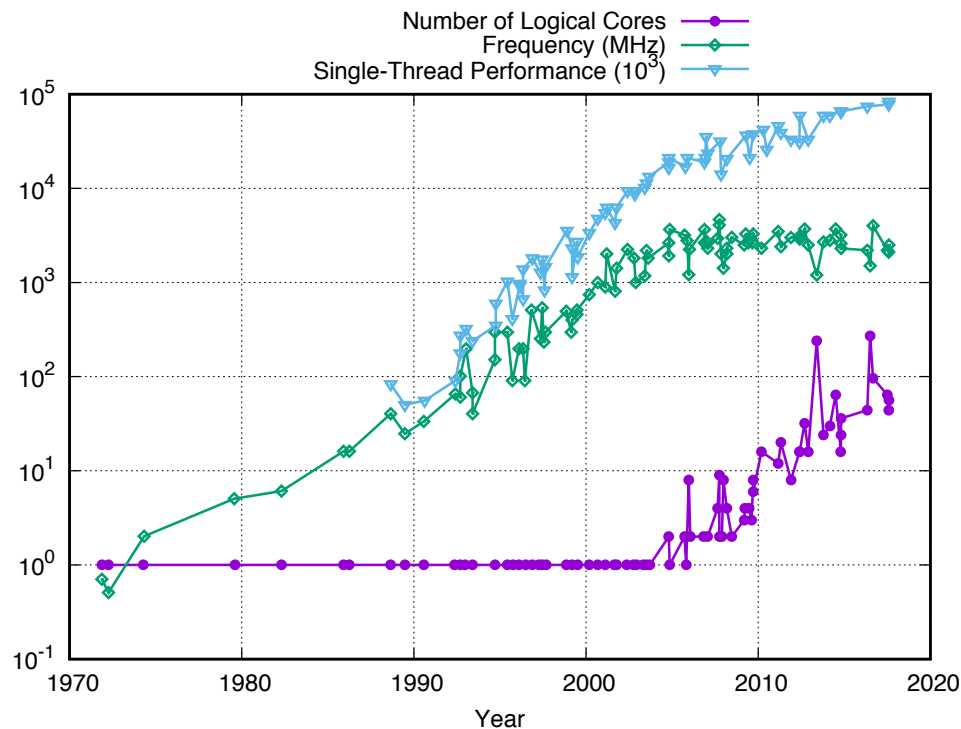
- **进程间通信**：两个(或多个)不同的进程，通过内核或其他共享资源进行通信，来传递控制信息或数据
 - 直接通讯/间接通讯
- **进程间协作**：基于消息传递的抽象



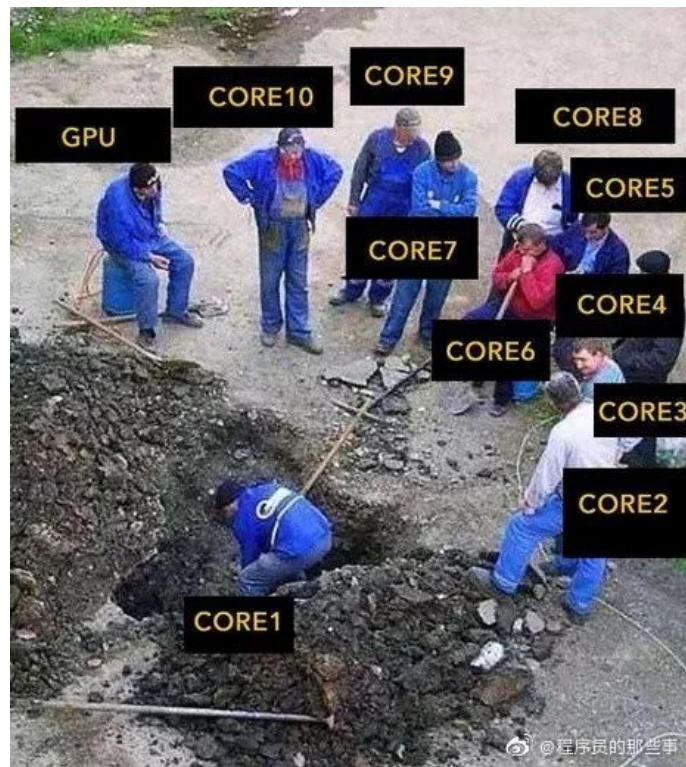
今天的主题：直接基于**共享内存**操作（如发送者直接修改全局变量）

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：

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）

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

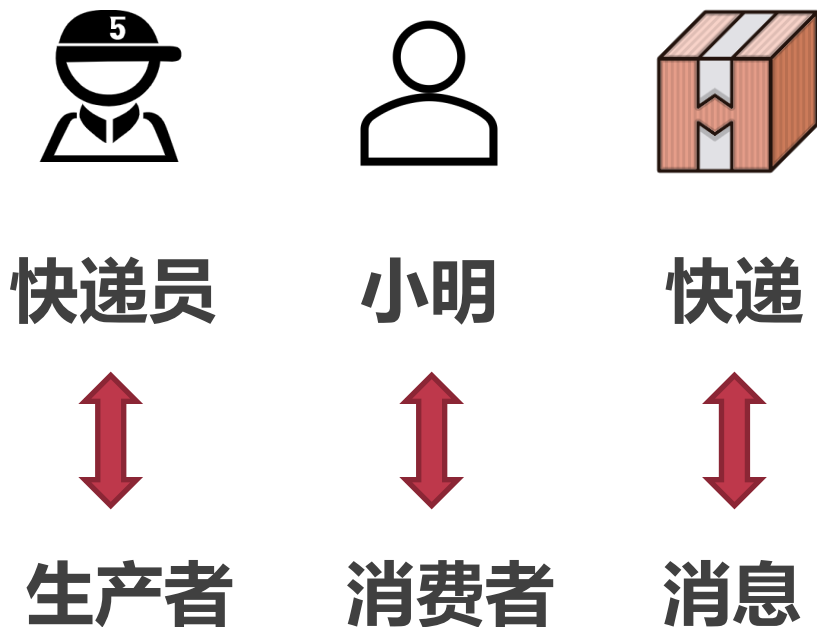
性能保证

- 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

生产者消费者问题，竞争条件，临界区问题

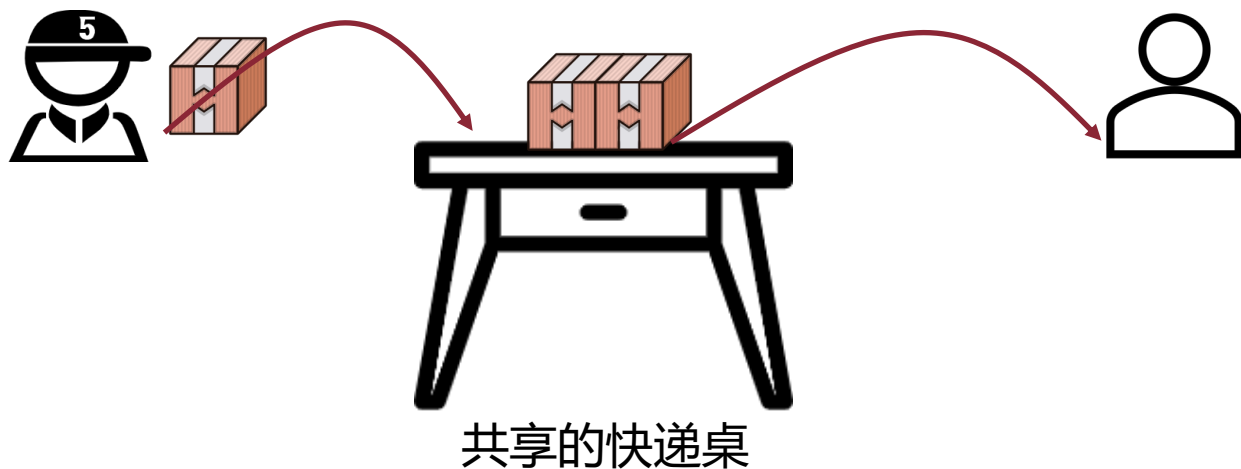


回顾：新冠疫情下快递员送货问题



回顾：新冠疫情下快递员送货问题

- 小区门口的桌子上允许临时放置快递
- 快递员在桌上放置快递，小明从桌上取快递



送货（生产者消费者）问题的基础实现

- 基础实现: 生产者(快递员)

当没有空间时，发送者盲目等待
(快递员等待桌子有空闲空间)

```
while (true) {  
    /* Produce an item */  
    while (prodCnt - consCnt == BUFFER_SIZE)  
        ; /* do nothing -- no free buffers */  
    buffer[prodCnt % BUFFER_SIZE] = item;  
    prodCnt = prodCnt + 1;  
}
```

发送者放置消息
(快递员将快递放在桌上空闲空间)

送货（生产者消费者）问题的基础实现

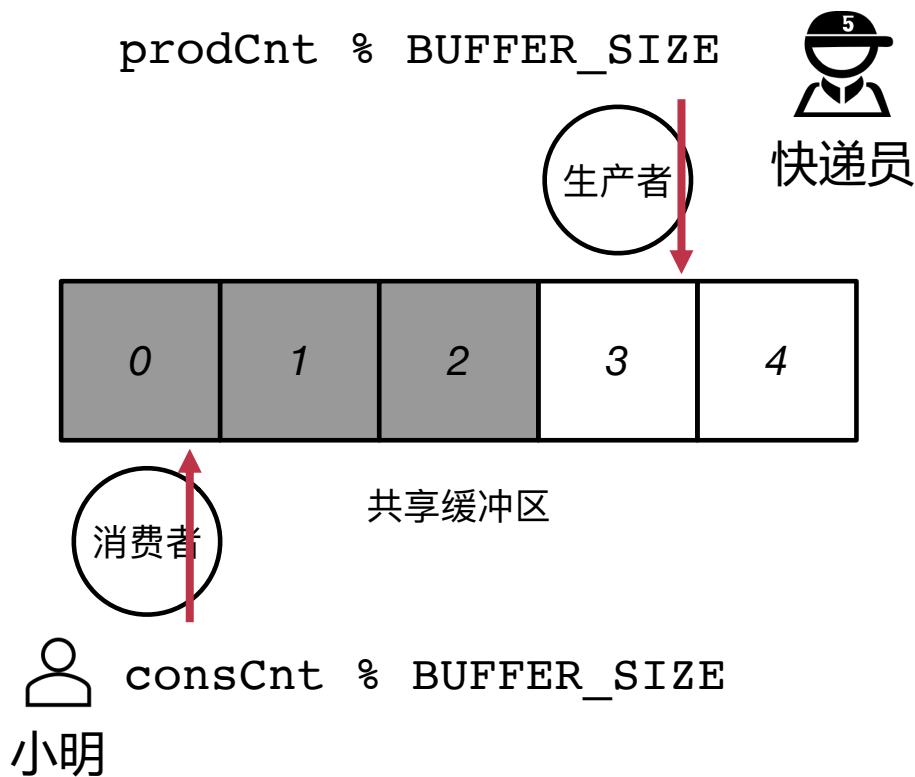
- 基础实现: 消费者(小明)

当没有新消息时，接收者盲目等待
(小明盲目查看桌上状态和等待)

```
while (true) {  
    while (prodCnt == consCnt)  
        ; /* do nothing */  
    item = [consCnt % BUFFER_SIZE] ;  
    consCnt = consCnt + 1;  
}
```

接收者获取消息
(小明拿到最先到达的一个快递)

送货（生产者消费者）问题方案总结



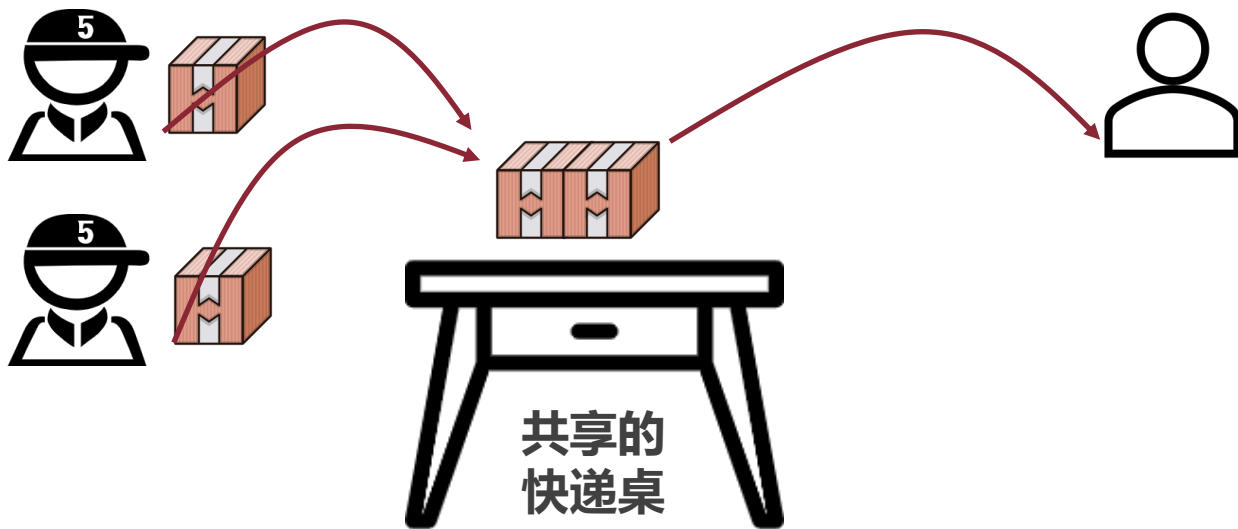
通过两个计数器来协调

生产者（快递员）与消费者（小明）

问：如果有多个生产者和消费者呢？

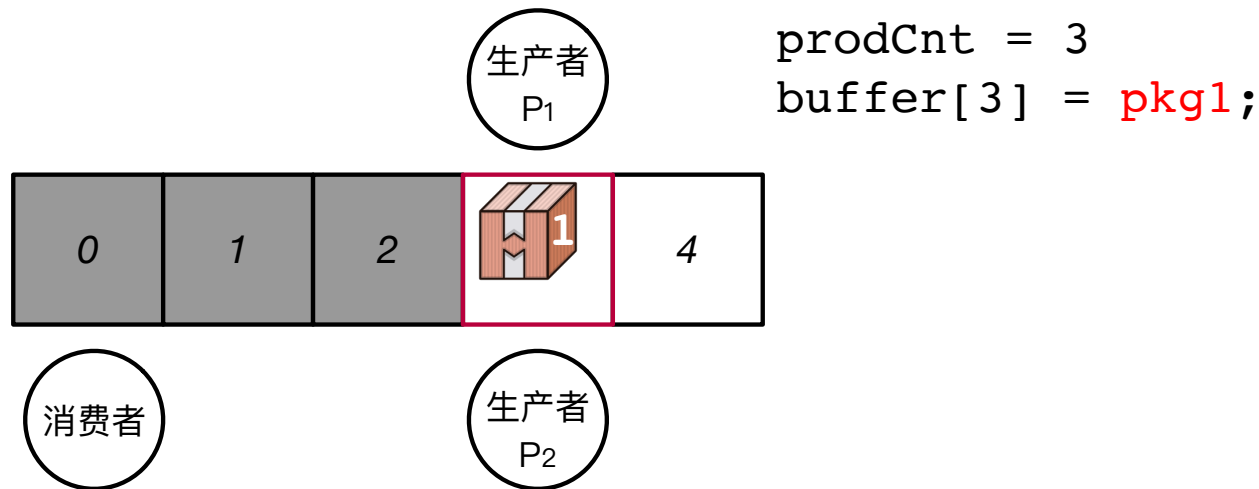
多生产者消费者问题

假设同一时刻有多个生产者：



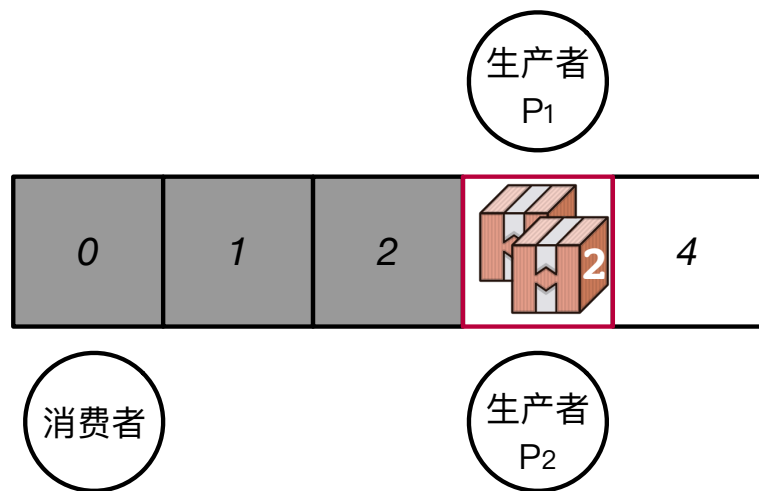
多生产者消费者问题

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ; /* do nothing -- no free buffers */
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```



多生产者消费者问题

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ; /* do nothing -- no free buffers */
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```



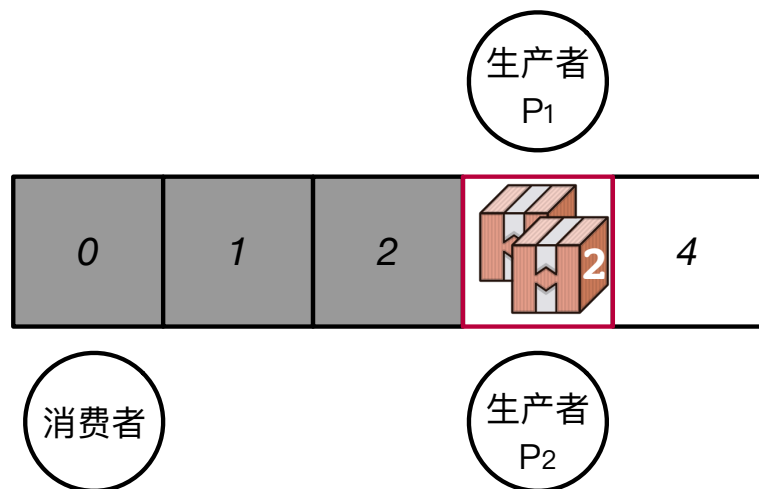
prodCnt = 3
buffer[3] = pkg1;

prodCnt = 3
buffer[3] = pkg2;

将快递碰到地上

多生产者消费者问题

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ; /* do nothing -- no free buffers */
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```



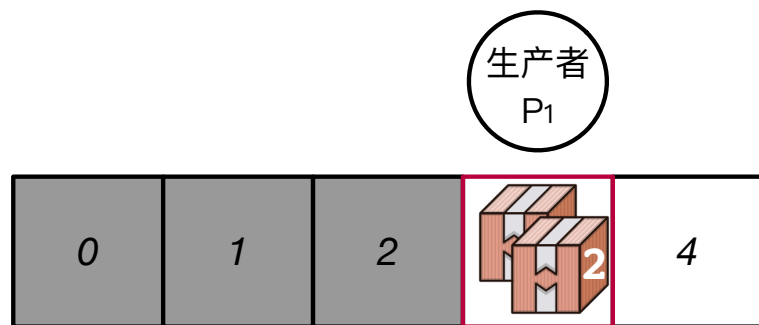
```
prodCnt = 3
buffer[3] = pkg1;
prcdCnt = 4
```

```
prodCnt = 3
buffer[3] = pkg2;
```

将快递碰到地上

多生产者消费者问题

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ; /* do nothing -- no free buffers */
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```



prodCnt = 3
buffer[3] = pkg1;
prcdCnt = 4

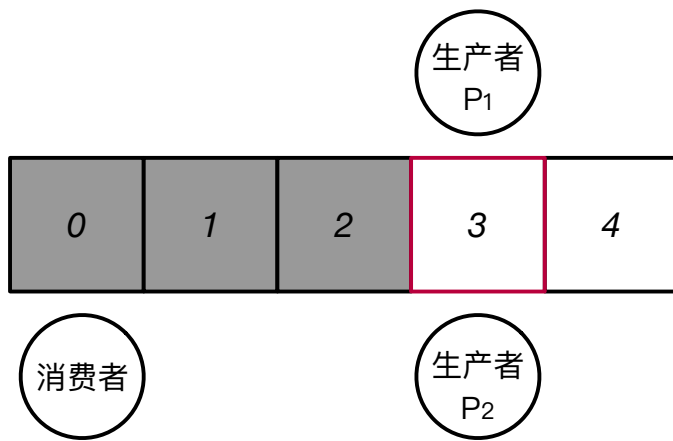
prodCnt = 3
buffer[3] = pkg2;
prodCnt = 5

将快递碰到地上

如何确保他们**不会**将新产生的数据放入到同一个缓冲区中，造成**数据覆盖**？

竞争条件 Race Condition

如何确保他们**不会**将新产生的数据放入到同一个缓冲区中，造成**数据覆盖**？



此时产生了**竞争条件**（竞争冒险、竞态条件）：

- 当多个进程同时对共享的数据进行操作
- 该共享数据最后的结果**依赖于这些进程特定的执行顺序**

临界区问题

如何确保他们**不会**将新产生的数据放入到同一个缓冲区中，造成**数据覆盖**？

```
while(TRUE) {
```

填写登记表，
看能不能进

申请进入临界区

放/取快递

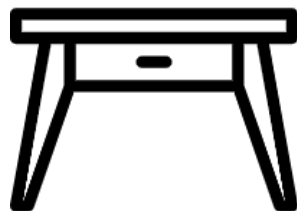
临界区部分

更新登记表

通知退出临界区

其他代码

```
}
```



给快递桌添加
登记本



必须保证：某一时刻有且只有一个进程可以进入临界区执行

解决临界区问题的三个要求

- **互斥访问**：在同一时刻，**有且仅有一个进程**可以进入临界区。（**一个快递员放快递**）
- **有限等待**：当一个进程申请进入临界区之后，必须在**有限的时间**内获得许可进入临界区而不能无限等待。（**快递员还要送其他的**）
- **空闲让进**：当没有进程在临界区中时，必须在申请进入临界区的进程中选择一个进入临界区，保证执行临界区的**进展**。（**没人放快递时要选一个快递员去放**）

```
while(TRUE) {
```

申请进入临界区

临界区部分

通知退出临界区

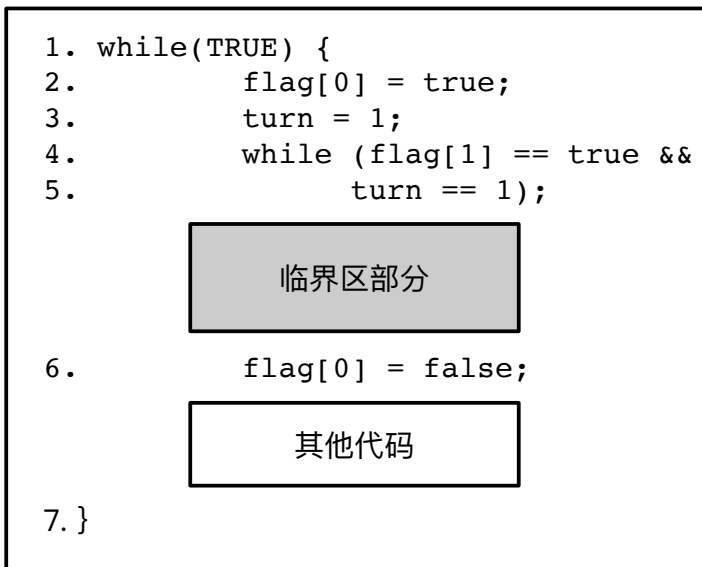
其他代码

```
}
```

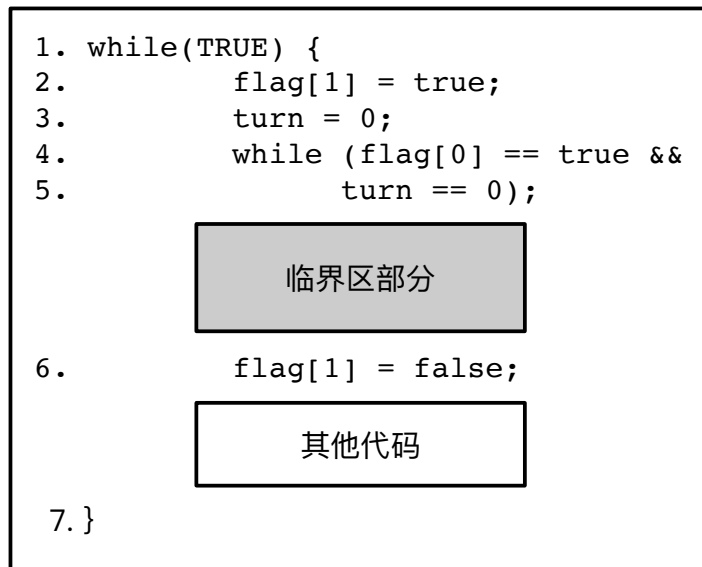
基本原语：软件实现与硬件实现

软件解决方案：皮特森算法

进程 - 0



进程 - 1



思考：是否满足解决临界区问题的三个必要条件？

互斥访问

有限等待

空闲让进

有没有更简单的方法：关闭中断

这样能解决临界区问题吗？

```
while(TRUE) {
```

申请进入临界区

关闭所有核心的中断

临界区部分

通知退出临界区

开启所有核心的中断

其他代码

```
}
```

有没有更简单的方法：关闭中断

```
while(TRUE) {
```

申请进入临界区

临界区部分

通知退出临界区

其他代码

```
}
```

这样能解决临界区问题吗？

关闭所有核心的中断

可以解决单个CPU

核上的临界区问题

如果在多个核心中，

关闭中断不能阻塞

开启所有核心的中断

其他进程执行

并不能阻止多个CPU核同时进入临界区

有没有更简单的方法？

如何确保他们**不会**将新产生的数据放入到同一个缓冲区中，造成**数据覆盖**？

```
while(TRUE) {
```

拿钥匙

申请进入临界区

放/取快递

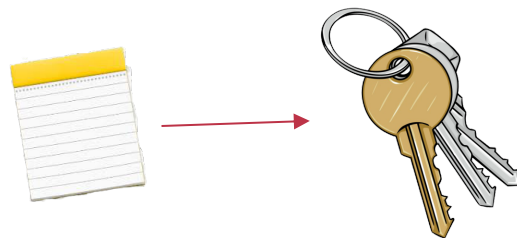
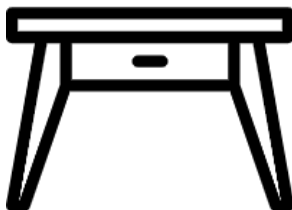
临界区部分

还钥匙

通知退出临界区

其他代码

```
}
```



给快递桌添加**互斥锁**



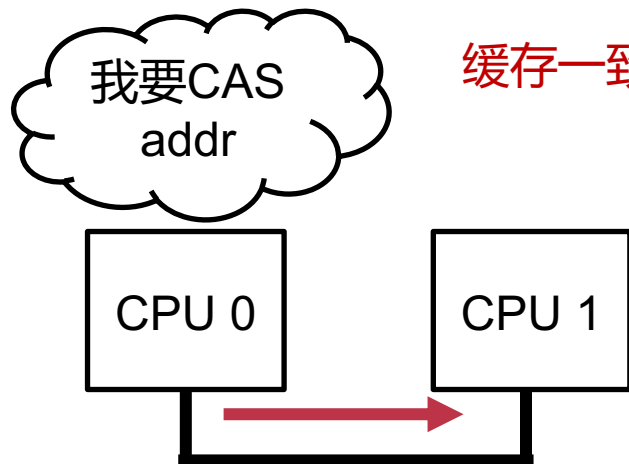
硬件原子操作

原子操作：

- 不可被打断的操作集合
- 如同执行一条指令
- 其他核心不会看到中间状态 all-or-nothing

```
int CAS(int *addr, int expected, int new_value) {  
    int tmp = *addr;  
    if (*addr == expected)  
        *addr = new_value;  
    return tmp;  
}
```

硬件原子操作：Intel锁总线实现



缓存一致性后面会介绍

互联总线

声明一下，并且锁总线

```
1. int CAS(int *addr, int expected,  
CPU 0      int new_value) {  
2.     int tmp = *addr;  
3.     if (*addr == expected)  
4.         *addr = new_value;  
5.     return tmp;  
6. }
```

对任意地址的修改都要经过**总线**的

通过**锁总线**来实现原子操作*

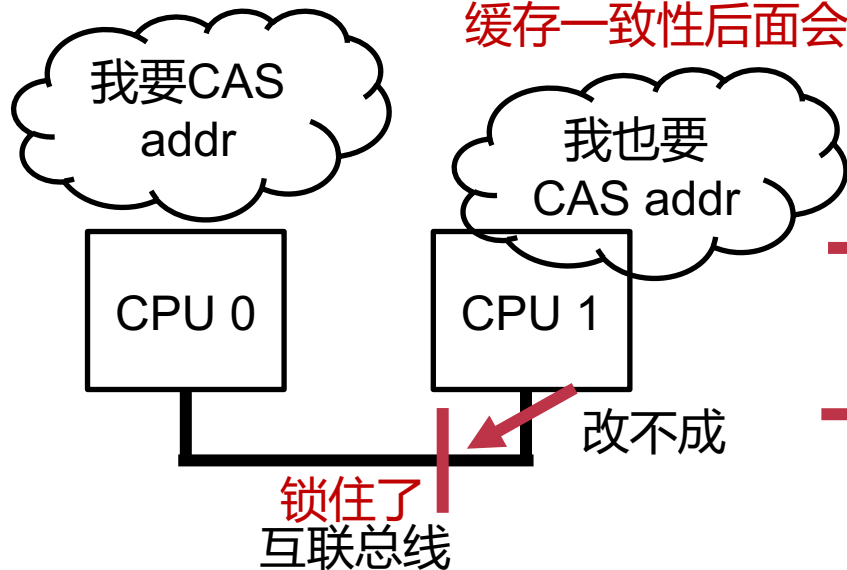
在第二行锁总线，在第五行放总线

*Intel手册描述，实际实现可能与此不同（更高效）

上海交通大学并行与分布式系统研究所（IPADS@SJTU）

硬件原子操作：Intel锁总线实现

缓存一致性后面会介绍



```
1. int CAS(int *addr, int expected,  
           int new_value) {  
2.     int tmp = *addr;  
3.     if (*addr == expected)  
4.         *addr = new_value;  
5.     return tmp;  
6. }
```

对任意地址的修改都要经过**总线**的

通过**锁总线**来实现原子操作*

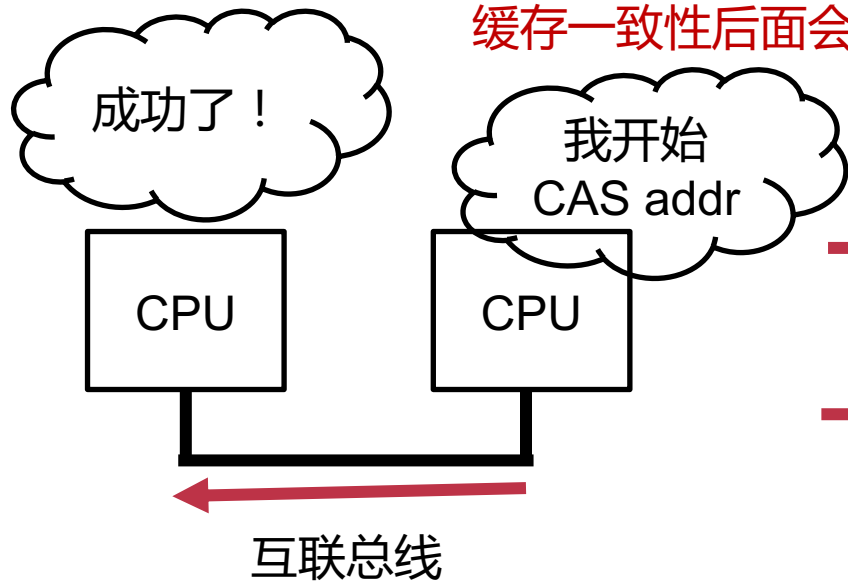
在第二行锁总线，在第五行放总线

*Intel手册描述，实际实现可能与此不同（更高效）

上海交通大学并行与分布式系统研究所（IPADS@SJTU）

硬件原子操作：Intel锁总线实现

缓存一致性后面会介绍



声明一下，并且锁总线

```
1. int CAS(int *addr, int expected,  
           int new_value) {  
2.     int tmp = *addr;  
3.     if (*addr == expected)  
4.         *addr = new_value;  
5.     return tmp;  
6. }
```

CPU 1

CPU 0

对任意地址的修改都要经过**总线**的

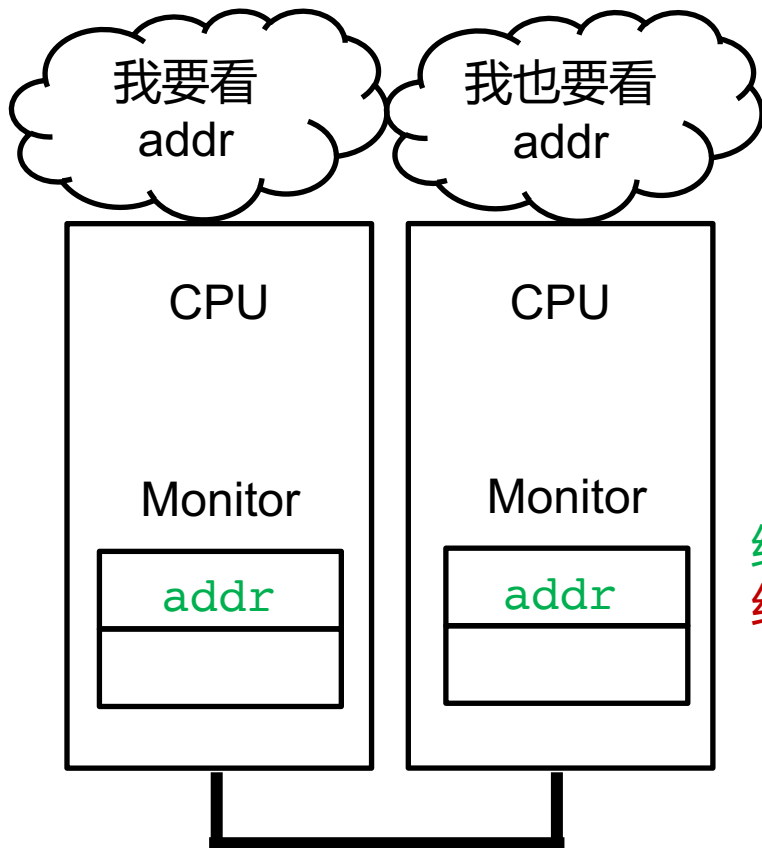
通过**锁总线**来实现原子操作*

在第二行锁总线，在第五行放总线

*Intel手册描述，实际实现可能与此不同（更高效）

上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

硬件原子操作：ARM使用LL/SC实现



```
1. int CAS(int *addr, int expected,
           int new_value) {
2. CPU 0/1 int tmp = *addr; LL
3.     if (*addr == expected)
4.         *addr = new_value; SC
5.     return tmp;
6. }
```

绿色没人修改

红色被修改

Load-linked & Store-conditional

第二行读的时候**监视**addr

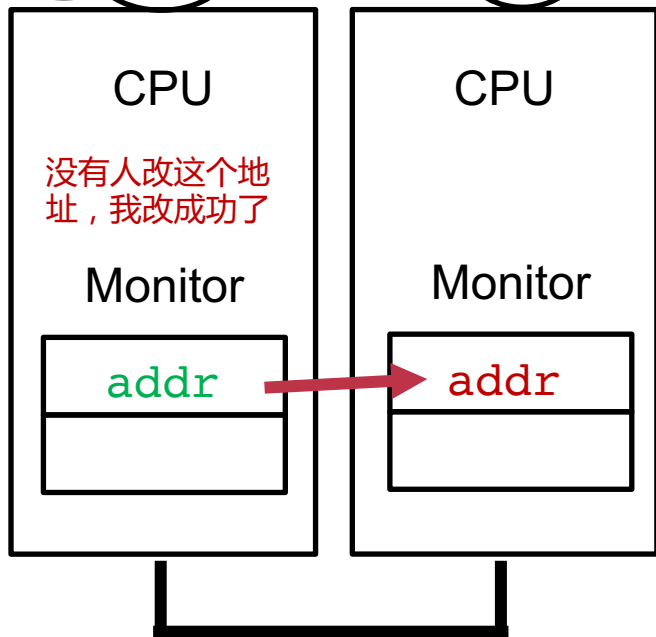
第四行修改的时候看addr**是否被其他人修改**

没人修改就写成功，否则回到第二行

硬件原子操作：ARM使用LL/SC实现

对比成功了，
我要改

我也要
看
addr



```
1. int CAS(int *addr, int expected,  
           int new_value) {  
2.     int tmp = *addr; LL  
3.     if (*addr == expected)  
4. CPU 0/1         *addr = new_value; SC  
5.     return tmp;  
6. }
```

绿色没人修改
红色被修改

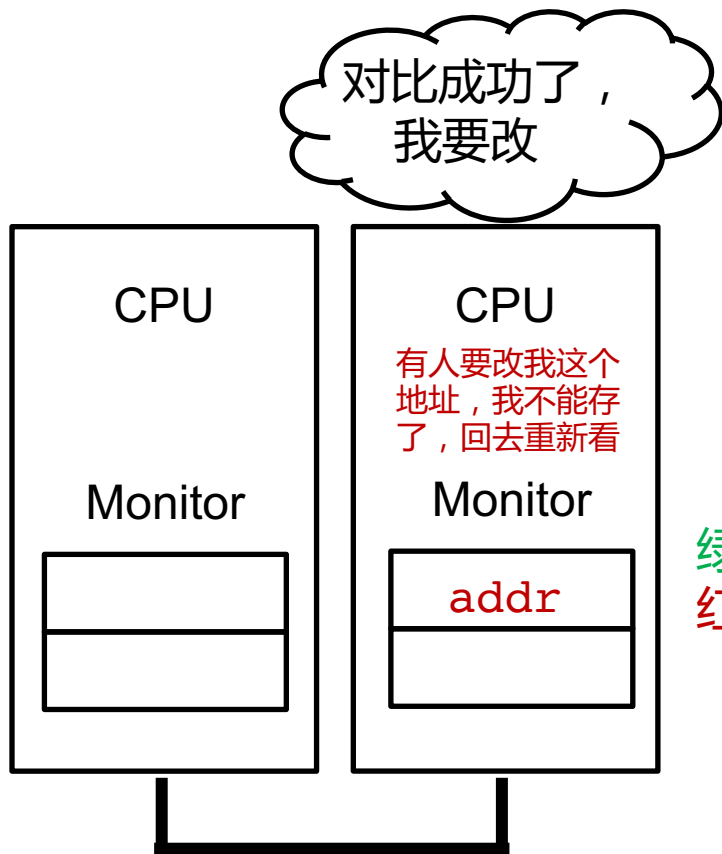
Load-linked & Store-conditional

第二行读的时候**监视**addr

第四行修改的时候看addr**是否被其他人修改**

没人修改就写成功，否则回到第二行

硬件原子操作：ARM使用LL/SC实现



```
1. int CAS(int *addr, int expected,  
    失败重试 int new_value) {  
2.     int tmp = *addr; LL  
3.     if (*addr == expected)  
4. CPU 1     *addr = new_value; SC  
5. CPU 0     return tmp;  
6. }
```

绿色没人修改
红色被修改

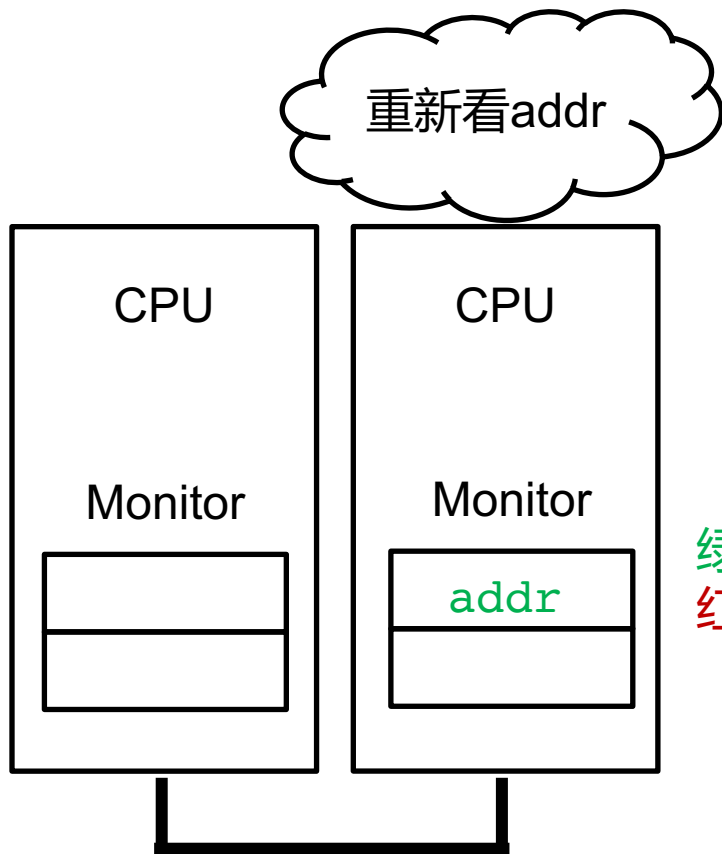
Load-linked & Store-conditional

第二行读的时候**监视**addr

第四行修改的时候看addr**是否被其他人修改**

没人修改就写成功, 否则回到第二行

硬件原子操作：ARM使用LL/SC实现



```
1. int CAS(int *addr, int expected,  
           int new_value) {  
2. CPU 1 int tmp = *addr; LL  
3.     if (*addr == expected)  
4.         *addr = new_value; SC  
5.     return tmp;  
6. }
```

绿色没人修改
红色被修改

Load-linked & Store-conditional

第二行读的时候**监视**addr

第四行修改的时候看addr**是否被其他人修改**

没人修改就写成功，否则回到第二行

硬件原子操作：使用硬件辅助

比较并替换 (Compare And Swap)

```
1. int CAS(int *addr, int expected, int new_value) {  
2.     int tmp = *addr;  
3.     if (*addr == expected)  
4.         *addr = new_value;  
5.     return tmp;  
6. }
```

获取并增加 (Fetch And Add)

```
1. int FAA(int *addr, int add_value) {  
2.     int tmp = *addr;  
3.     *addr = *addr + add_value;  
4.     return tmp;  
5. }
```

锁的实现

自旋锁 (Spinlock)

```
while(TRUE) {
```

申请进入临界区

临界区部分

通知退出临界区

其他代码

```
}
```

全局标记 *lock : 0表示空闲, 1表示锁

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

lock操作

```
*lock = 0;
```

unlock操作

自旋锁 (Spinlock)

思考：是否满足解决临界区问题的三个必要条件？

- 互斥访问 ✓
- 有限等待？
 - 有的“运气差”的进程可能永远也不能成功CAS => 出现饥饿
- 空闲让进？
 - 依赖于硬件 => 当多个核同时对一个地址执行原子操作时，能否保证至少有一个能够成功*

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

自旋锁实现

*这里我们认为硬件能够确保原子操作make progress

排号锁 (Ticket Lock)

思考：我们如何保证竞争者的公平性？

排号锁 (Ticket Lock)

思考：我们如何保证竞争者的公平性？

通过遵循竞争者到达的顺序来传递锁。

owner：表示当前在吃的食客

next：表示目前放号的最新值



假设只有一桌...



owner = 3
next = 6



2. 等待叫号
`while(owner != my_ticket);`

1. 拿号 => 6号
`my_ticket = atomic_FAA(&next, 1)`

排号锁 (Ticket Lock)

思考：我们如何保证竞争者的公平性？

通过遵循竞争者到达的**顺序**来传递锁。

owner：表示当前在吃的食客

next：表示目前放号的最新值



```
while(owner != my_ticket);
```

 海底捞

假设只有一桌...



1. 吃完了，买单

2. 叫下个人进来

`owner += 1`



`owner = 3`

`next = 7`

排号锁 (Ticket Lock)

思考：我们如何保证竞争者的公平性？

通过遵循竞争者到达的顺序来传递锁。

owner：表示当前的持有者 next：表示目前放号的最新值

lock操作

```
1. my_ticket = atomic_FAA(  
    &lock->next, 1);  
2. while(lock->owner !=  
    my_ticket)  
    /* waiting */;
```

拿号

等号

unlock操作

```
1. lock->owner ++;
```

叫号

排号锁 (Ticket Lock)

思考：是否满足解决临界区问题的三个必要条件？

- 互斥访问 ✓
- 有限等待？
 - 按照顺序，在前序竞争者保证有限时间释放时，可以达到有限等待
- 空闲让进* ✓

```
void lock(int *lock) {  
    volatile unsigned my_ticket =  
        atomic_FAA(&lock->next, 1);  
    while(lock->owner != my_ticket)  
        /* busy waiting */;  
}  
  
void unlock(int *lock) {  
    lock->owner ++;  
}
```

排号锁实现

*这里我们认为硬件能够确保原子操作make progress

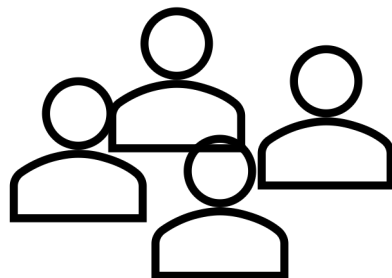


读写锁

公告栏问题



写者



读者



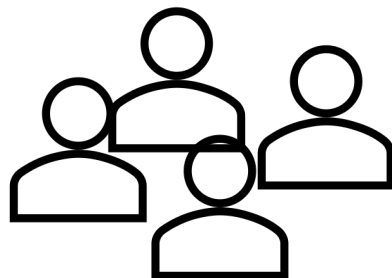
思考：多个读者如果希望读公告栏，他们互斥吗？

思考：如何避免读者看到一半就被写者撤走了，我们怎么办？

公告栏问题



写者



读者



思考：多个读者如果希望读公告栏，他们互斥吗？

不互斥

思考：如何避免读者看到一半就被写者撤走了，我们怎么办？

使用互斥锁
且读者也要用互斥锁

读写锁的使用示例

```
struct rwlock *lock;
char data[SIZE];

void reader(void) {
    lock_reader(lock);
    read_data(data);
    unlock_reader(lock);
}

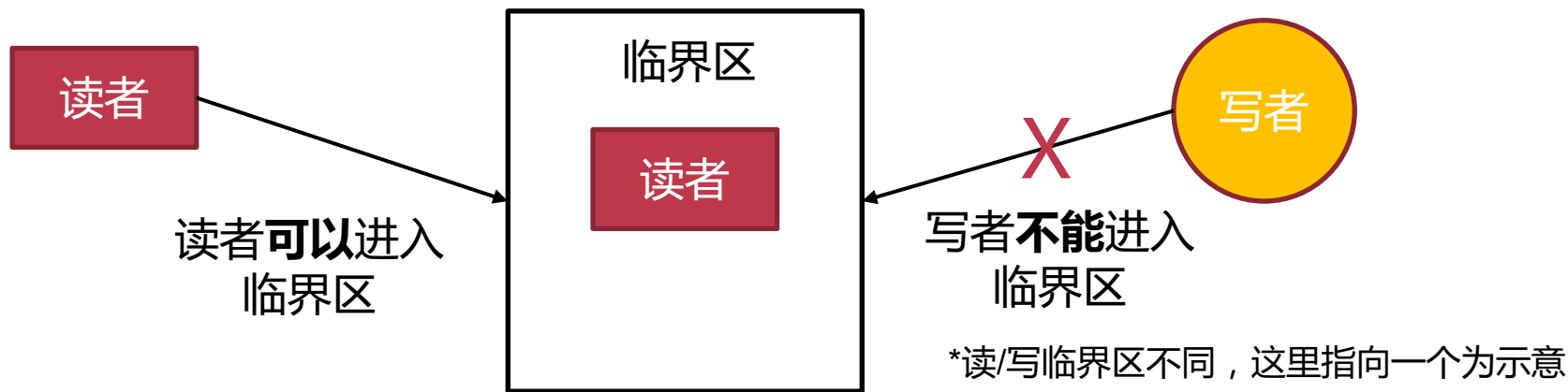
void writer(void) {
    lock_writer(lock);
    update_data(data);
    unlock_writer(lock);
}
```

读写锁

互斥锁：所有的进程均互斥，同一时刻**只能有一个进程**进入临界区

对于部分只读取共享数据的进程过于严厉

读写锁：区分读者与写者，允许读者之间并行，读者与写者之间互斥

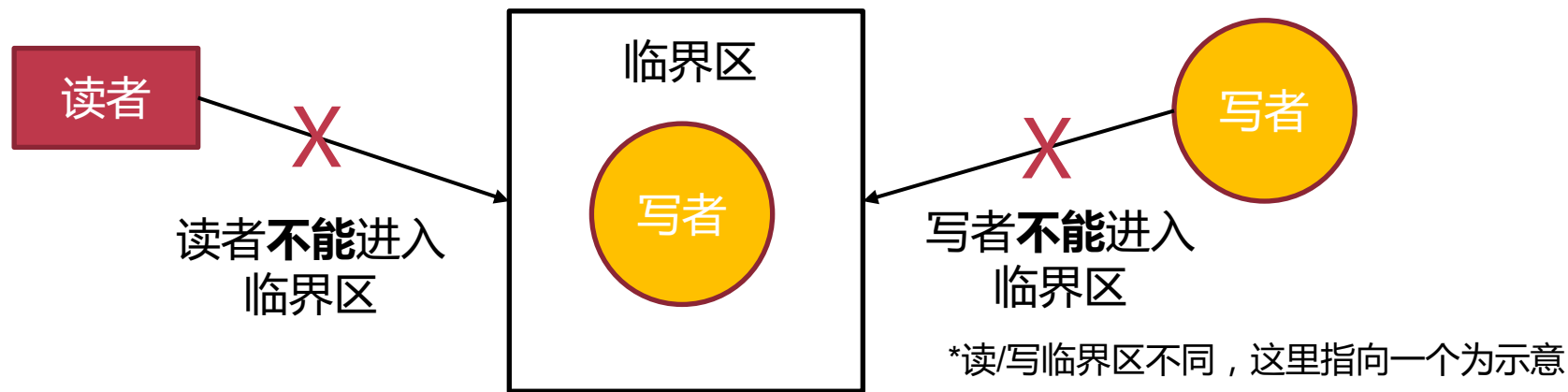


读写锁

互斥锁：所有的进程均互斥，同一时刻只能有一个进程进入临界区

对于部分只读取共享数据的进程过于严厉

读写锁：区分读者与写者，允许读者之间并行，读者与写者之间互斥



读写锁的偏向性

- **考虑这种情况：**

- t_0 ：有读者在临界区
- t_1 ：有新的写者在等待
- t_2 ：另一个读者能否进入临界区？

- **不能：偏向写者的读写锁**

- 后序读者必须等待写者进入后才进入 **更加公平**

- **能：偏向读者的读写锁**

- 后序读者可以直接进入临界区 **更好的并行性**

偏向读者的读写锁实现示例

```
struct rwlock {
    int reader;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader += 1;
    if (lock->reader == 1) /* No reader there */
        lock(&lock->writer_lock);
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader -= 1;
    if (lock->reader == 0) /* Is the last reader */
        unlock(&lock->writer_lock);
    unlock(&lock->reader_lock);
}

void lock_writer(struct rwlock *lock) {
    lock(&lock->writer_lock);
}

void unlock_writer(struct rwlock *lock) {
    unlock(&lock->writer_lock);
}
```

读写锁的实现：偏向读者为例



读者锁

读者




临界区



写者锁



读者计数器

1. 获取读者锁，更新读计数器 
2. 如果没有读者在，拿写锁避免写者进入 
3. 释放读者锁 

在读临界区中的读者数量

*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



读者锁



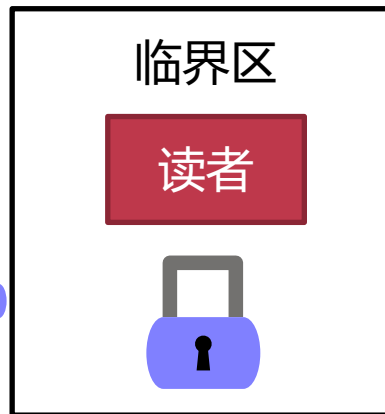
写者锁



读者计数器



1. 尝试拿写锁，等待



在读临界区中的读者数量

*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



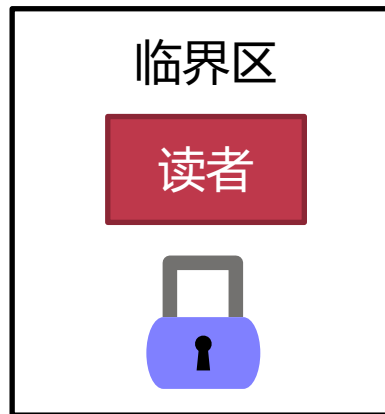
读者锁





写者锁



读者计数器



在读临界区中的读者数量

1. 获取读者锁，更新读计数器 
2. 有读者在，无需再次获取写锁
3. 释放读者锁 

*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



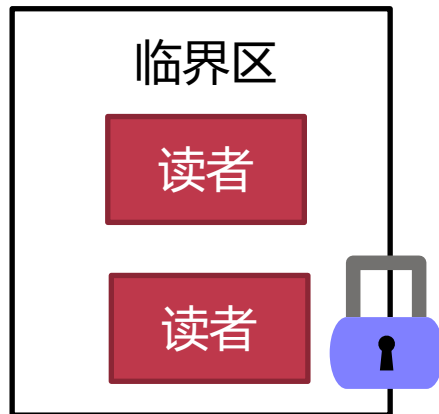
读者锁



写者锁



读者计数器



在读临界区中的读者数量

*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



读者锁

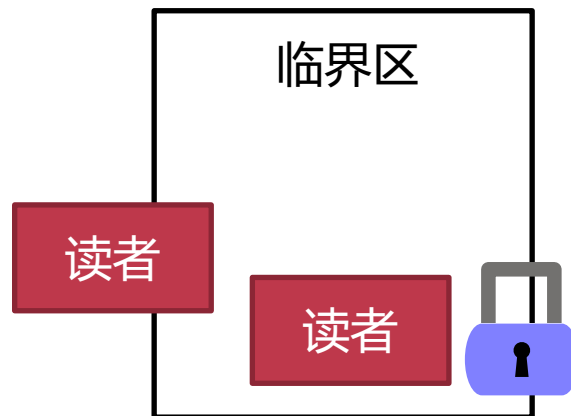




写者锁



读者计数器

在读临界区中的读者数量



1. 获取读者锁，减少计数器 
2. 还有其他读者在，无需释放写锁
3. 释放读者锁 

*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例

此时写者可以进入



*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



读者锁





写者锁

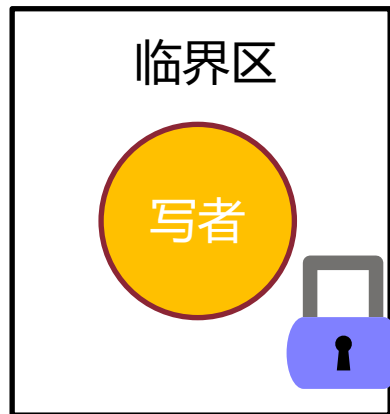


读者计数器

在读临界区中的读者数量

读者

1. 获取读者锁，更新读者计数器 
2. 如果没有读者在，尝试拿写锁避免写者进入，等待。 



*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



读者锁




写者锁

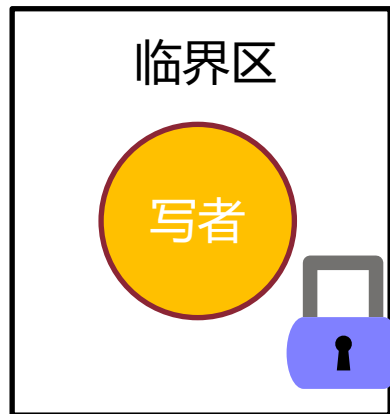


读者计数器

读者

读者

1. 尝试拿读者锁，上面的读者还没释放，等待 



在读临界区中的读者数量

思考：既然读者也要一个读者锁，

那怎么提高读者的效率？

注意：读者锁还有阻塞其他读者的语义，
因此不能用原子操作来替代

*读/写临界区不同，这里指向一个为示意



Read Copy Update, RCU

读写锁读者进入读临界区之前，还是需要**繁杂的操作**

思考：如果我们想去除这些操作，让读者即使在有写者写的时候随意读，我们需要做什么？

Read Copy Update, RCU

读写锁读者进入读临界区之前，还是需要**繁杂的操作**

思考：如何让读者即使在有写者写的时候也能随意读？

需求1：需要一种能够**类似之前硬件原子操作**的方式，让读者要么看到旧的值，要么看到新的值，不会读到任何中间结果。

硬件原子操作：

1. 硬件原子操作有大小限制（最大128 bit）
2. 性能瓶颈

Read Copy Update, RCU

读写锁读者进入读临界区之前，还是需要**繁杂的操作**

思考：如果我们想去除这些操作，让读者即使在有写者写的时候随意读，我们需要做什么？

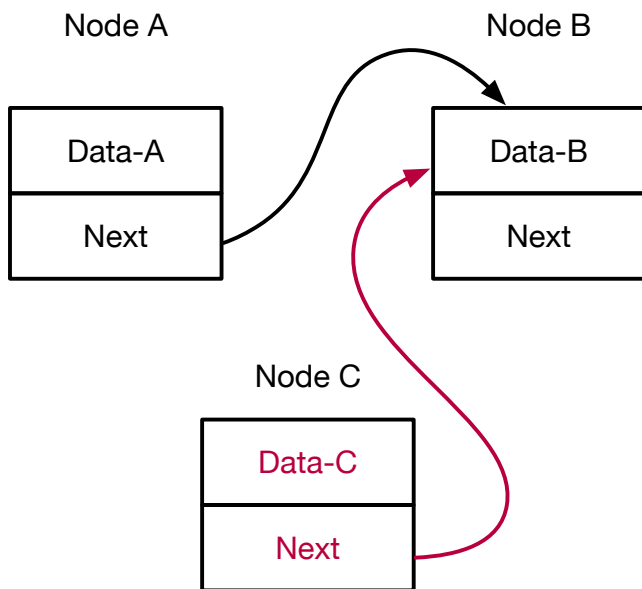
需求1：需要一种能够**类似之前硬件原子操作**的方式，让读者要么看到旧的值，要么看到新的值，不会读到任何中间结果。

单拷贝原子性 (Single-copy atomicity) :

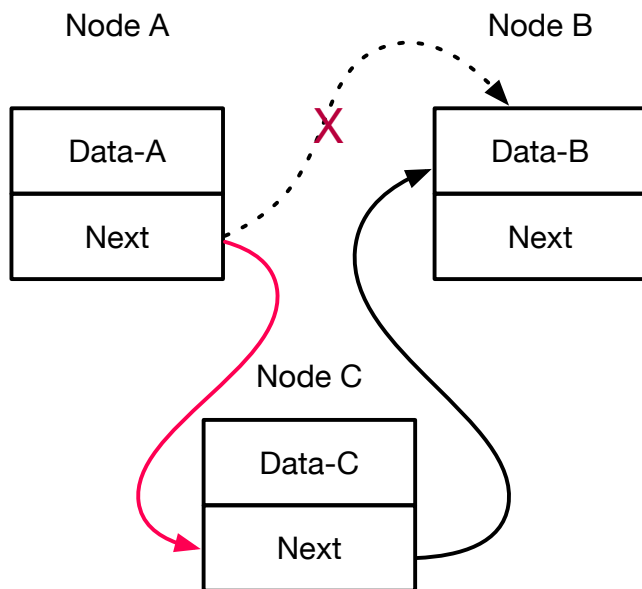
处理器任意一个操作的是否能够原子的可见，如更新一个指针

RCU 订阅/发布机制

以链表为例：插入结点Node C



① 填入Data与Pointer

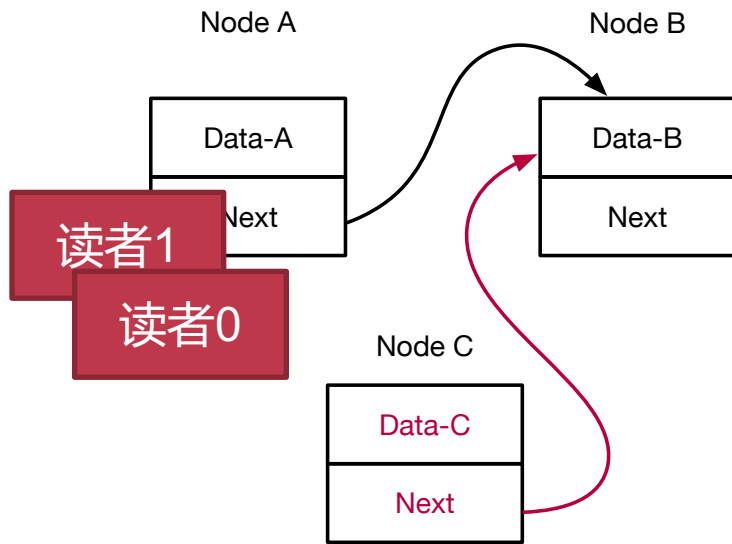


② 利用单拷贝原子性，原子地更新Node A的指针

读者看不到Node C

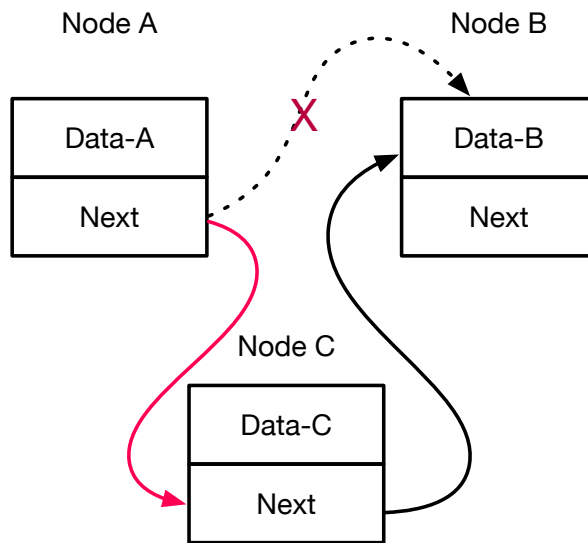
读者看到Node C 63

RCU 订阅/发布机制：此时的读者



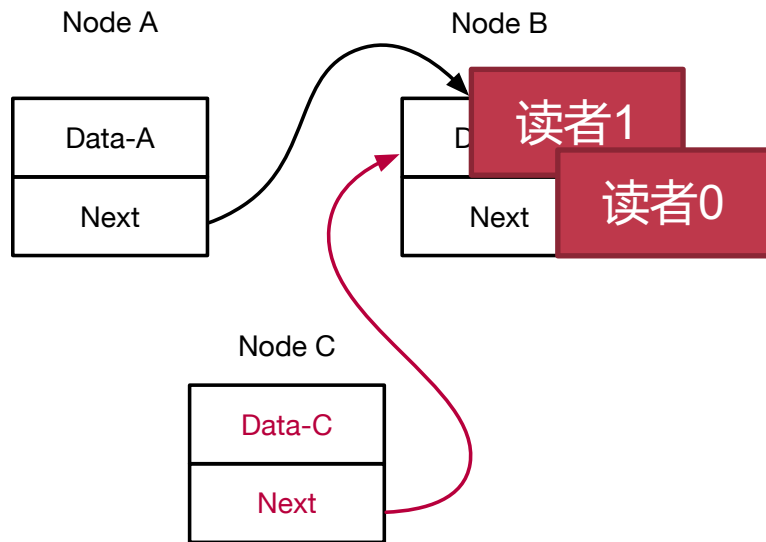
Writing Node C

① 填入Data与Pointer



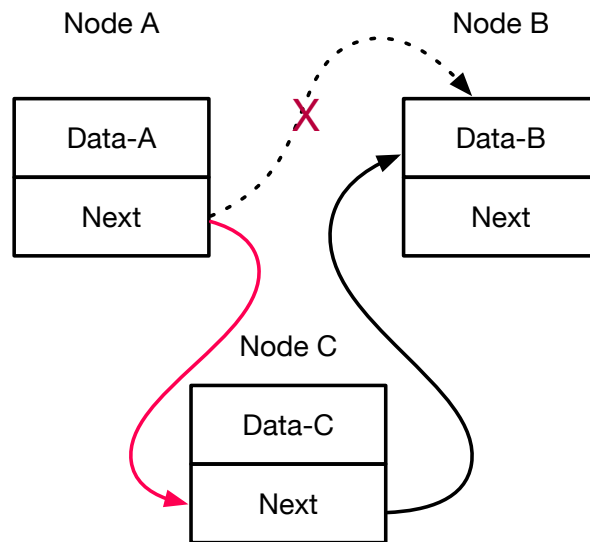
② 利用单拷贝原子性，原子地更新Node A的指针

RCU 订阅/发布机制：此时的读者



Writing Node C

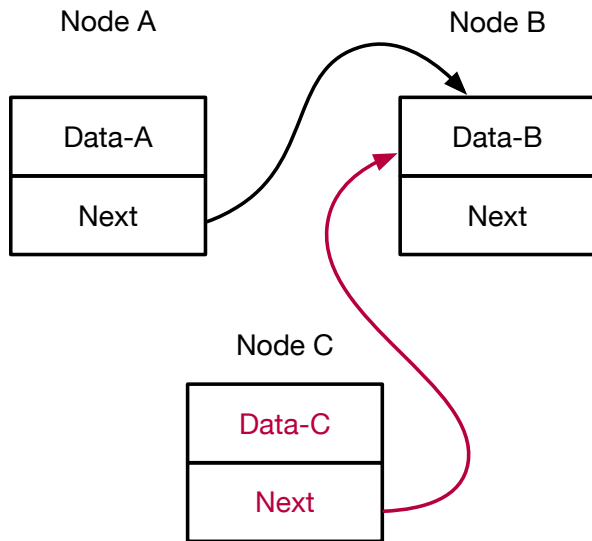
① 填入Data与Pointer



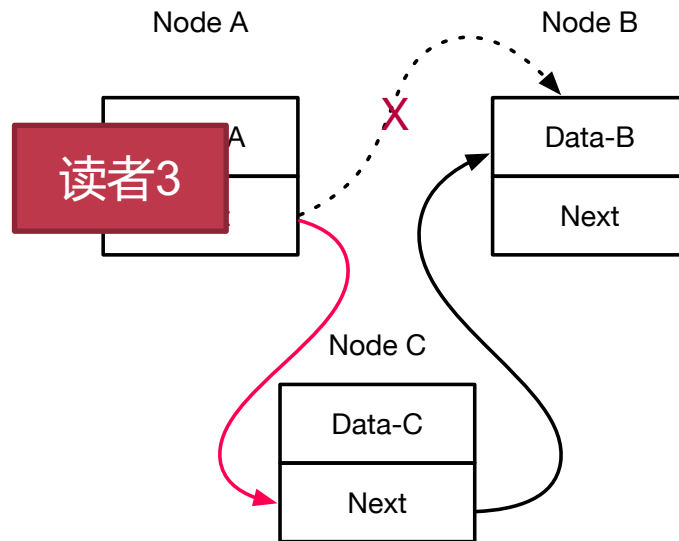
② 利用单拷贝原子性，原子地更新Node A的指针

看不到Node C

RCU 订阅/发布机制：此时的读者

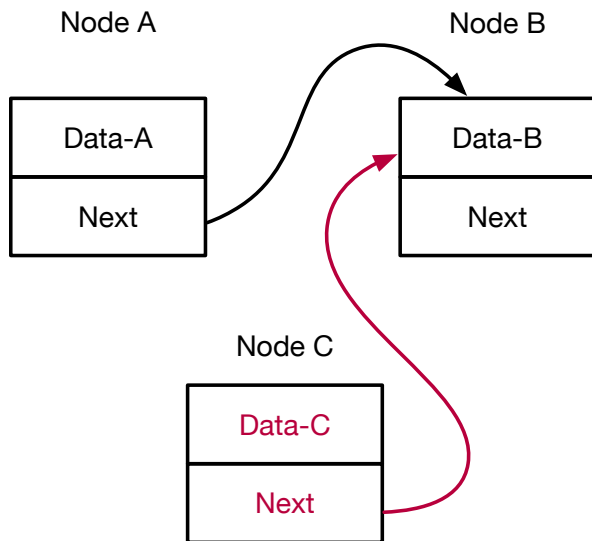


① 填入Data与Pointer

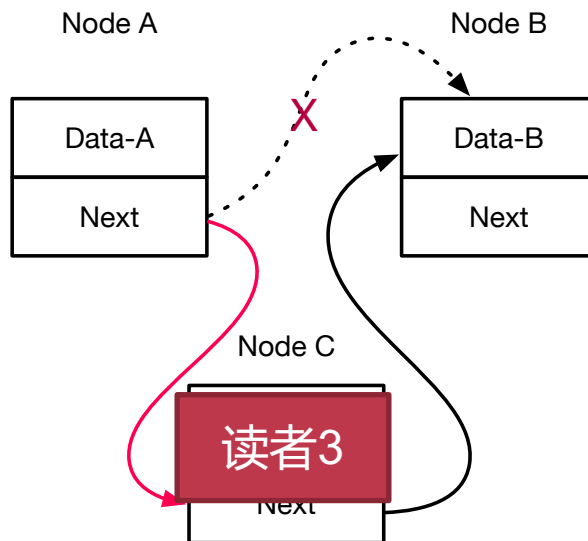


② 利用单拷贝原子性，原子地更新Node A的指针

RCU 订阅/发布机制：此时的读者



① 填入Data与Pointer

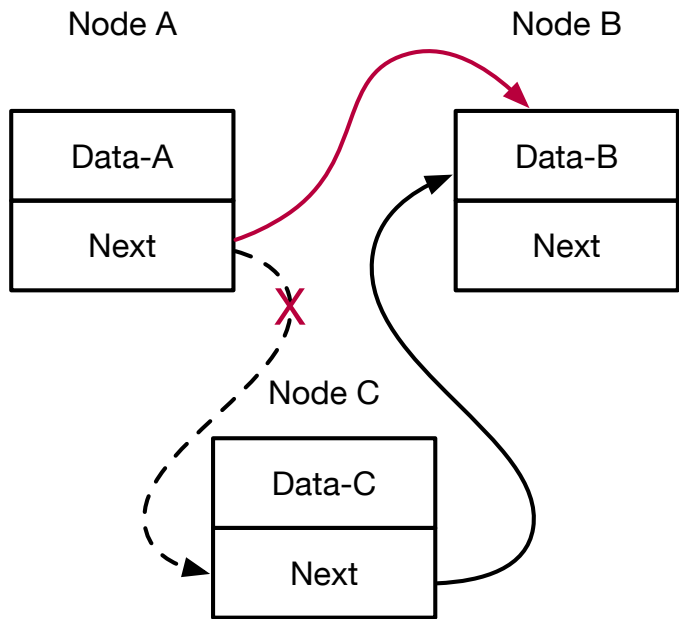


② 利用单拷贝原子性，原子地更新Node A的指针

可以看到Node C

RCU 订阅/发布机制

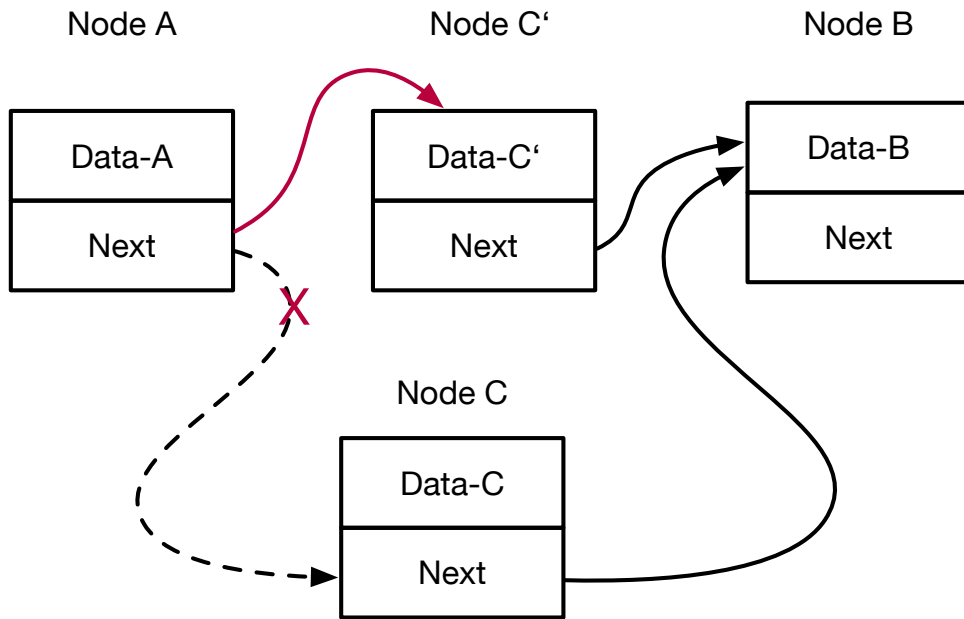
删除结点Node C



$A.Next = \&B$

思考：局限性在哪？

更新结点Node C



$A.Next = \&C'$

复制一个新的Node C'

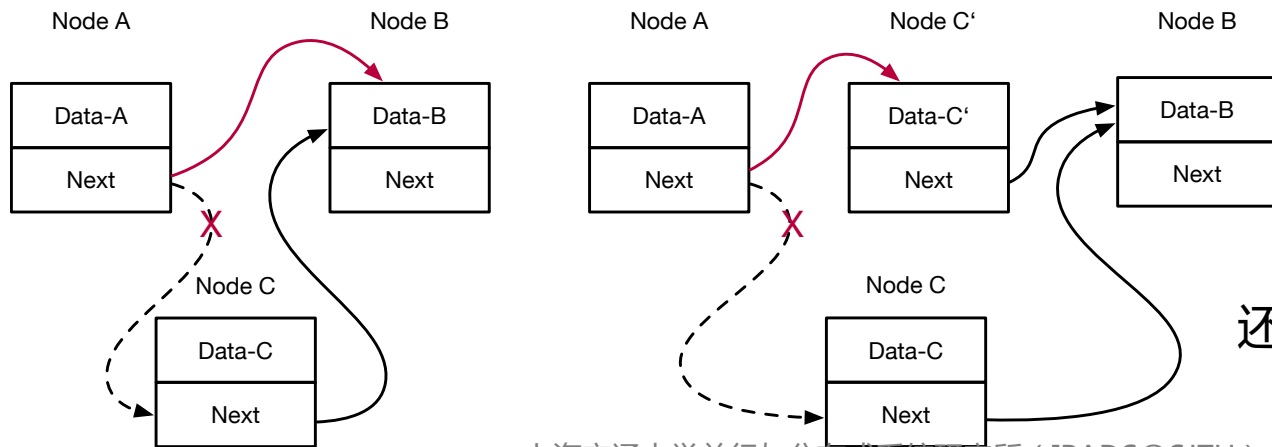
更新Next指针

Read Copy Update, RCU

读写锁读者进入读临界区之前，还是需要**繁杂的操作**

思考：局限性在哪？ 我们需要回收无用的旧拷贝

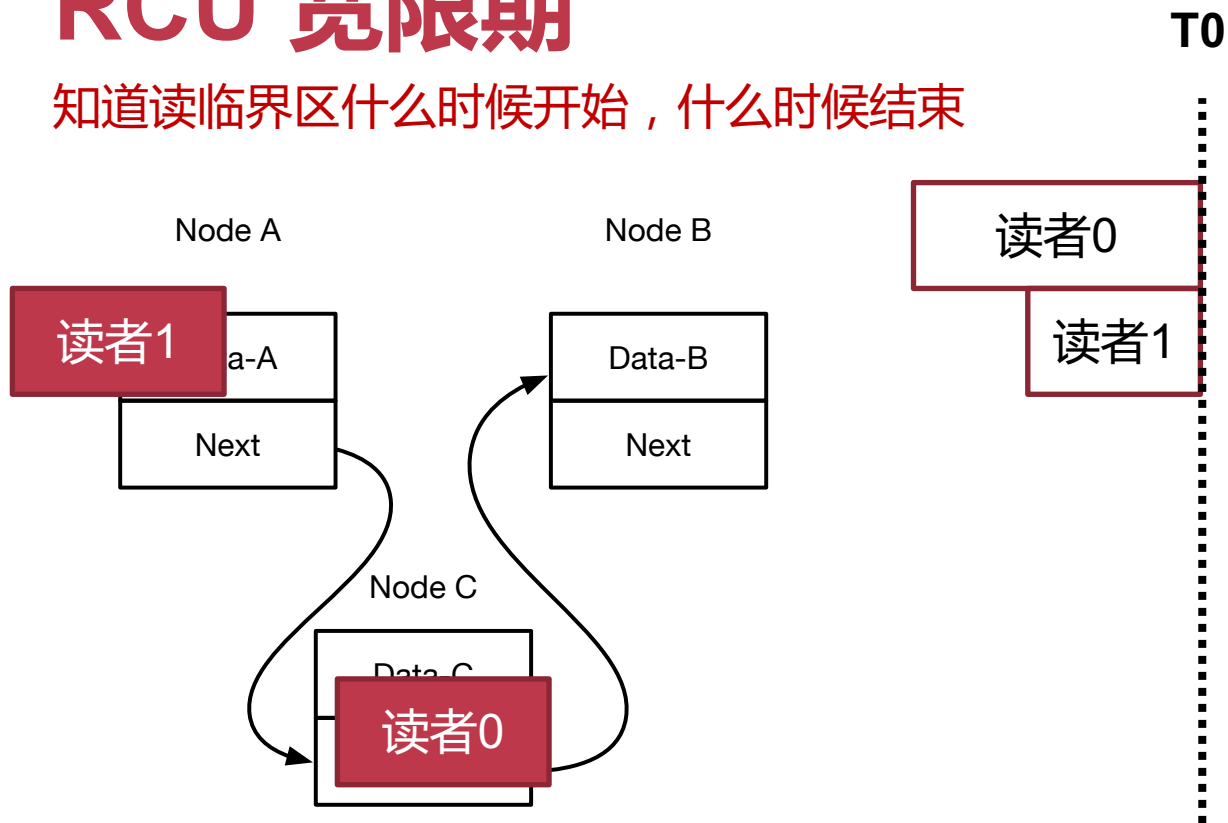
需求2：在**合适的时间**，**回收**无用的旧拷贝



更新完指针时
还有读者在Node C上读
此时不能回收

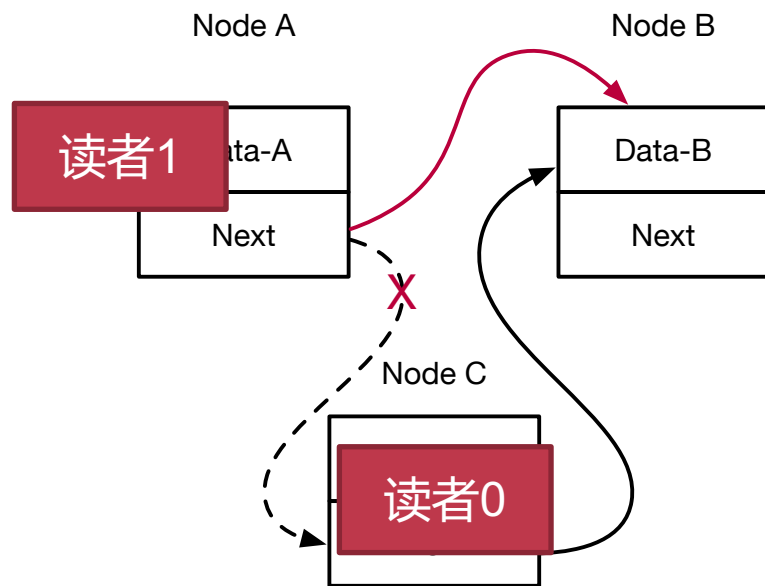
RCU 宽限期

知道读临界区什么时候开始，什么时候结束



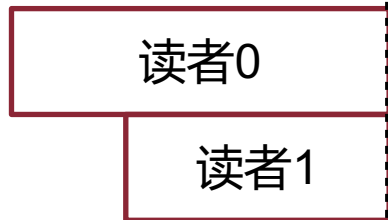
RCU 宽限期

知道读临界区什么时候开始，什么时候结束



$A.Next = \&B$

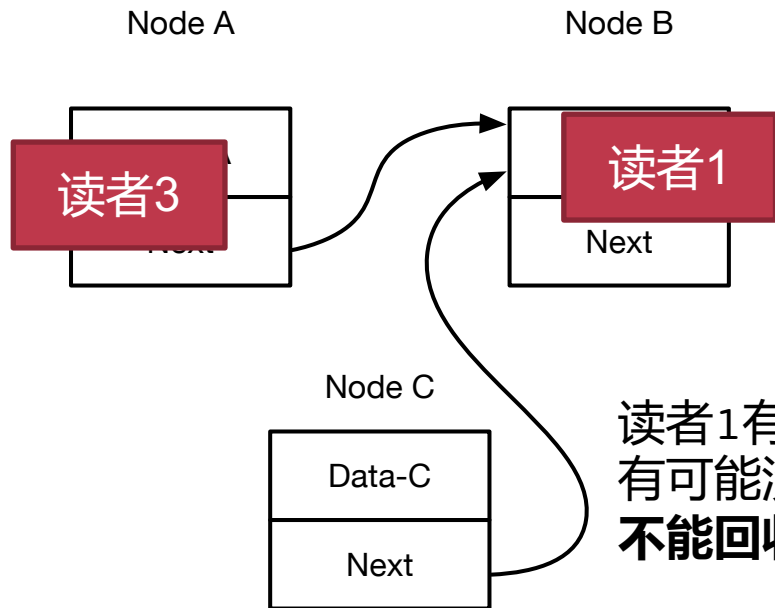
T0 T1



$A.Next = \&B$

RCU 宽限期

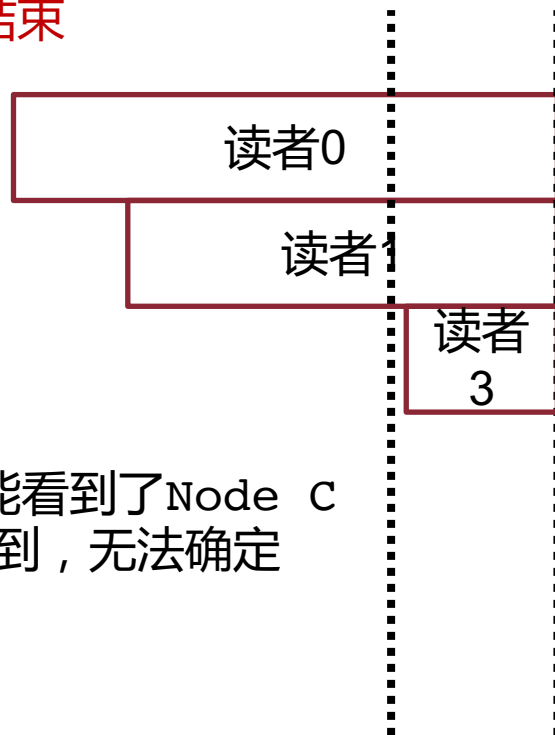
知道读临界区什么时候开始，什么时候结束



$A.Next = \&B$

读者1有可能看到了Node C
有可能没看到，无法确定
不能回收！

T0 T1 T2



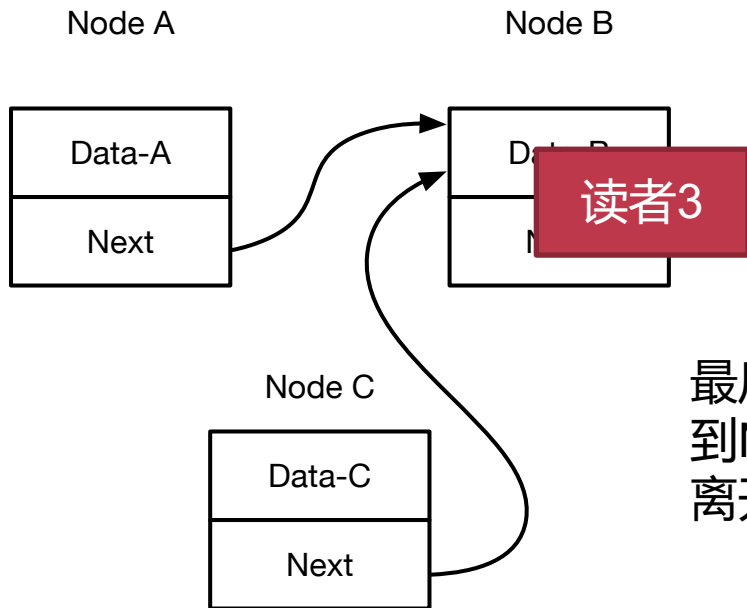
读者0结束

此时能回收吗？

$A.Next = \&B$

RCU 宽限期

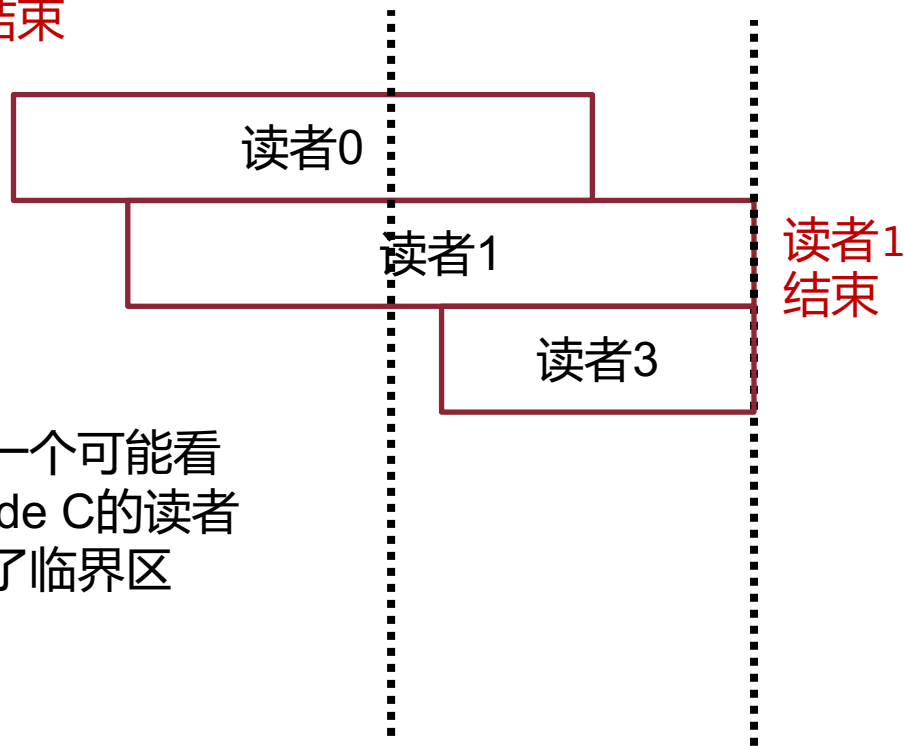
知道读临界区什么时候开始，什么时候结束



$A.Next = \&B$

最后一个可能看到Node C的读者离开了临界区

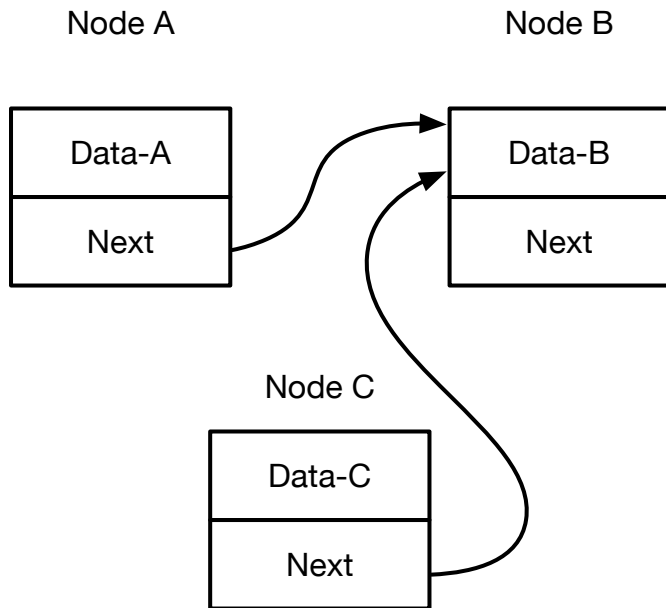
T0 T1 T2 T3



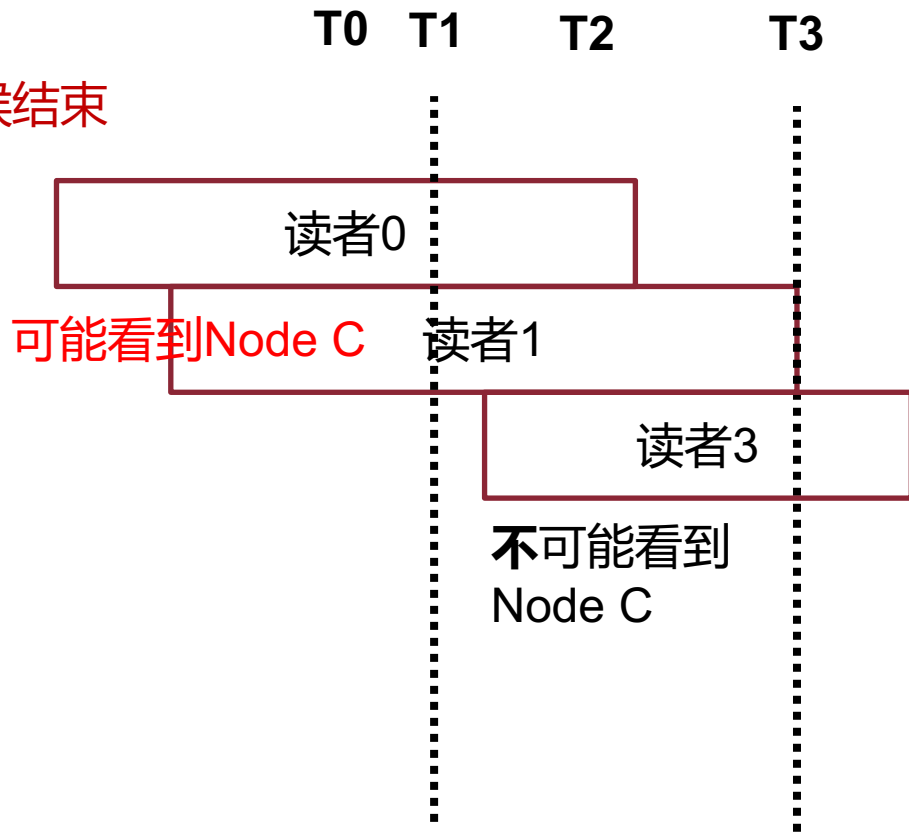
$A.Next = \&B$ 可以回收Node C

RCU 宽限期

知道读临界区什么时候开始，什么时候结束



$A.Next = \&B$



$A.Next = \&B$ 可以回收Node C

管程

开发者常常**用错**同步原语

管程提供一系列thread-safe的接口

开发者直接使用这些接口

```
monitor monitor name {  
    共享数据  
    function P1 () {  
        使用其他同步原语保证正确性  
    }  
    function P2 () {  
        使用其他同步原语保证正确性  
    }  
}
```

同步原语对比：读写锁 vs RCU

读写锁

RCU

相同点：

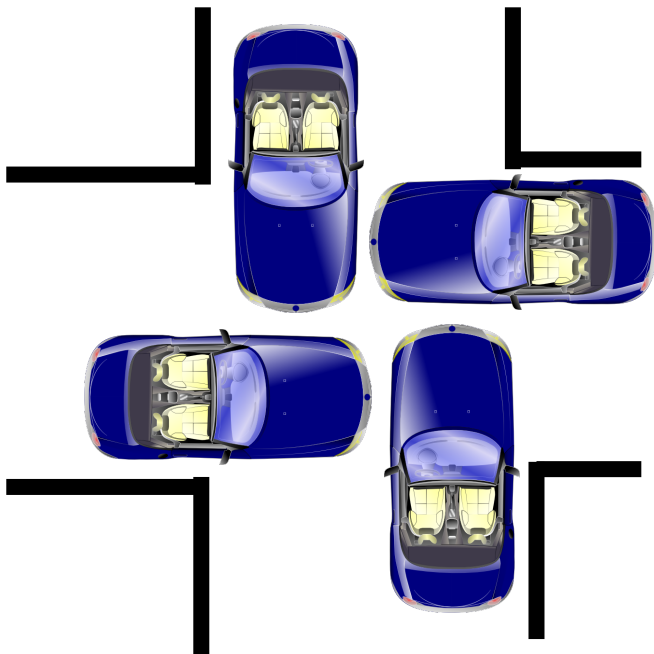
允许读者并行

不同点：

- 读者也需要上读者锁
- 读者无需上锁
- 关键路径上有额外开销
- 使用较繁琐
- 方便使用
- 写者开销大
- 可以选择对写者开销不大的读写锁

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

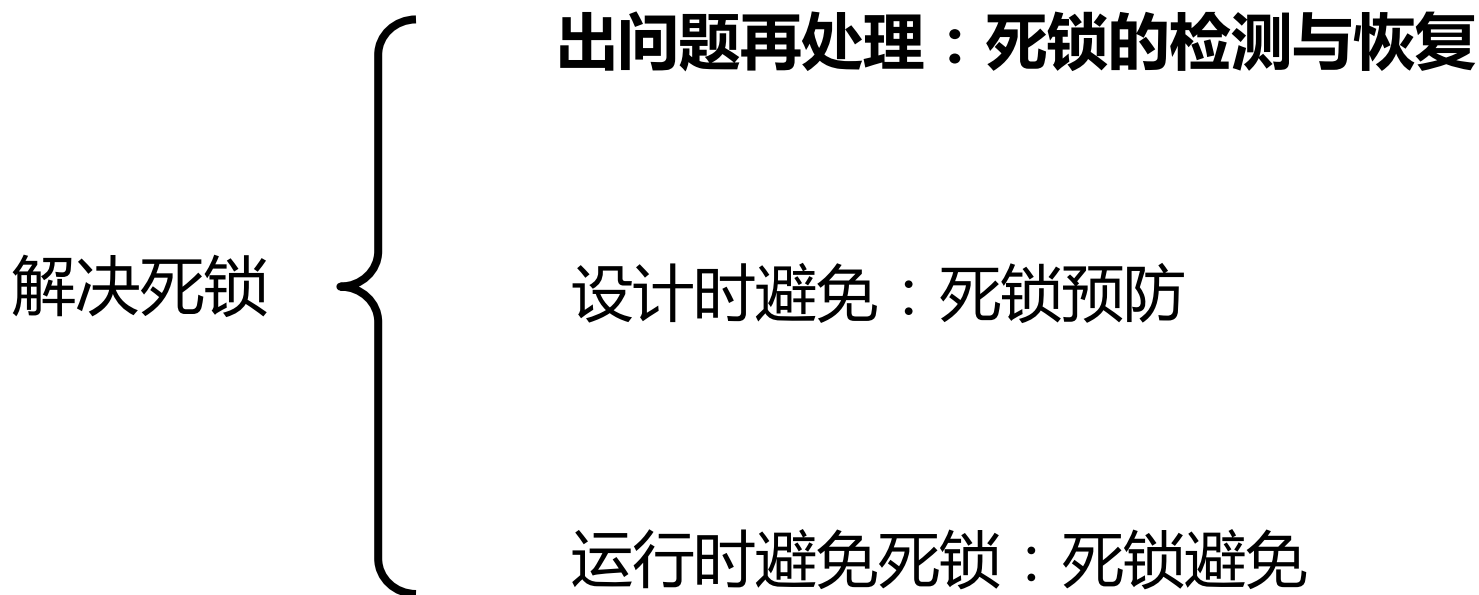
死锁产生的原因

- 互斥访问
- 持有并等待
- 资源非抢占
- 循环等待

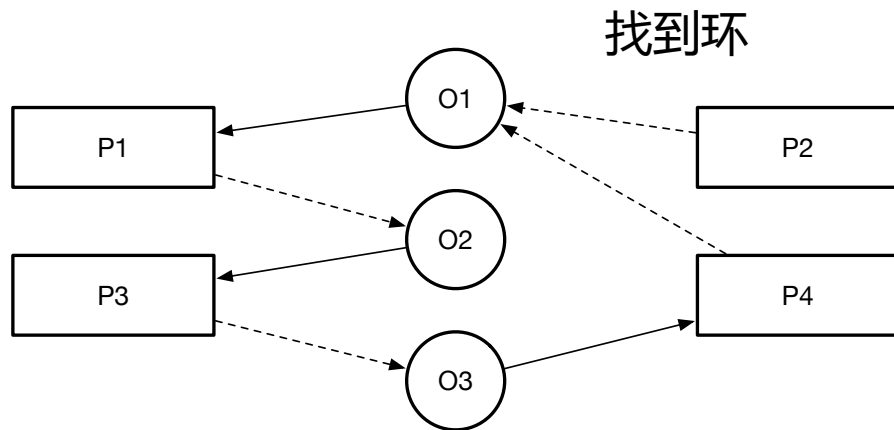
```
void proc_A(void) {  
    lock(A);  
    /* Time T1 */  
    lock(B);  
    /* Critical Section */  
    unlock(B);  
    unlock(A);  
}  
  
void proc_B(void) {  
    lock(B);  
    /* Time T1 */  
    lock(A);  
    /* Critical Section */  
    unlock(A);  
    unlock(B);  
}
```

T1时刻的死锁

如何解决死锁？



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

- 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

如何恢复？打破循环等待！

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

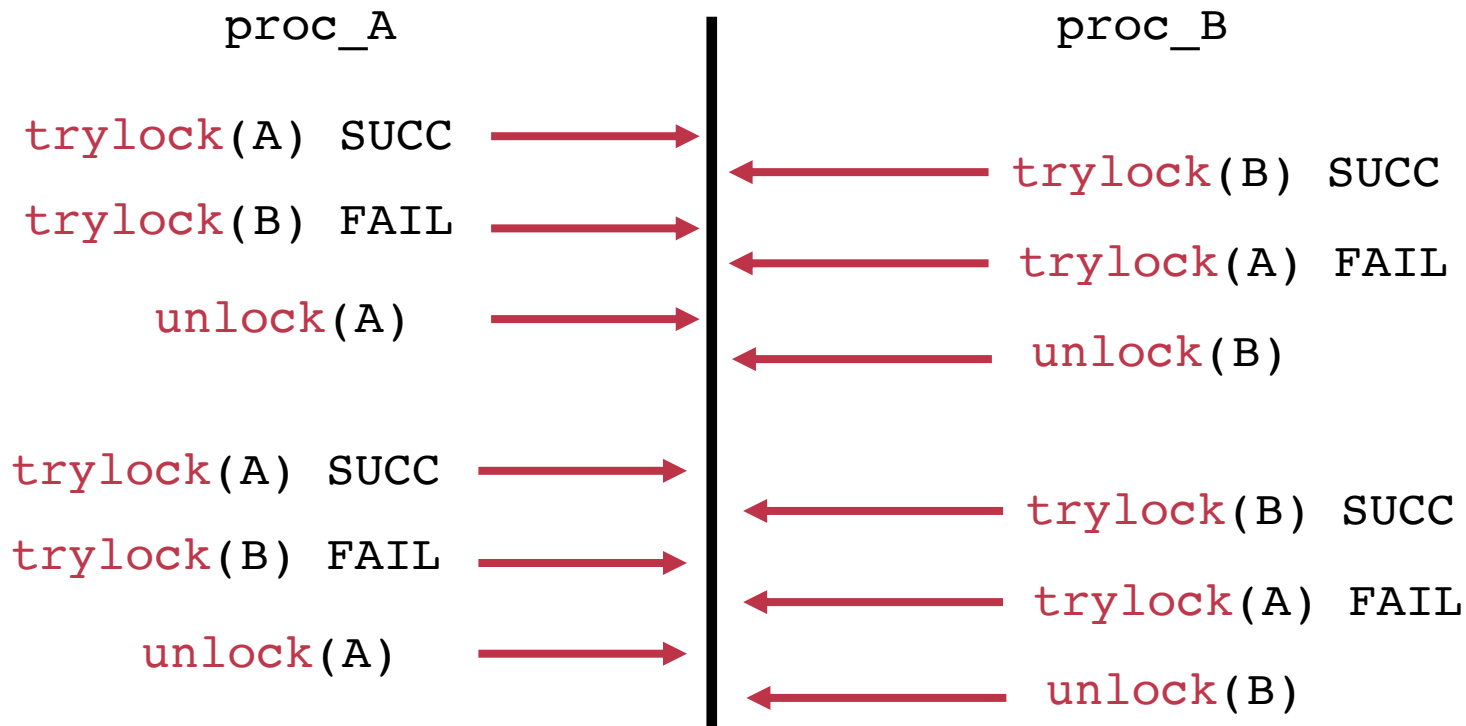
1. 避免互斥访问：通过其他手段（如代理执行）
2. 不允许持有并等待：一次性申请所有资源

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

trylock非阻塞
立即返回成功或失败

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**

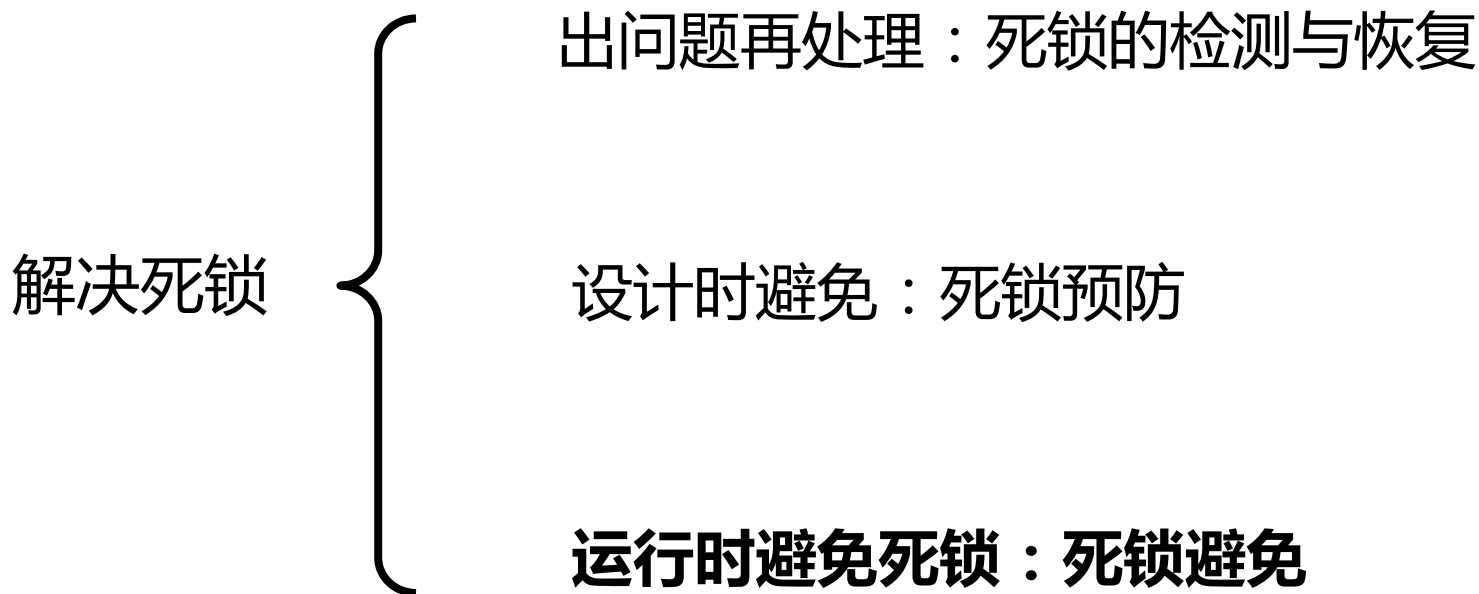
死锁预防：四个方向

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？



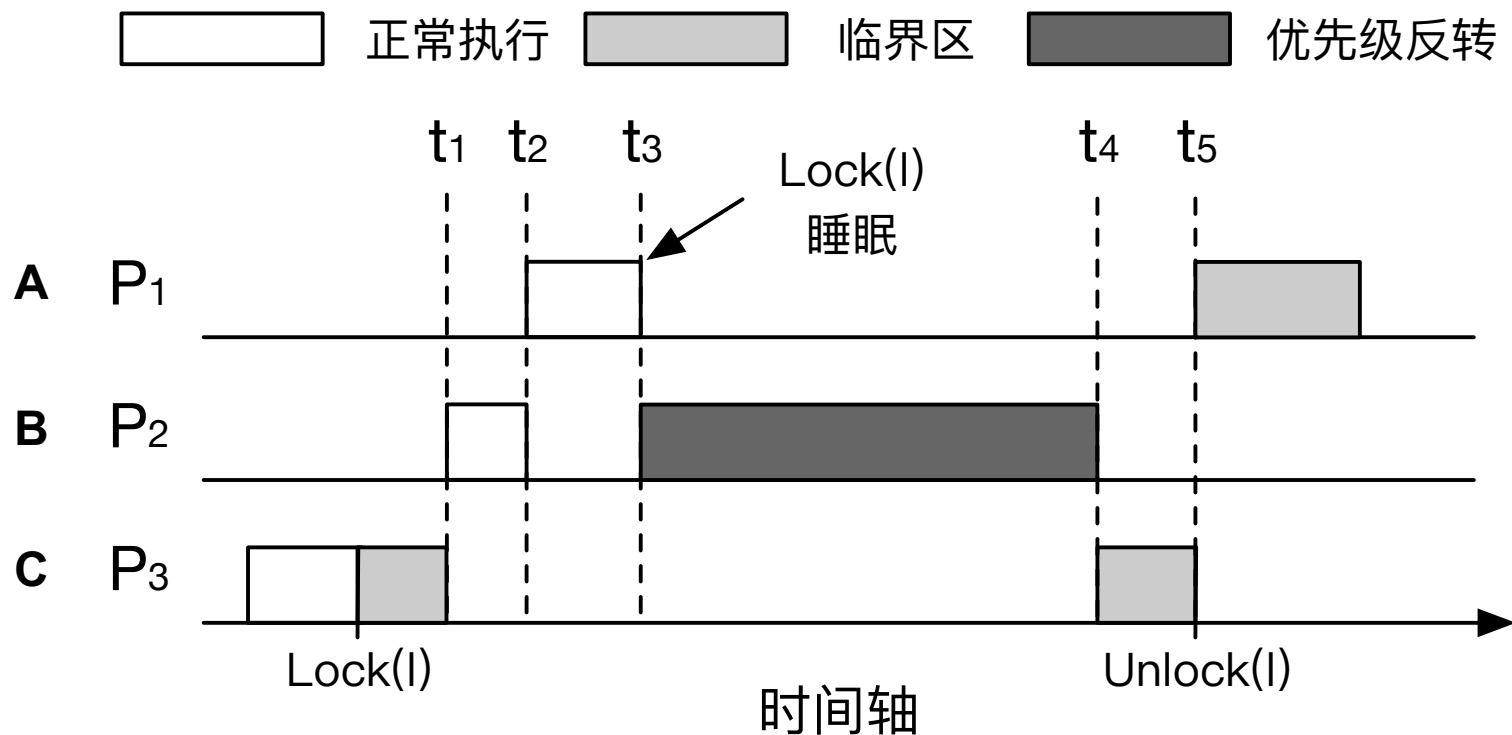
死锁避免：运行时检查是否会出现死锁

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



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

双重调度导致

锁：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

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：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

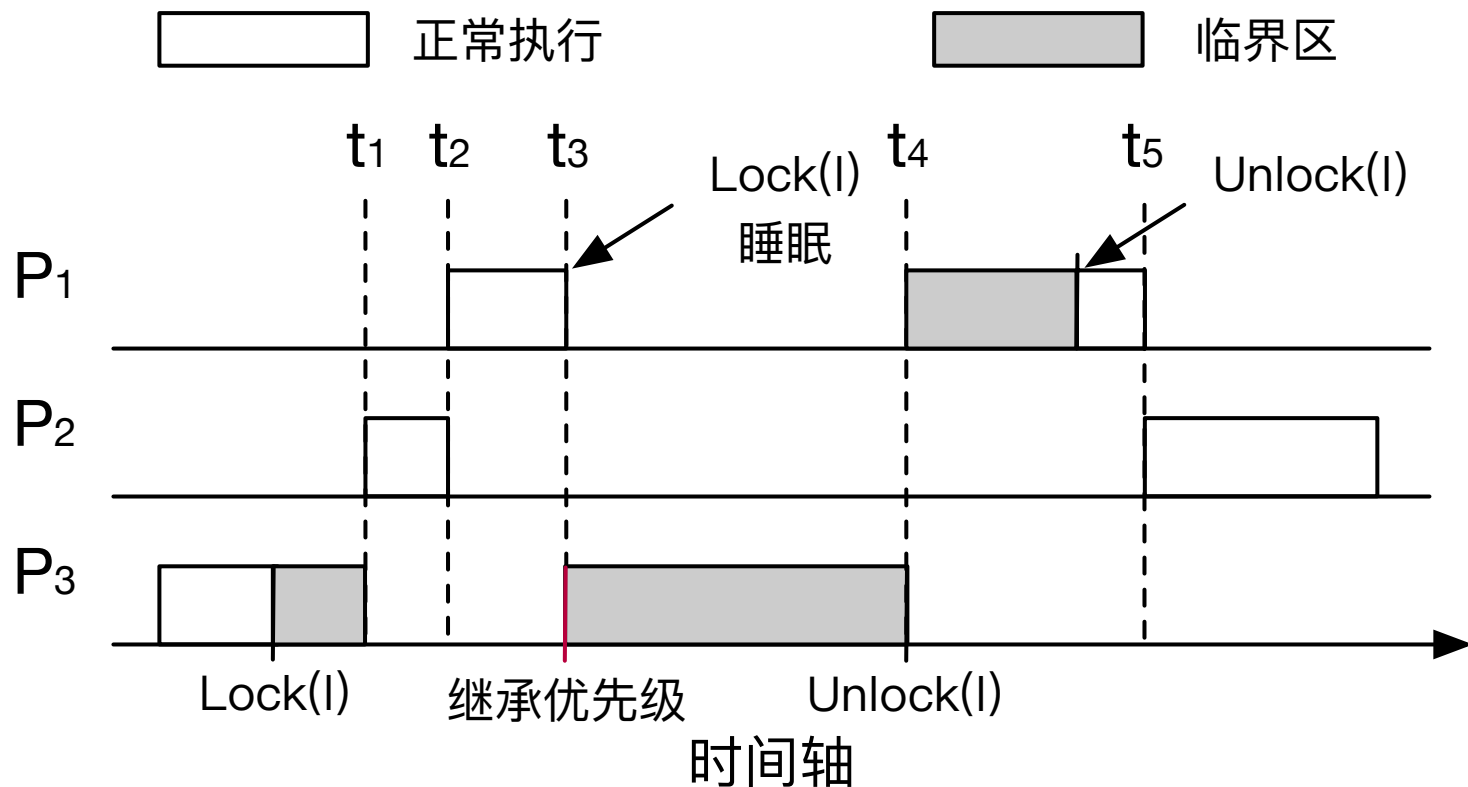
锁：按照锁使用的策略进行调度

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



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

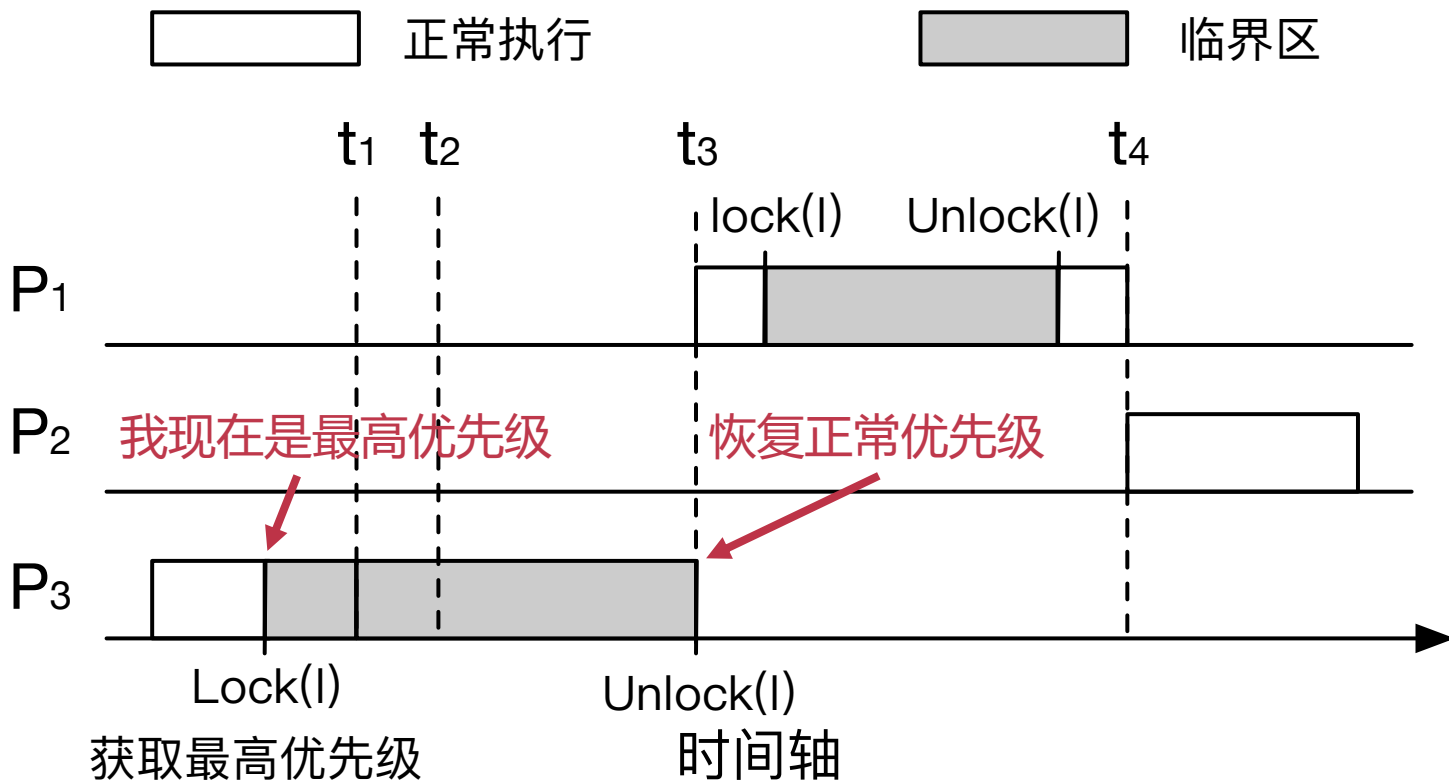
锁：按照锁使用的策略进行调度

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



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：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)
- 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

