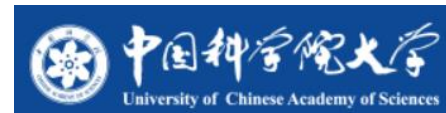




中国科学院软件研究所
Institute of Software, Chinese Academy
of Sciences



同步原语

郑晨

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
 - <https://ipads.se.sjtu.edu.cn/courses/os/index.shtml>
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。

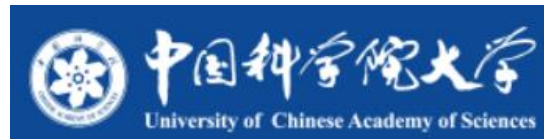


中国科学院软件研究所

Institute of Software, Chinese Academy of Sciences



上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

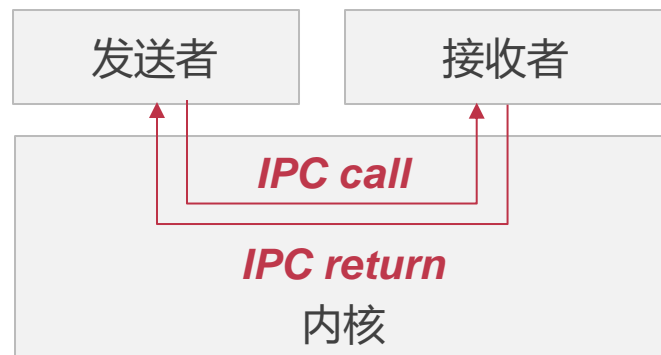


大纲

- 多线程问题：竞争条件
- 四种同步原语
 - 互斥锁：保证互斥访问
 - 条件变量：提供睡眠/唤醒
 - 信号量：资源管理
 - 读写锁：区分读者以提高并行度
- 同步原语带来的问题
 - 死锁的检测、预防与避免

回顾：进程间通讯

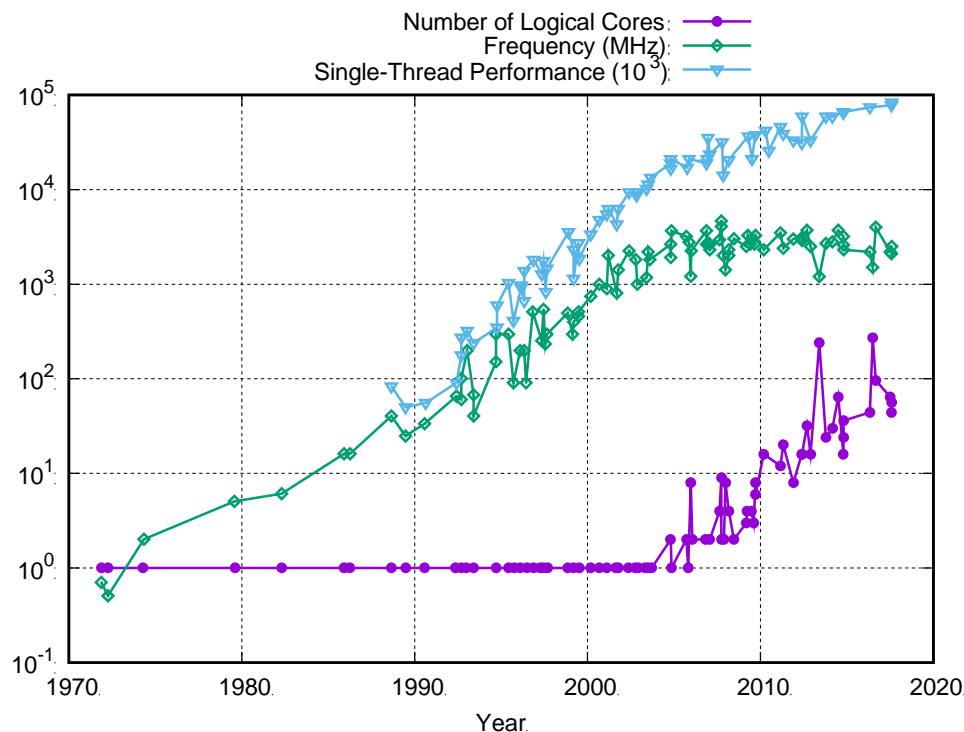
- **进程间通信**: 两个(或多个)不同的进程，通过内核或其他共享资源进行通信，来传递控制信息或数据
 - 直接通讯/间接通讯
- **进程间协作**: 基于消息传递的抽象



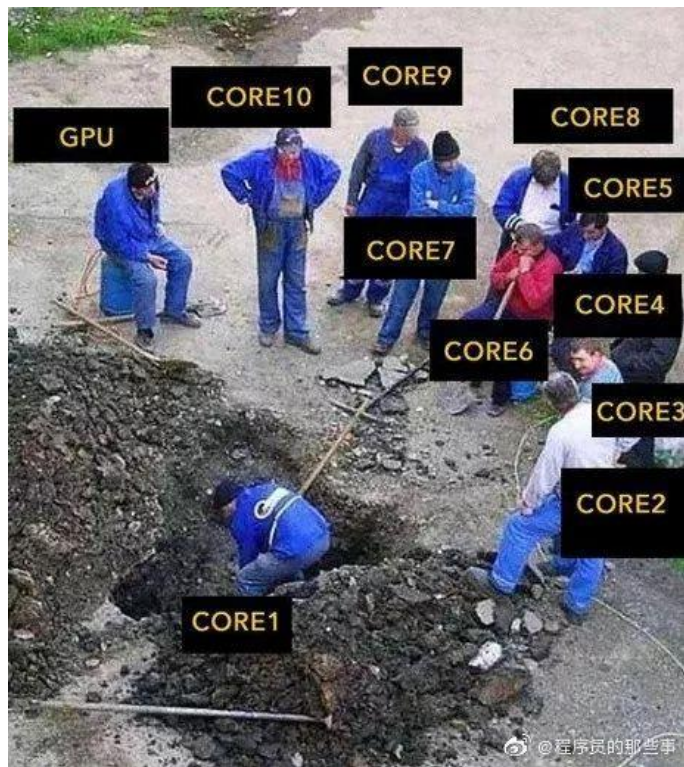
今天的主题：直接基于**共享内存**操作（如发送者直接修改全局变量）

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

四个场景与对应的同步原语

场景一：共享资源互斥访问

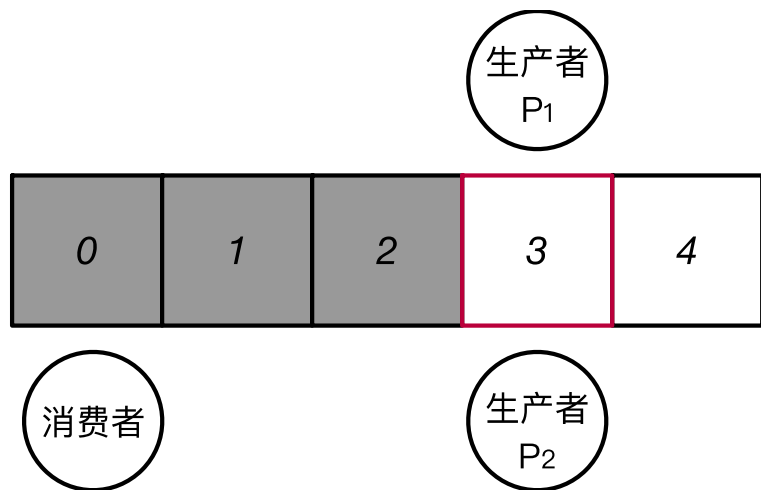
多个线程需要同时访问同一共享数据

应用程序需要保证**互斥访问**避免数据竞争

```
int shared_var = 0;
void thread_1(void) {
    shared_var = shared_var + 1;
}
```

```
void thread_2(void) {
    shared_var = shared_var - 1;
}
```

使用**互斥锁**保证**互斥访问**



回顾：多生产者之间协同

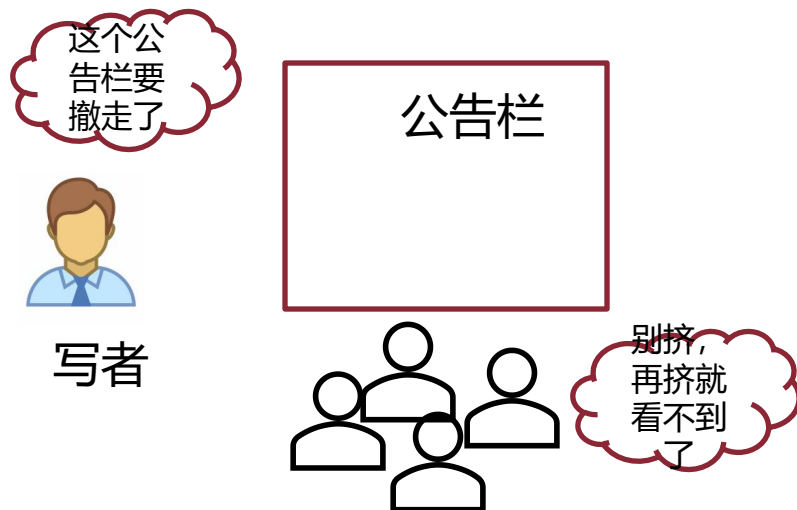
衍生场景一：读写场景并发读取

多个线程**只会读取**共享数据

允许读者线程**并发执行**

```
int shared_var;  
void reader(void) {  
    local_var = shared_var;  
}  
  
void writer(void) {  
    shared_var = shared_var++;  
}
```

可使用**读写锁**提升读者并行度



回顾：公告栏问题

场景二：条件等待与唤醒

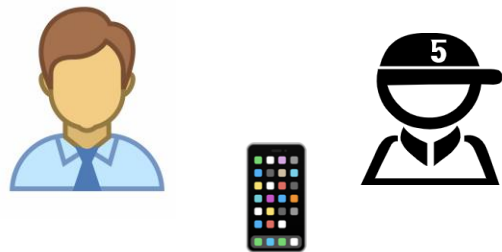
线程等待某条件时**睡眠**

达成该条件后**唤醒**

```
void thread_1(void) {  
    doing_something; /* 完成当前线程的工作 */  
    notify_thread_2; /* 通知线程2完成 */  
}
```

```
void thread_2(void) {  
    if (thread_1_not_finish)  
        wait; /* 等待线程1完成其工作 */  
    doing_something; /* 完成线程2的工作 */  
}
```

使用**条件变量**完成线程睡眠/唤醒



快递员送快递

场景三：多资源协调管理

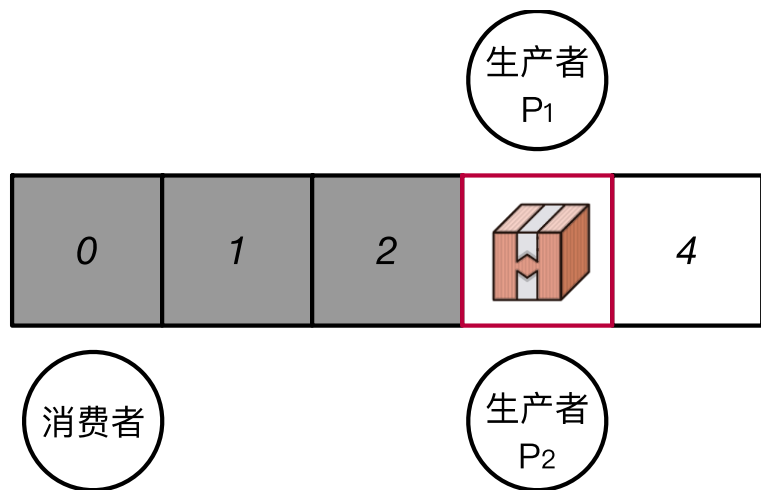
多个资源可以被多个线程**消耗或释放**

正确协同线程获取资源或等待

```
void producer_thread(void) {  
    release_resource(shared_resources);  
    notify_waiters;  
}
```

```
void consumer_thread(void) {  
    if (not_have_resources)  
        wait;  
    consume_resource(resource);  
}
```

使用**信号量**完成资源管理与线程协同



回顾：生产者消费者之间协同

四个场景与同步原语

同步原语	描述	使用场景
互斥锁	保证对共享资源的 互斥访问	场景一 共享资源互斥访问
读写锁	允许读者线程 并发读取 共享资源	衍生场景一 读写场景并发读取
条件变量	提供线程 睡眠 与 唤醒 机制	场景二 条件等待与唤醒
信号量	协调 有限数量 资源的消耗与释放	场景三 多资源协调管理

同步与临界区

生产者消费者问题的基础实现

- 基础实现: 生产者

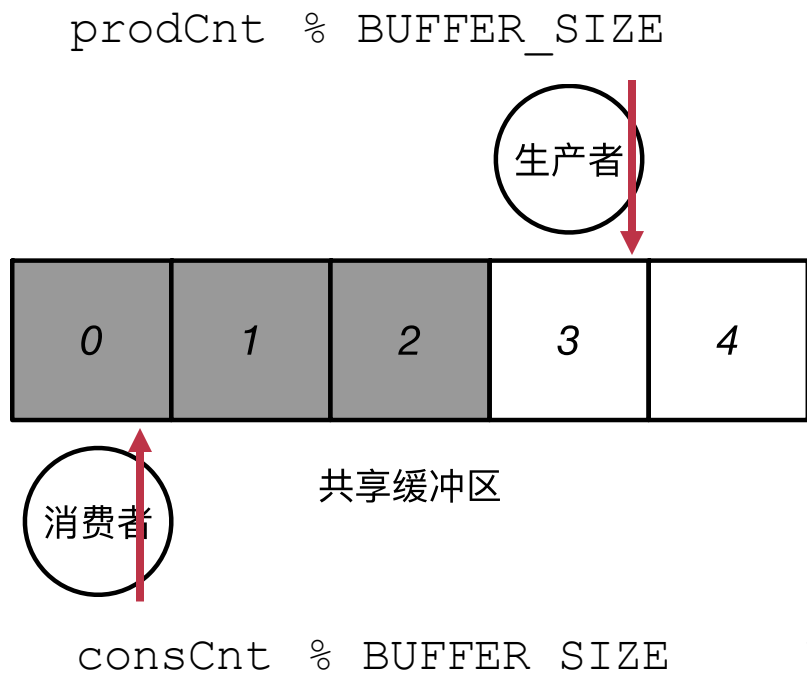
```
while (true) {  
    /* Produce an item */  
    while (prodCnt - consCnt == BUFFER_SIZE)  
        ; /* do nothing -- no free buffers */  
    buffer[prodCnt % BUFFER_SIZE] = item;  
    prodCnt = prodCnt + 1;  
}
```

生产者消费者问题的基础实现

- 基础实现: 消费者

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

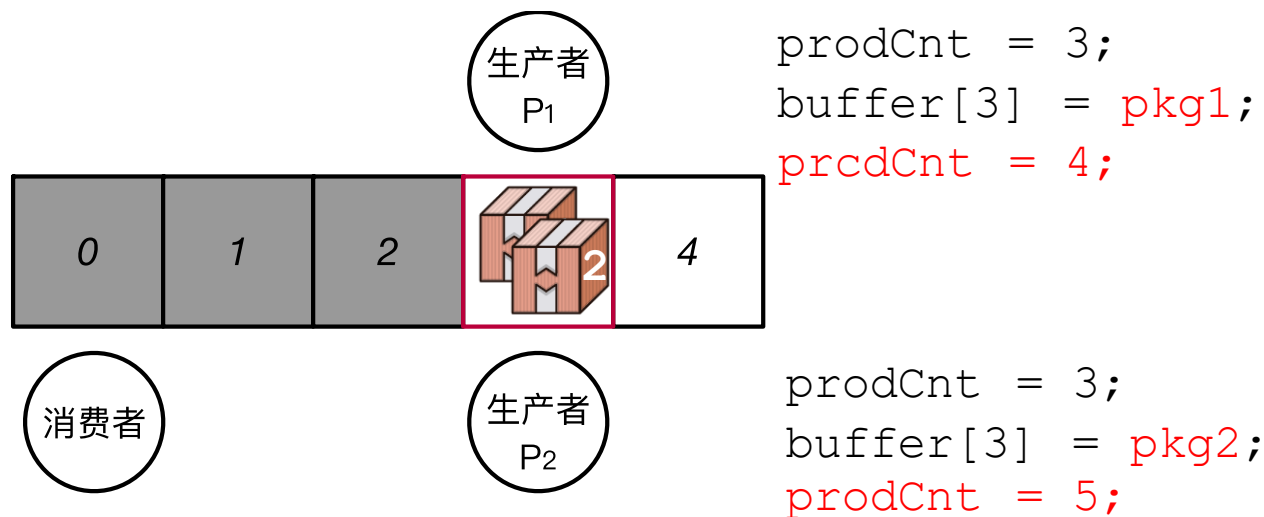

生产者消费者问题方案总结



通过两个计数器来协调生产者与消费者，
是少数符合竞争定义却没有竞争问题的实现

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;*
```



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，防止**数据覆盖**？

*这里假设该操作为原子操作

新的抽象：临界区 (Critical Section)

```
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

```
}
```

任意时刻，有且只有一个线程
可以进入临界区执行

实现临界区抽象的三个要求

- **互斥访问**：在同一时刻，**有且仅有一个线程**可以进入临界区
- **有限等待**：当一个线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
- **空闲让进**：当没有线程在临界区中时，必须在申请进入临界区的线程中选择一个进入临界区，保证执行临界区的**进展**

```
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

```
}
```

什么是同步原语?

同步原语 (Synchronization Primitives) 是一个平台 (如**操作系统**) 提供的用于帮助开发者实现线程之间**同步**的**软件工具**

在生产者/消费者例子中:

有限的共享资源上

正确的协同工作



有限的共享缓冲区;

生产者/消费者能有序地从
共享缓冲区中存放/拿取数据

解决并发冲突的基本思路

- **破坏资源共享条件**
 - 将共享资源更改为非共享资源
- **破坏时间重叠条件**
 - 打乱时间重叠的执行次序
 - 单核系统中，禁用中断，消除可能的进程执行交叉
 - 早起的Linux内核中用全局锁，现在呢？

解决并发冲突的主要技术

技术手段		
Per-CPU变量	将共享资源split为非共享资源	破坏资源共享条件
资源静态划分	同上	同上
原子操作	通过原子操作分割并发访问	解除冲突的时间条件
内存屏障	限定全局内存访问顺序	
锁、信号量	通过锁协调机制，分离访问	解除冲突的时间条件
RC锁	通过指针实现无锁访问	同上
禁止中断、软中断	通过禁止中断，避免并发执行	同上

解决临界区问题的三个要求

1. **互斥访问**：在同一时刻，**有且仅有一个进程**可以进入临界区
2. **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
3. **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**

```
while(TRUE) {
```

申请进入临界区

临界区部分

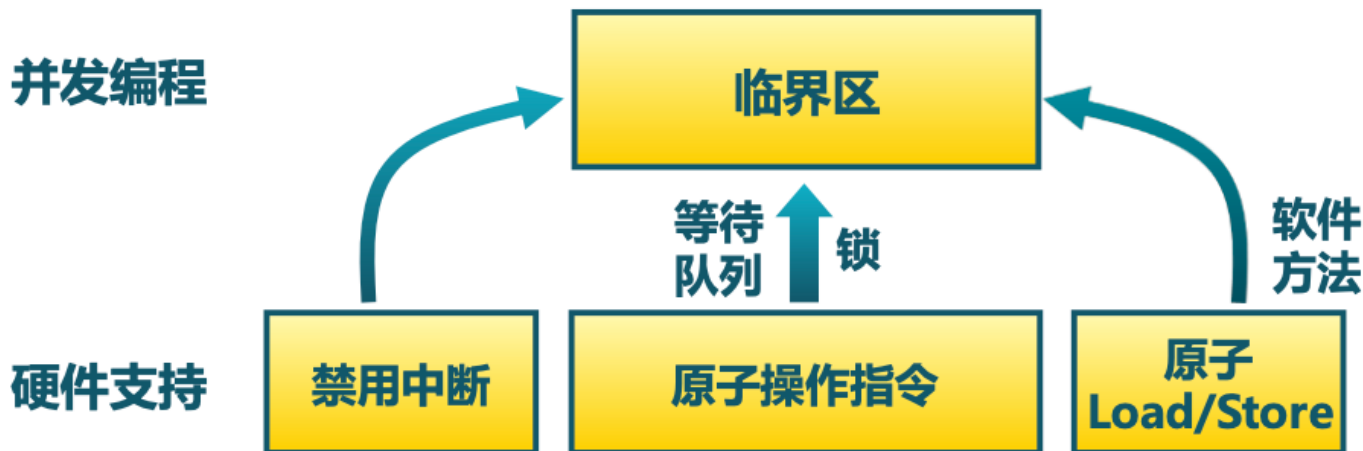
通知退出临界区

其他代码

```
}
```


临界区的保障

- 硬件中断
- 原子操作指令与互斥锁
- 基于软件



软件解决方案：皮特森算法

线程 - 0

```
1. while(TRUE) {  
2.     flag[0] = true;  
3.     turn = 1;  
4.     while (flag[1] == true &&  
5.         turn == 1);
```

临界区部分

```
6.     flag[0] = false;
```

其他代码

```
7. }
```

线程 - 1

```
1. while(TRUE) {  
2.     flag[1] = true;  
3.     turn = 0;  
4.     while (flag[0] == true &&  
5.         turn == 0);
```

临界区部分

```
6.     flag[1] = false;
```

其他代码

```
7. }
```

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

Dekkers算法

```
flag[0] := false; flag[1] := false; turn := 0; // or 1
do {
    flag[i] = true;
    while flag[j] == true {
        if turn ≠ i {
            flag[i] := false
            while turn ≠ i { }
            flag[i] := true
        }
    }
    CRITICAL SECTION
    turn := j
    flag[i] = false;
    EMAINDER SECTION
} while (true);
```

有没有更简单的方法？比如关闭中断？

```
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

开启所有核心的中断

有没有更简单的方法？比如关闭中断？

```
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

禁止中断实现互斥

- 禁用中断以屏蔽外部事件
 - 引入不可中断的代码区域
 - 大多数时候用串行思维
 - 延迟处理外部事件

```
struct lock {  
}  
void acquire (lock) {  
    disable interrupts;  
}  
void release (lock) {  
    enable interrupts;  
}
```

禁止中断

- 使用锁变量

```
Acquire(lock)
{
    disable interrupts;
    while (lock.value != FREE)
        ;
    lock.value = BUSY;
    enable interrupts;
}
```

```
Release(lock)
{
    disable interrupts;
    lock.value = FREE;
    enable interrupts;
}
```

- 问题?

禁止中断

- 使用锁变量，并只在对锁变量进行测试和赋值时通过中断实现互斥

```
Acquire(lock)
{
    disable interrupts;
    while (lock.value != FREE) {
        enable interrupts;
        disable interrupts;
    }
    lock.value = BUSY;
    enable interrupts;
}
```

```
Release(lock)
{
    disable interrupts;
    lock.value = FREE;
    enable interrupts;
}
```


禁止中断

- 引入队列

```
Acquire(lock)
{
    disable interrupts;
    while (lock.value == BUSY){
        add TCB to wait queue q;
        Yield();
    }
    lock.value = BUSY;
    enable interrupts;
}
```

```
Release(lock)
{
    disable interrupts;
    if (q is not empty) {
        remove thread t from q
        Wakeup(t);
    }
    lock.value = FREE;
    enable interrupts;
}
```

互斥锁

互斥锁的接口：拿锁和放锁

- **互斥锁 (Mutual Exclusive Lock) 接口**
 - Lock(lock): 尝试拿到锁 “lock”
 - 若当前没有其他线程拿着lock, 则拿到lock, 并继续往下执行
 - 若lock被其他线程拿着, 则不断循环等待放锁 (busy loop)
 - Unlock(lock)
 - 释放锁
- **保证同时只有一个线程能够拿到锁**

用互斥锁解决多生产者消费者问题

```
while (prodCnt - consCnt == BUFFER_SIZE)
    ;    /* do nothing -- no free buffers */
```

```
lock(&buffer_lock);    // 申请进入临界区
```

```
buffer[bufCnt % BUFFER_SIZE] = item;
bufCnt = bufCnt + 1;
```

临界区

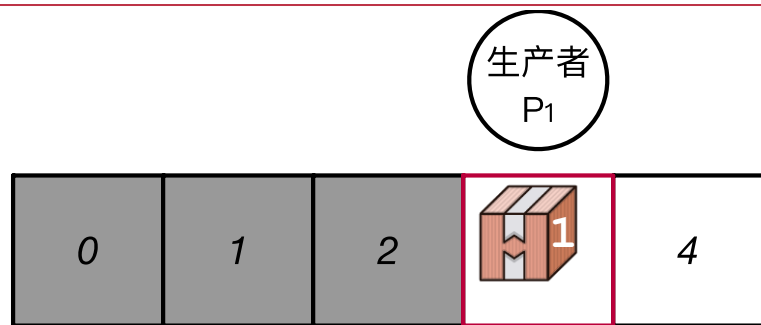
```
unlock(&buffer_lock);    // 通知离开临界区
```

```
prodCnt = prodCnt + 1;*
```

*这里假设该操作为原子操作

用互斥锁解决多生产者消费者问题

```
lock(&buffer_lock);  
buffer[bufCnt % BUFFER_SIZE] = item;  
bufCnt = bufCnt + 1;  
unlock(&buffer_lock);
```



生产者
P₁

```
(bufCnt = 3)  
lock(&buffer_lock);  
buffer[3] = pkg1;
```

获取互斥锁
进入临界区

消费者

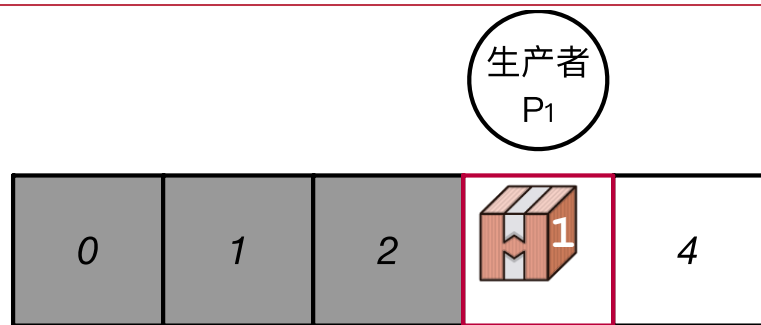
```
(bufCnt = 3)  
lock(&buffer_lock);
```

没有获取互斥锁,
在原地等待

*这里假设该操作为原子操作

用互斥锁解决多生产者消费者问题

```
lock(&buffer_lock);  
buffer[bufCnt % BUFFER_SIZE] = item;  
bufCnt = bufCnt + 1;  
unlock(&buffer_lock);
```



```
(bufCnt = 3)  
lock(&buffer_lock);  
buffer[3] = pkg1;  
bufCnt = 4  
unlock(&buffer_lock);
```

获取互斥锁
进入临界区



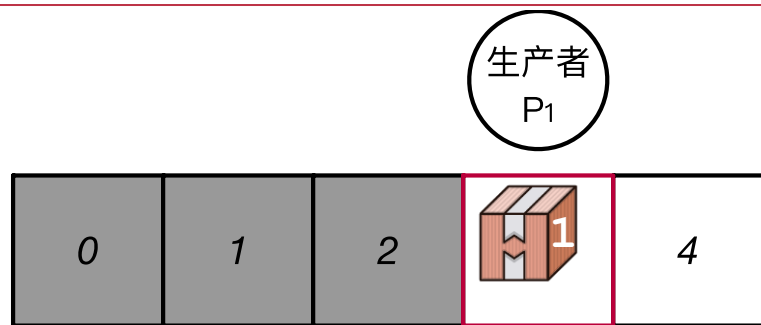
```
(bufCnt = 4)  
lock(&buffer_lock);
```

没有获取互斥锁,
在原地等待

*这里假设该操作为原子操作

用互斥锁解决多生产者消费者问题

```
lock(&buffer_lock);  
buffer[bufCnt % BUFFER_SIZE] = item;  
bufCnt = bufCnt + 1;  
unlock(&buffer_lock);
```



```
(bufCnt = 3)  
lock(&buffer_lock);  
buffer[3] = pkg1;  
bufCnt = 4  
unlock(&buffer_lock);
```



```
(bufCnt = 4)  
lock(&buffer_lock);  
buffer[4] = pkg2;
```

获取互斥锁
进入临界区

*这里假设该操作为原子操作

用互斥锁解决多线程计数问题

创建3个线程，同时执行下面程序：

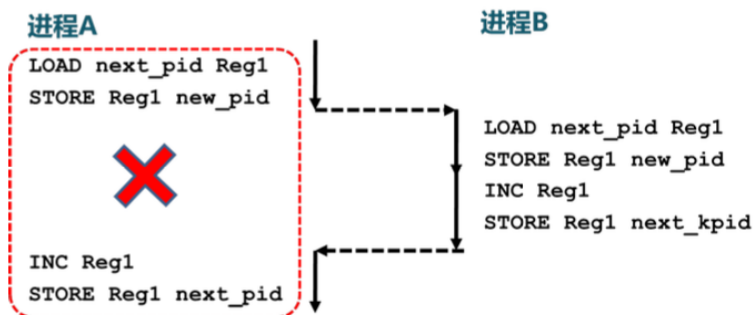
```
unsigned long a = 0;
void *routine(void *arg) {
    for (int i = 0; i < 1000000000; i++) {
        pthread_mutex_lock(&global_lock);
        a++;
        pthread_mutex_unlock(&global_lock);
    }
    return NULL;
}
```

pthread库提供的互斥锁实现

输出结果为： 3000000000

原子操作 (Atomic Operation)

- 原子操作是指一次不存在任何中断或失败的操作
 - 操作成功完成退出
 - 操作没有执行
 - 不会出现部分执行的状态
- 对临界区的操作必须是原子操作



- 操作系统需要利用同步机制在并发执行的同时，保证一些操作为原子操作

原子操作指令

- **测试和置位(Test-and-Set, TAS/TS)指令**
 - 从内存单元中读取旧的值
 - 设置该值为真
 - 返回旧的值到内存单元

```
bool test_and_set (bool *flag) {  
    bool old = *flag;  
    *flag = True;  
    return old;  
}
```

原子操作指令

- While循环什么时候返回

? Held的值是?

```
struct lock {  
    int held = 0;  
}  
void acquire (lock) {  
    while (test-and-set(&lock→held));  
}  
void release (lock) {  
    lock→held = 0;  
}
```

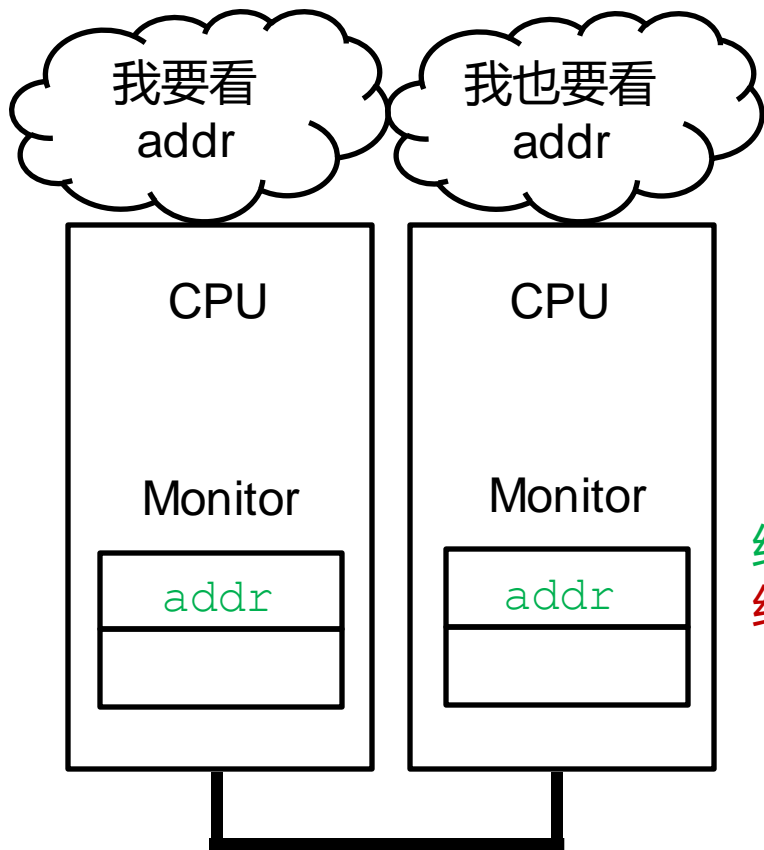
如果多核处理器呢?

基于原子指令的锁实现

- 常见的原子指令

- Test-and-set
- Compare-and-swap
- Load-linked & Store-conditional. (RISC-V(LR/SC))
 - 在一条指令中读一个值(Loadlinked)
 - 做一些操作
 - Store 时, 检查 load linked 之后, 值是否被修改过。如果没有, 则 OK, 否则, 从头再来
- Fetch-and-add

硬件原子操作：使用LL/SC实现



CPU 0/1

```
retry: ldxr    x0, addr      LL
       cmp     x0, expected
       bne     out
       stxr    x1, new_value, addr  SC
       cbnz   x1, retry
```

out:

绿色没人修改 Load-linked & Store-conditional

红色被修改 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LL/SC实现

对比成功了，
我要改

我也要
看
addr

CPU

没有人改这个地
址，我改成功了

Monitor

addr

CPU

Monitor

addr

```
retry: ldxr    x0, addr      LL
       cmp     x0, expected
       bne     out          是否成功
CPU 0  stxr    x1, new_value, addr  SC
       cbnz    x1, retry
CPU 1  out:
```

绿色没人修改

红色被修改

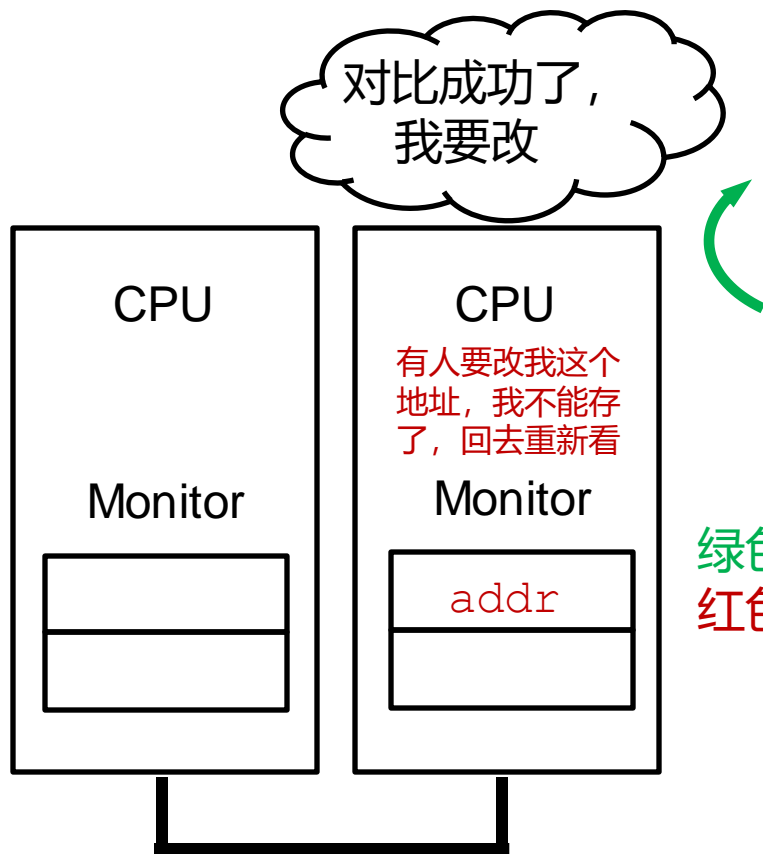
Load-linked & Store-conditional

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LL/SC实现



失败重试

```
retry: ldxr    x0, addr
      cmp     x0, expected
      bne     out
CPU 1: stxr    x1, new_value, addr
      cbnz   x1, retry
out:
```

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

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？

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

条件变量

条件变量

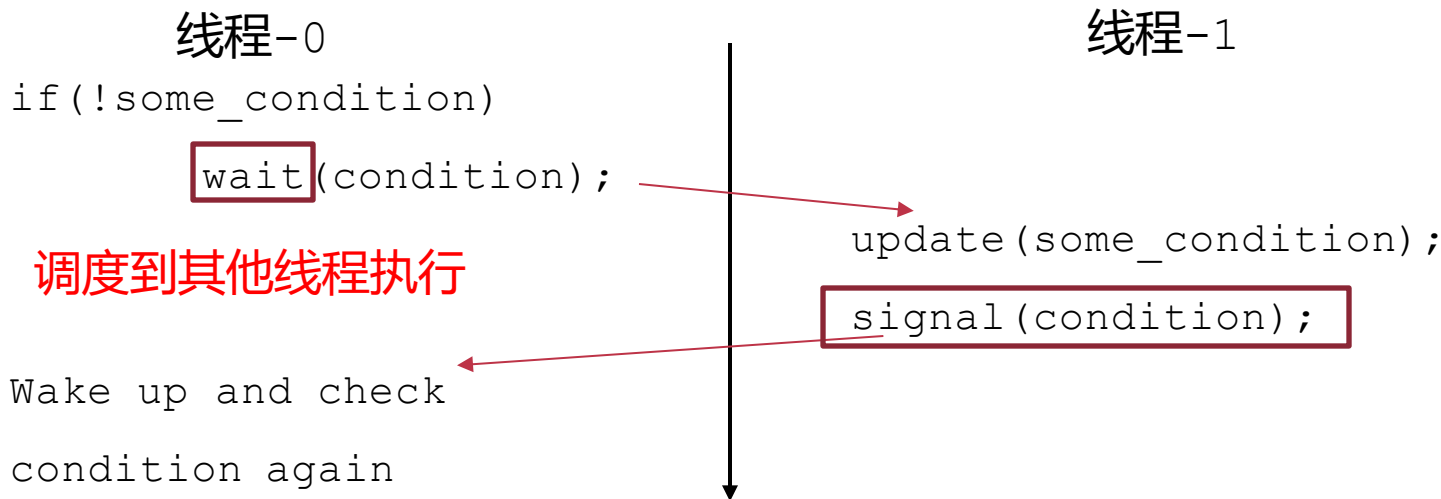
条件变量：利用睡眠/唤醒机制，避免无意义的等待

之前互斥锁的实现中：

让操作系统的调度器调度其他进程/线程执行

```
while (locked)
    /* busy waiting */;
```

条件变量：利用睡眠/唤醒机制，避免无意义的等待



条件变量的接口

提供的两个接口：

等待的接口：等待需要在临界区中

```
void cond_wait(struct cond *cond, struct lock *mutex);
```

1. 放入条件变量的**等待队列**
2. 阻塞自己同时**释放锁**：即调度器可以调度到其他线程
3. 被唤醒后重新**获取锁**

唤醒的接口：

```
void cond_signal(struct cond *cond);
```

1. 检查**等待队列**
2. 如果有等待者则**移出等待队列并唤醒**

条件变量的使用示例

等待空位代码

```
1. ...
2. /* Wait empty slot */
3. lock(empty_cnt_lock);
4. while (empty_slot == 0)
5.     cond_wait(empty_cond,
6.               empty_cnt_lock);
7. empty_slot--;
8. unlock(empty_cnt_lock);
9. ...
```

思考：为什么这里要用while?

生产空位代码

```
1. ...
2. /* Add empty slot */
3. lock(empty_cnt_lock);
4. empty_slot++;
5. cond_signal(empty_cond);
6. unlock(empty_cnt_lock);
7. ...
```

条件变量的使用示例

思考：为什么这里要用while?

线程 1

```
lock(empty_cnt_lock);  
if (empty_slot == 0)  
    cond_wait(empty_cond,  
              empty_cnt_lock);
```

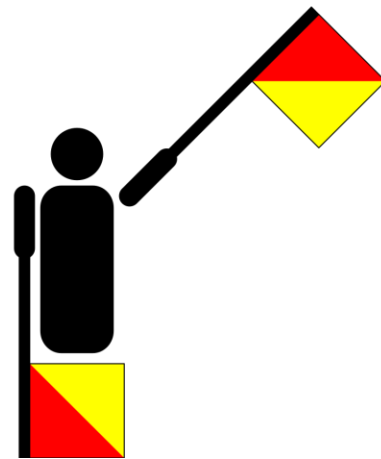
```
empty_slot--;  
unlock(empty_cnt_lock);  
empty_slot = -1
```

线程 2

有新的空位, 唤醒
empty_slot = 1

```
lock(empty_cnt_lock);  
empty_slot--;  
unlock(empty_cnt_lock); ...  
empty_slot = 0
```

重新拿到锁



信号量 (SEMAPHORE)

生产者消费者问题的另一种实现

生产者:

```
while(true) {  
    new_msg = produce_new();  
    while (empty_slot == 0)  
        ; /* No more empty slot. */  
    empty_slot--;  
    buffer_add(new_msg);  
    filled_slot++;  
}
```

消费者: while(true) {

```
    while (filled_slot == 0)  
        ; /* No new data. */  
    filled_slot--;  
    cur_msg = buffer_remove();  
    empty_slot++;  
    handle_msg(cur_msg);
```

}

思考: 为了保护计数器并
发正确, 需要在哪里加锁?
为了避免忙等, 在哪里用
条件变量?

生产者消费者问题的另一种实现

生产者：使用 **互斥锁** 搭配 **条件变量** 完成资源的等待与消耗

```
while(true) {  
    new_msg = produce_new();  
    ➡ lock(&empty_slot_lock);  
    while (empty_slot == 0)  
        ➡ cond_wait(&empty_cond, &empty_slot_lock);  
    empty_slot--;  
    ➡ unlock(&empty_slot_lock);  
  
    buffer_add(new_msg);  
    // ...  
}
```

当前实现：需要单独创建互斥锁与条件变量，并手动通过计数器来管理资源数量
为何不提出一种新的同步原语，便于在多个线程之间**管理资源**？

信号量 (PV原语)

信号量: 协调 (阻塞/放行)

多个线程共享有限数量的资源

语义上: 信号量的值`val`记录了**当前可用资源的数量**

提供了两个原语 `P` 和 `V` 用于**等待/消耗资源**

P操作: 消耗资源

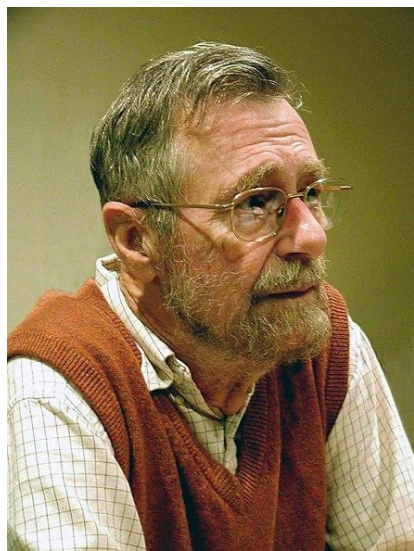
```
void sem_wait(sem_t *sem) {  
    while(sem->val <= 0)  
        /* Waiting */;  
    S--;  
}
```

`val`代表剩余资源数量

V操作: 增加资源

```
void sem_signal(sem_t *sem) {  
    sem->val++;  
}
```

注意: 此处代码只展示语义, 并非真实实现



Edsger W. Dijkstra

P操作: 荷兰语Passeren, 相当于pass

V操作: 荷兰语Verhoog, 相当于increment

信号量的使用

使用信号量可以将其压缩到一行代码

```
while(true) {  
    new_msg = produce_new();  
    lock(&empty_slot_lock);  
    while (empty_slot == 0)  
        cond_wait(&empty_cond,  
                  &empty_slot_lock);  
    empty_slot --;  
    unlock(&empty_slot_lock);  
  
    buffer_add(new_msg);  
    // ...  
}
```

```
void producer(void) {  
    new_msg = produce_new();  
    sem_wait(&empty_slot_sem);  
    buffer_add(new_msg);  
    // ...  
}
```

消耗empty_slot

信号量的使用

```
void producer(void) {  
    new_msg = produce_new();  
    sem_wait(&empty_slot_sem);  
    buffer_add(new_msg);  
    sem_signal(&filled_slot_sem);  
}
```

消耗empty_slot

增加filled_slot


```
void consumer(void) {  
    sem_wait(&filled_slot_sem);  
    cur_msg = buffer_remove();  
    sem_signal(&empty_slot_sem);  
    handle_msg(cur_msg);  
}
```

消耗filled_slot

增加empty_slot

二元信号量与计数信号量

```
void sem_init(sem_t *sem, int init_val) {  
    sem->val = init_val;  
}
```



当初初始化的资源数量为1时，为二元信号量

其计数器（counter）只有可能为0、1两个值，故被称为二元信号量

同一时刻**只有一个**线程能够拿到资源

当初初始化的资源数量大于1时，为计数信号量

同一时刻**可能有多个**线程能够拿到资源

信号量的数据结构

- 整型信号量
- 记录型信号量
- AND型信号量
- 信号量集

AND信号量

- **AND信号量集：同时需要多个资源且每种占用一个资源时的信号量操作**

- 将进程运行过程中所需要的所有资源，一次性全部分配给进程
- 待进程使用完成以后再一起释放。
- 只要有一个资源尚未分配给进程，则其他可能分配的资源也不能分配给它。

```
Wait(S1, S2, ..., Sn)
  if(S1>=1 and ... and Sn>=1 then
    for i: =1 to n do
      Si: = Si -1
    endfor
  else
    将进程放入阻塞队列
  endif
```

```
Signal(S1,S2,...,Sn)
  for i: =1 to n do
    Si = Si +1;
    唤醒所有因Si不能满足
    而进入阻塞队列的进程
  endfor;
```

信号量集

- **基本思想:**

- 在AND型信号量集基础上进一步扩展, 进程对信号量 S_i 的测试值为 T_i , 占用值为 d_i 。
- `Swait($S_1, t_1, d_1; \dots; S_n, t_n, d_n$);`
- `Ssignal($S_1, d_1; \dots; S_n, d_n$);`

- **一般信号量集的几种特定情况**

- `Swait(S, d, d)`表示每次申请 d 个资源, 当少于 d 个时, 不分配。
- `Swait($S, 1, 1$)`表示互斥信号量
- `Swait($S, 1, 0$)`可作为一个可控开关(当 $S \geq 1$ 时, 允许多个进程进入临界区; 当 $S=0$ 时禁止任何进程进入临界区)。
- 信号量集未必成对使用`Swait()`和`Ssignal()`, 如一起申请资源, 但可以不一一起释放资源。

信号量

- **信号量的使用**

- 适用于被占用较长时间的锁
- 短时间的加锁场景不适合采用信号量
 - 维护等待队列、切换上下文会有开销
- 信号量只能在进程上下文中使用，不可用于中断上下文，why?
- 在获取信号量时，不能拥有自旋锁
 - 获取信号量不成功时，会出现“带锁睡眠”

- **对于互斥**

- 保证同时只有一个进程可以访问共享数据，使用哪种信号量？

- **对于有条件同步**

- 允许进程等待特定条件发生，使用哪种？

信号量vs锁

- 相较于锁

- 信号量有更多的语义
- 当信号量大于1，可以允许多个进程同时访问临界资源
- 当信号量等于1，可以用来做互斥访问

读写锁

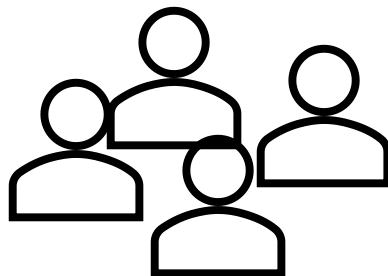
公告栏问题



写者



公告栏



读者



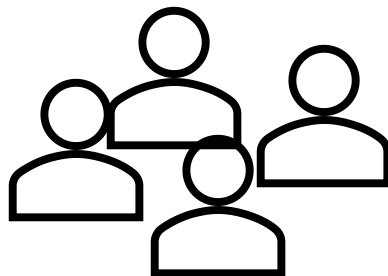
思考：多个读者如果希望读公告栏，他们互斥吗？

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);
}

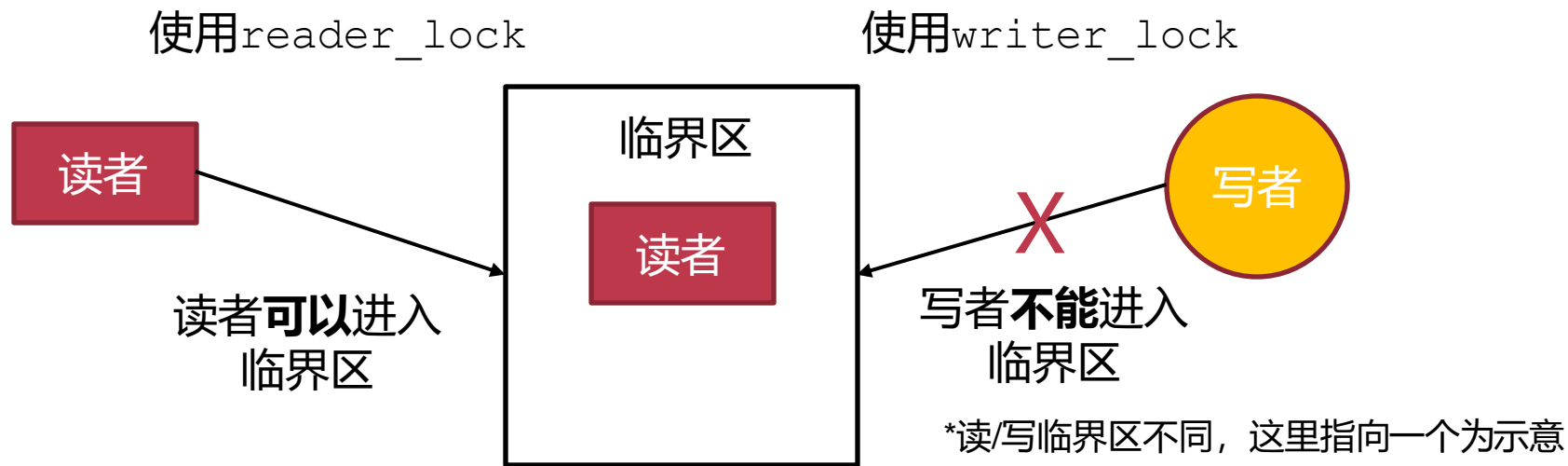
void writer(void) {
    lock_writer(lock);
    update_data(data);
    unlock_writer(lock);
}
```

读写锁

互斥锁：所有的线程均互斥，同一时刻**只能有一个线程**进入临界区

对于部分只读取共享数据的线程过于严厉

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

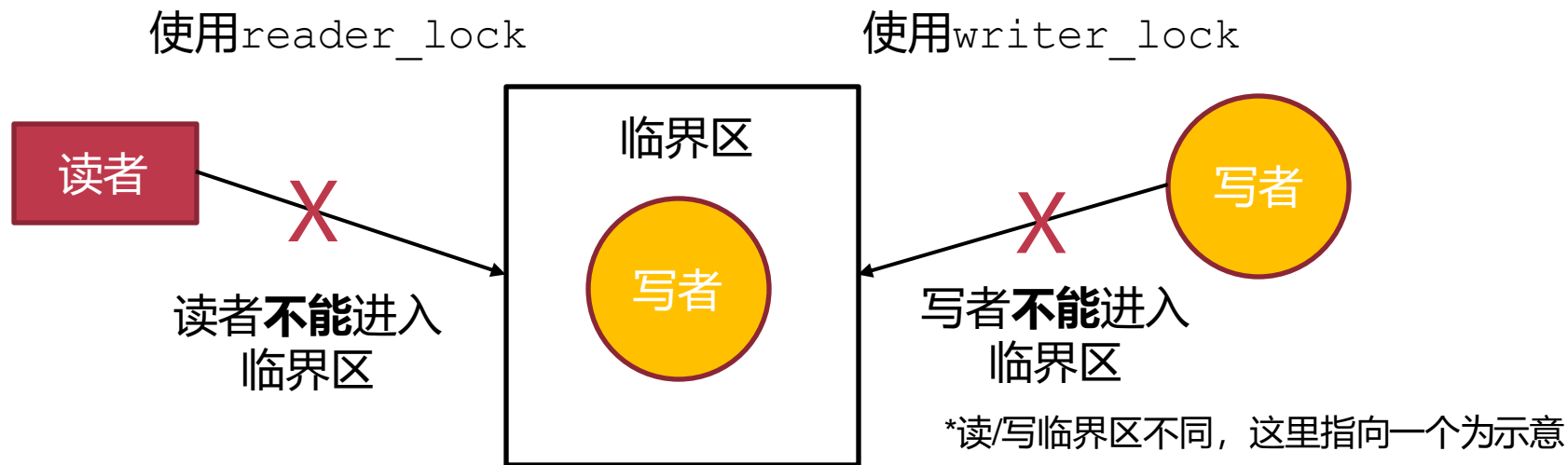


读写锁

互斥锁：所有的线程均互斥，同一时刻只能有一个线程进入临界区

对于部分只读取共享数据的线程过于严厉

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：**

- t0：有读者在临界区
- t1：有新的写者在等待
- t2：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

Reader计数器：
表示有多少读者

```
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);
}
```

第一个/最后一个reader负责获取/释放写锁

只有当完全没有读者时
写者才能进入临界区

读写锁的实现：偏向读者为例






读者锁



写者锁



读者计数器

1. 获取读者锁，更新读者计数器 
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



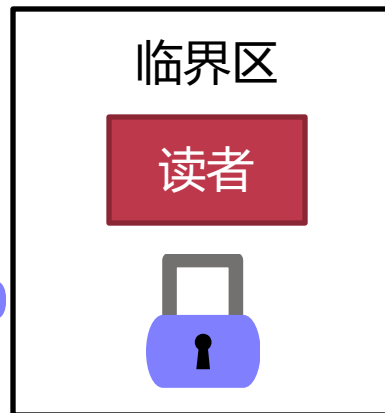
写者锁



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



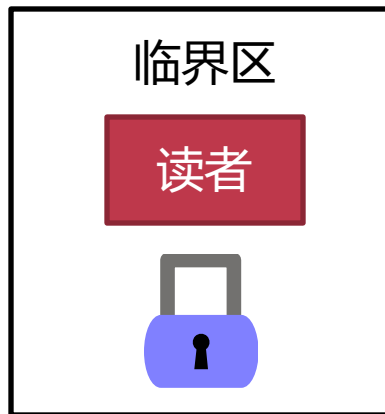
读者计数器





写者



读者



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 

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



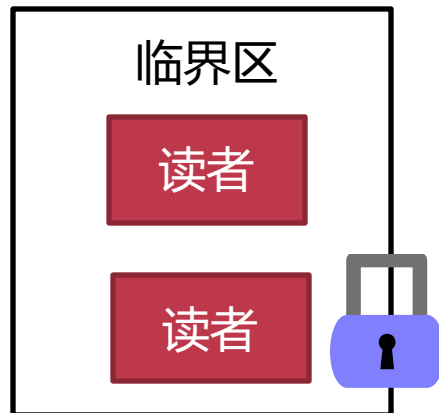
读者锁



写者锁



读者计数器



在读临界区中的读者数量

*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

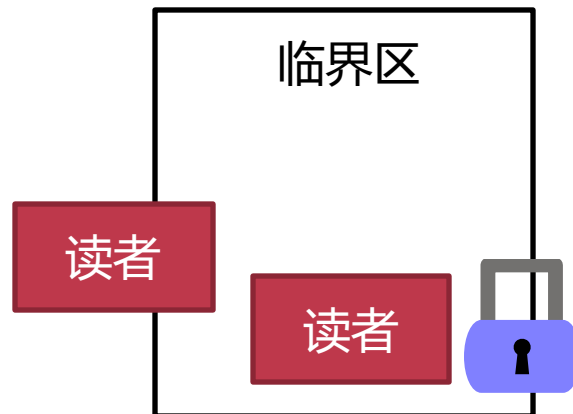




写者锁



读者计数器

在读临界区中的读者数量

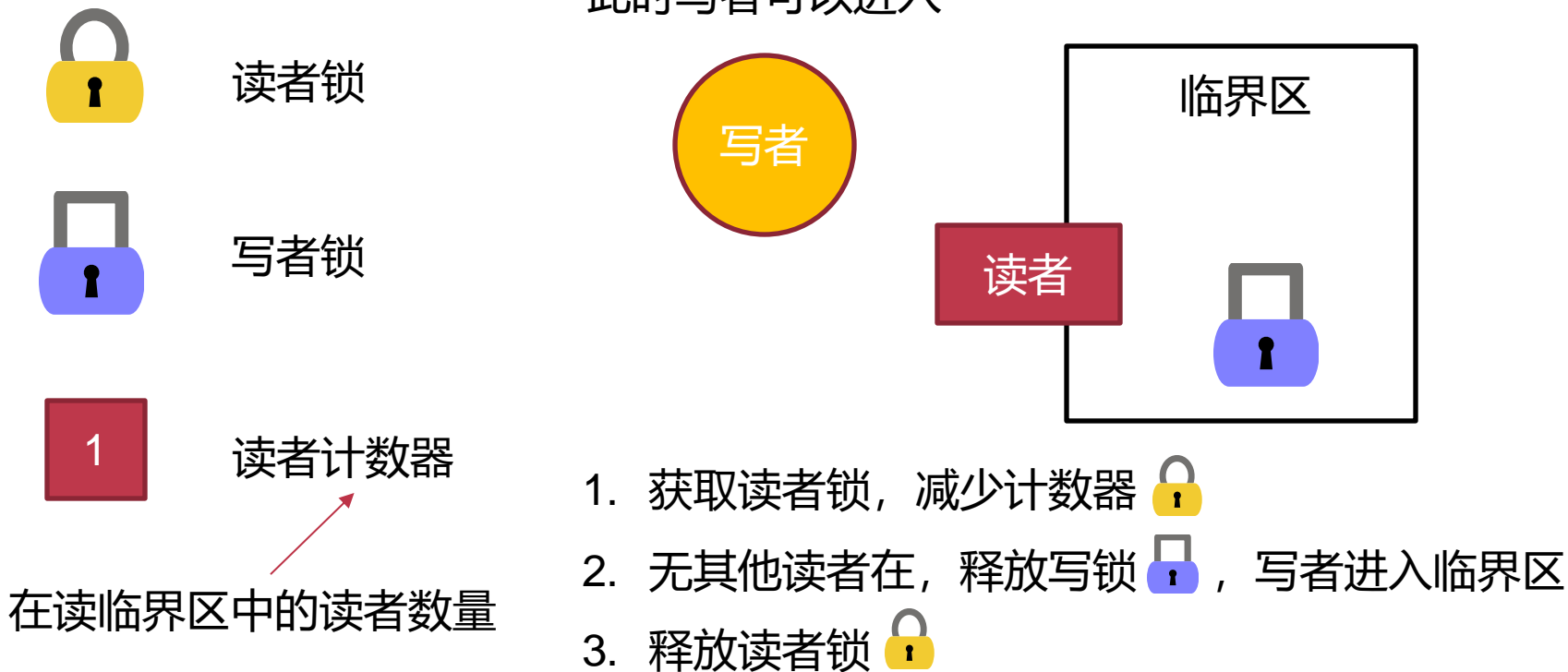


1. 获取读者锁，减少计数器 
2. 还有其他读者在，无需释放写锁
3. 释放读者锁 

*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

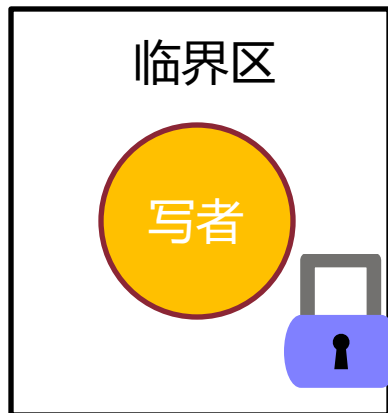


读者计数器

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



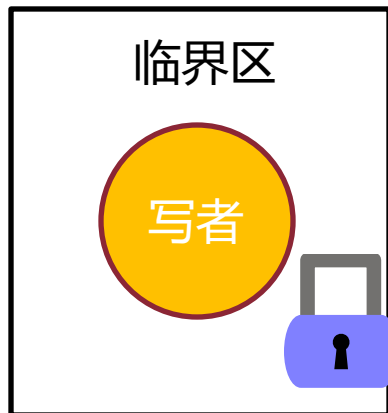
读者锁




写者锁



读者计数器



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

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

Linux中的读写自旋锁

Method	Description
<code>read_lock()</code>	Acquires given lock for reading
<code>read_lock_irq()</code>	Disables local interrupts and acquires given lock for reading
<code>read_lock_irqsave()</code>	Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading
<code>read_unlock()</code>	Releases given lock for reading
<code>read_unlock_irq()</code>	Releases given lock and enables local interrupts
<code>read_unlock_irqrestore()</code>	Releases given lock and restores local interrupts to the given previous state
<code>write_lock()</code>	Acquires given lock for writing
<code>write_lock_irq()</code>	Disables local interrupts and acquires the given lock for writing
<code>write_lock_irqsave()</code>	Saves current state of local interrupts, disables local interrupts, and acquires the given lock for writing
<code>write_unlock()</code>	Releases given lock
<code>write_unlock_irq()</code>	Releases given lock and enables local interrupts

读写锁的实现

- 读写锁的设计实现

- 由32位数据表示，含两部分编码
 - 第0-23位：读者数量
 - 第24位：可写标记，为1表示为可写
- 初始值：0X01000000

```
int _raw_read_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    atomic_dec(count);
    if (atomic_read(count) >= 0) ← 减1后不小于0:
        return 1;                    表示加读锁成功
    atomic_inc(count); ← 减1后小于0:
    return 0;                    表示加读锁失败，加1恢复原值后返回
}
```


写锁

```
int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    if (atomic_sub_and_test(0x01000000, count))
        return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

减0x01000000后不为零
表示锁被读锁占用或者被写锁占用
恢复原值后返回

减0x01000000后为零
表示写锁成功，返回1

seqlock

- **一种“写操作”优先的锁**
 - 解决普通read_lock读写优先级相同的问题
- **实现机制**
 - 锁带序列号，每次写时加1
 - 读数据前后判断序列号是否变化，有变化则重试
 - 并发写会使读操作循环重复，直到写锁已释放
- **使用范围**
 - 大量读、少量写
 - 优先写操作，避免读操作阻塞写操作
 - 典型场景：jiffies变量的更新

Seqlock使用示例

- 使用与开销

- 读者在读取数据前后，需要两次加锁获取锁
- 对前后获取锁的seq进行比较，值相等判断加锁成功

```
do {  
    seq = read_seqbegin(&mr_seq_lock);  
    /* read data here ... */  
} while (read_seqretry(&mr_seq_lock, seq));
```

► RCU: 更高效的读写互斥

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？

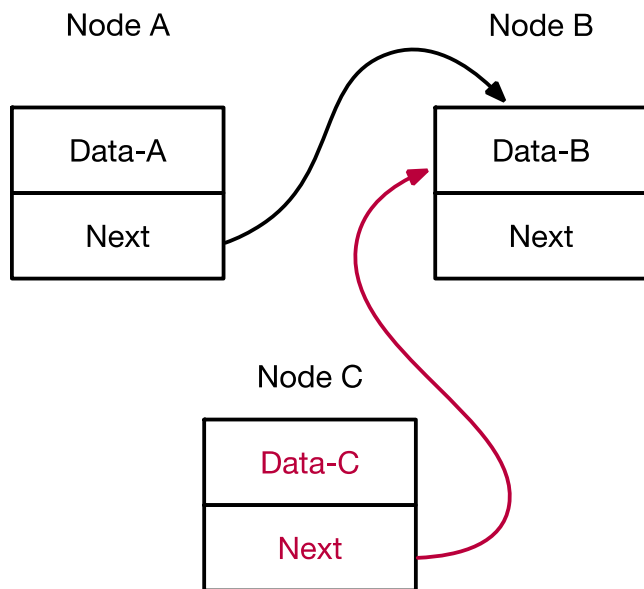
需求1：需要一种能够**类似之前硬件原子操作**的方式，让读者要么看到旧的值，要么看到新的值，不会读到任何中间结果。

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

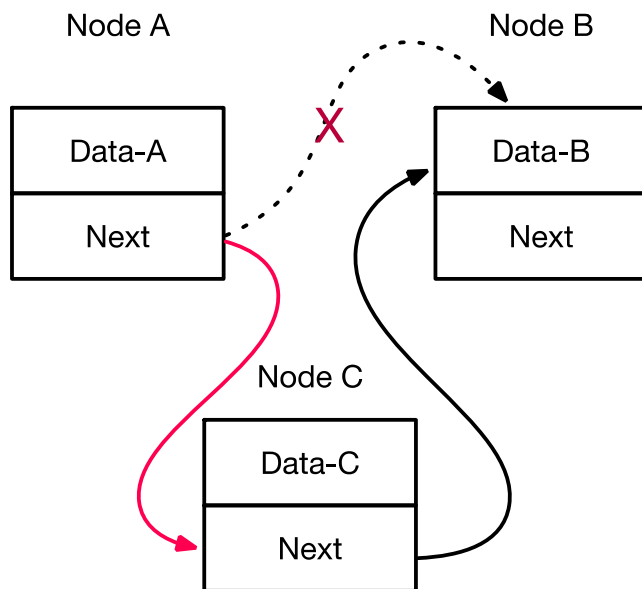
RCU 订阅/发布机制

以链表为例：插入结点Node C



① 填入Data与Pointer

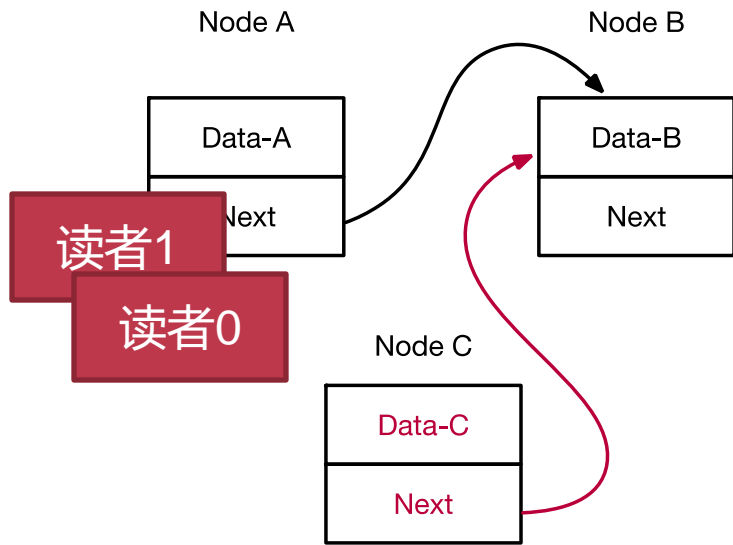
读者看不到Node C



② 利用单拷贝原子性，原子地更新Node A的指针

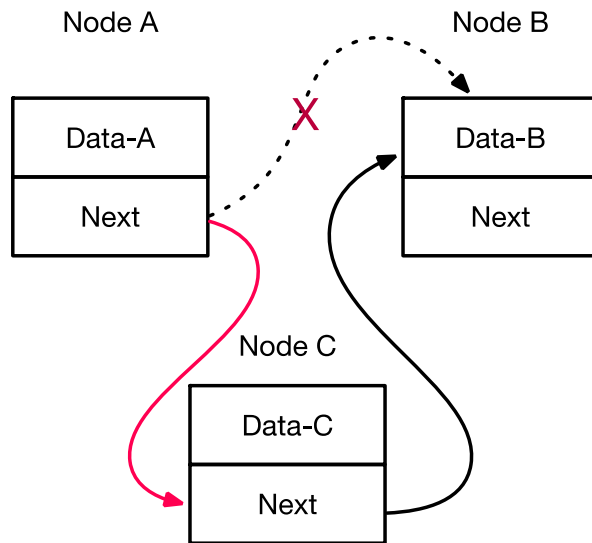
读者看到Node C₁₀₃

RCU 订阅/发布机制：此时的读者



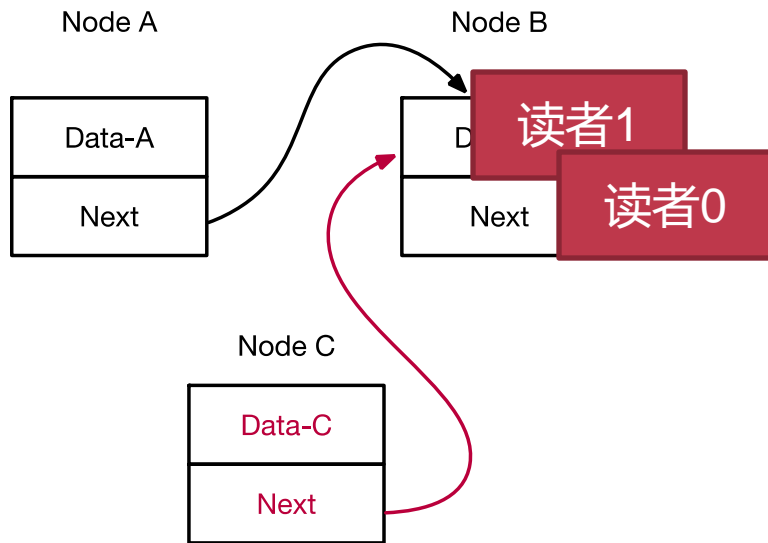
Writing Node C

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

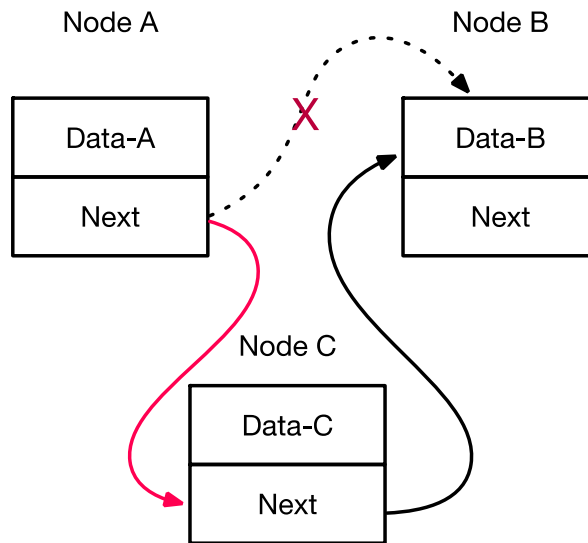
RCU 订阅/发布机制：此时的读者



Writing Node C

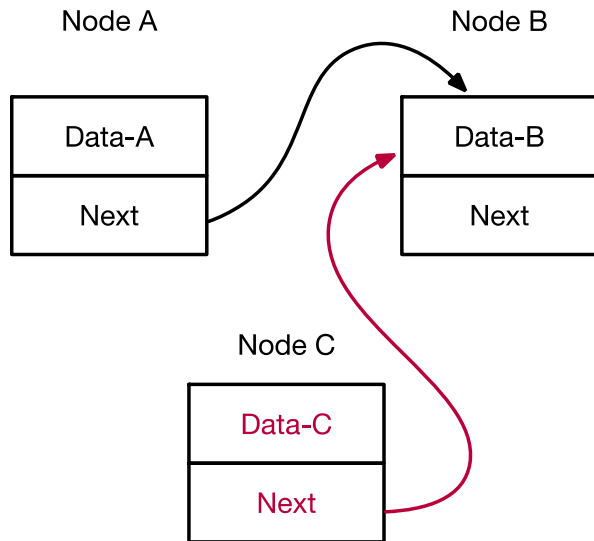
① 填入Data与Pointer

看不到Node C

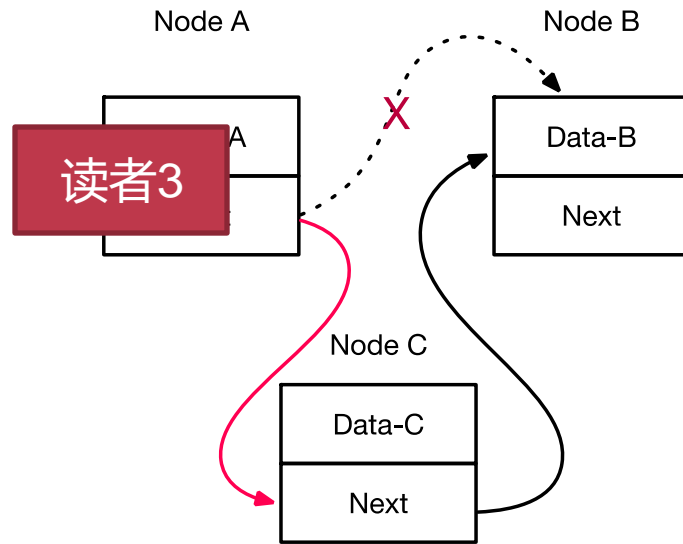


② 利用单拷贝原子性，原子地更新Node A的指针

RCU 订阅/发布机制：此时的读者

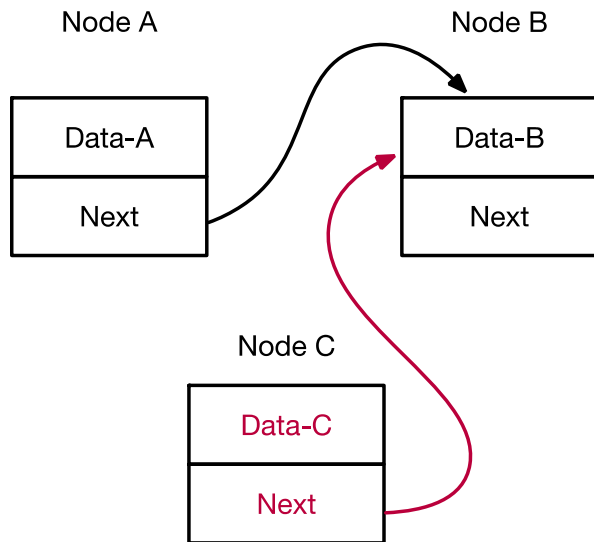


① 填入Data与Pointer

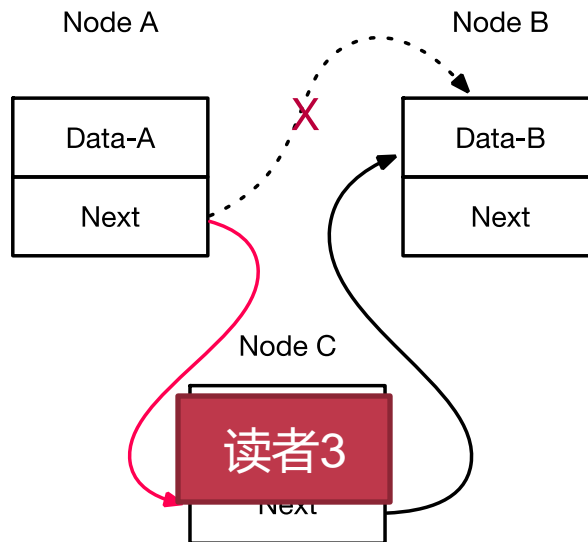


② 利用单拷贝原子性，原子地更新Node A的指针

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

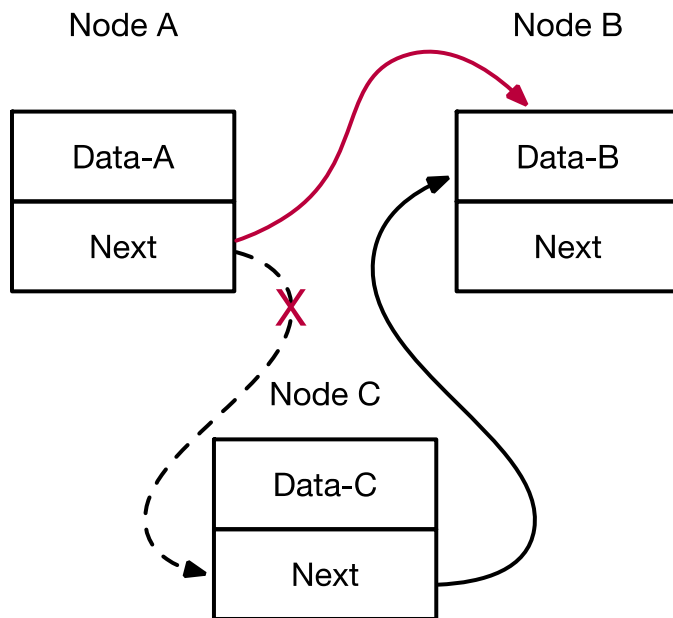
可以看到Node C

RCU 订阅/发布机制

删除结点Node C

`A.Next = &B`

思考：局限性在哪？



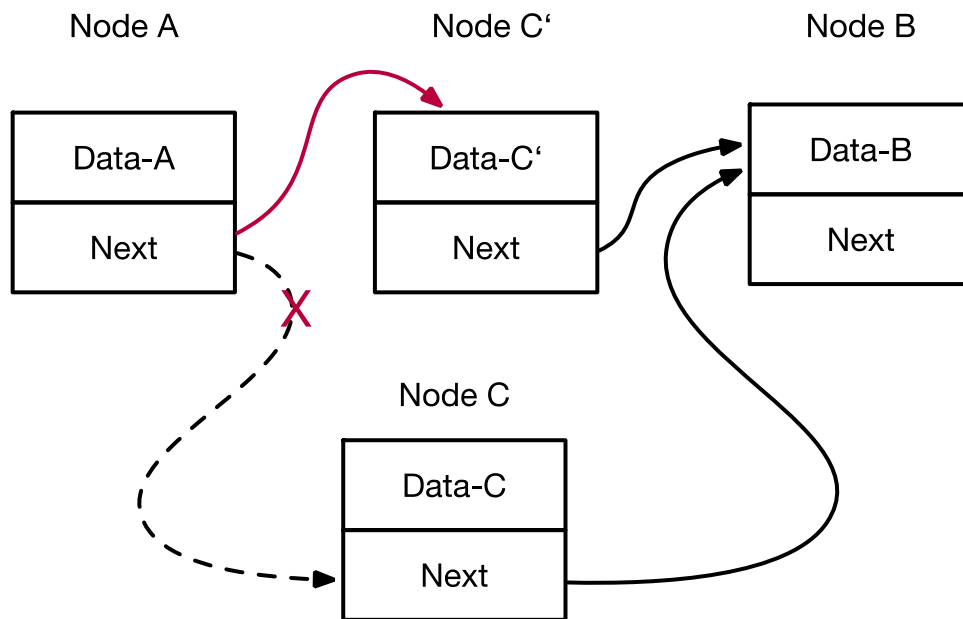
RCU 订阅/发布机制

更新结点Node C

$A.Next = \&C'$

复制一个新的Node C'

更新Next指针

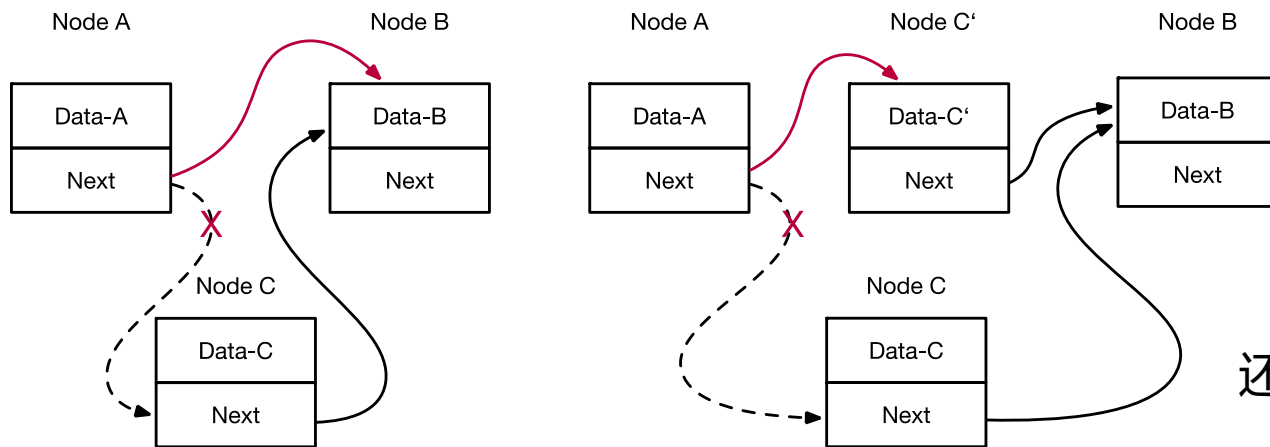


Read Copy Update, RCU

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

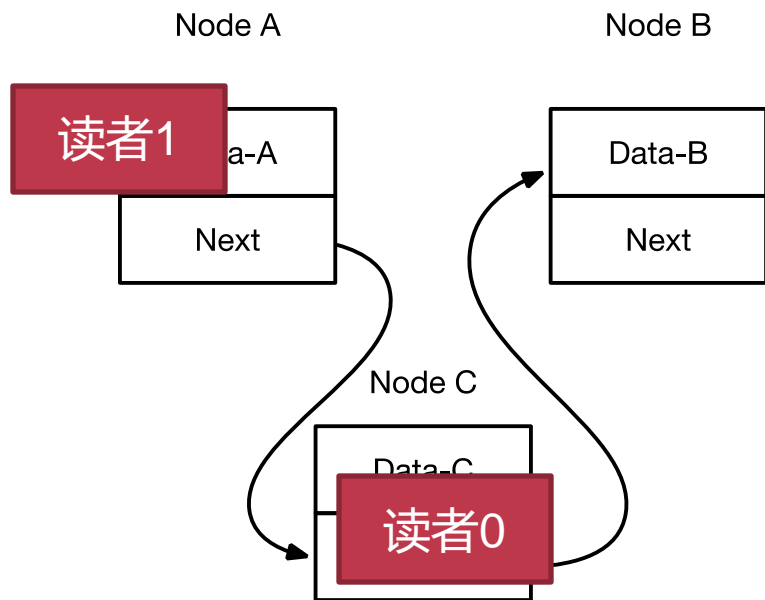
需求2：在**合适的时间**，**回收**无用的旧拷贝



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



T0

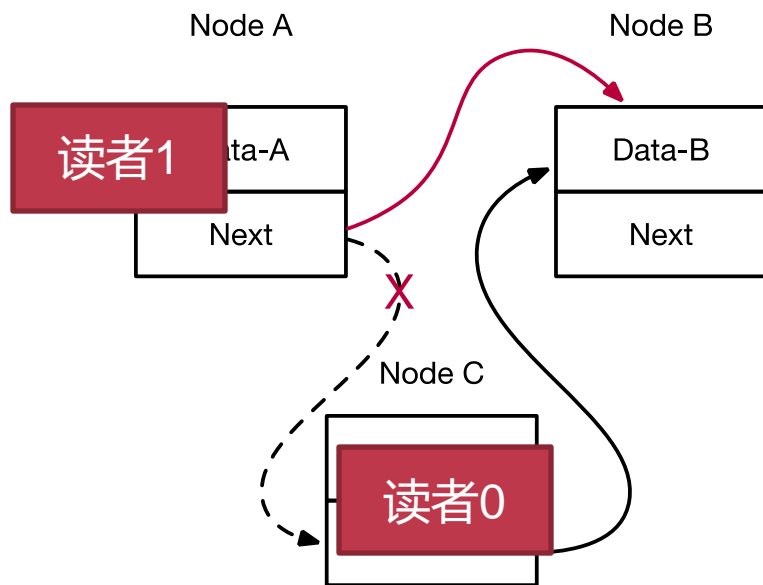
读者0

读者1

读者0

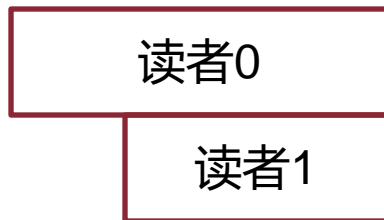
RCU 宽限期

知道读临界区什么时候开始，什么时候结束



$A.Next = \&B$

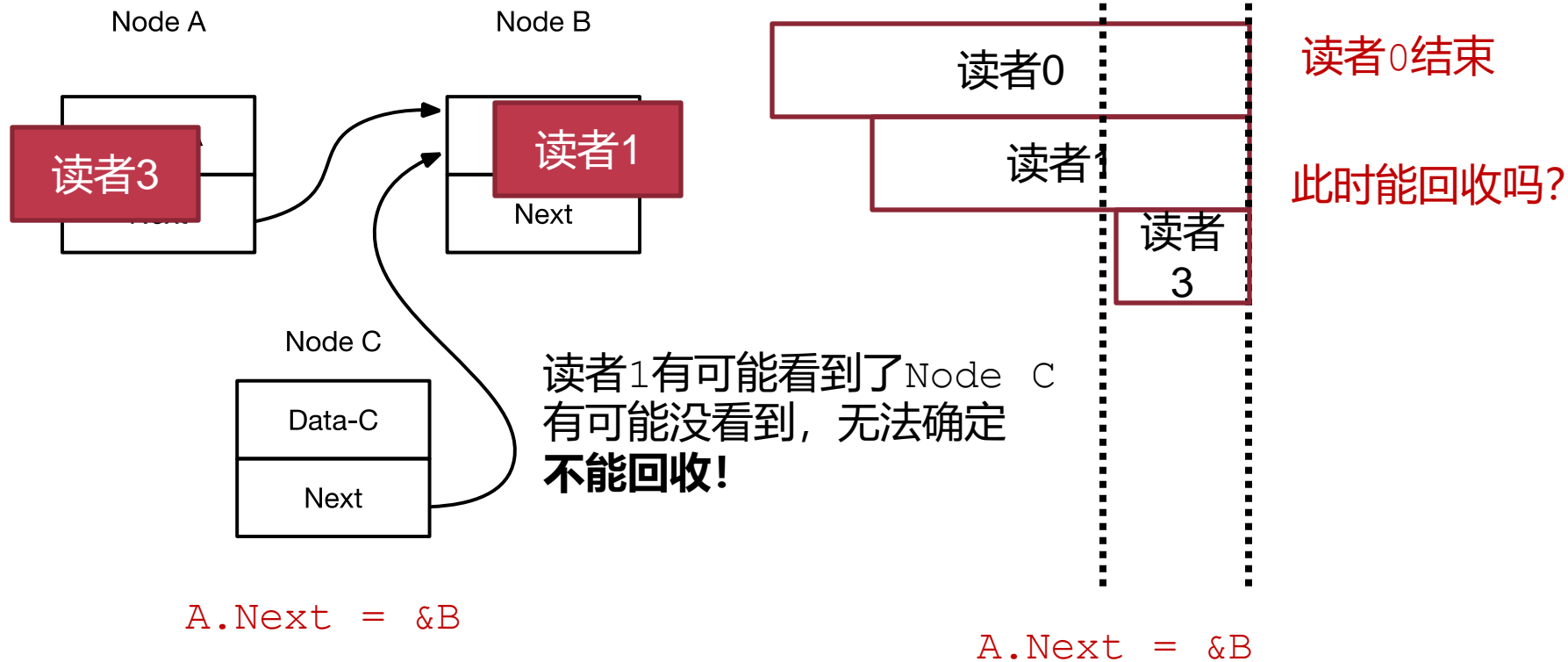
T0 T1



$A.Next = \&B$

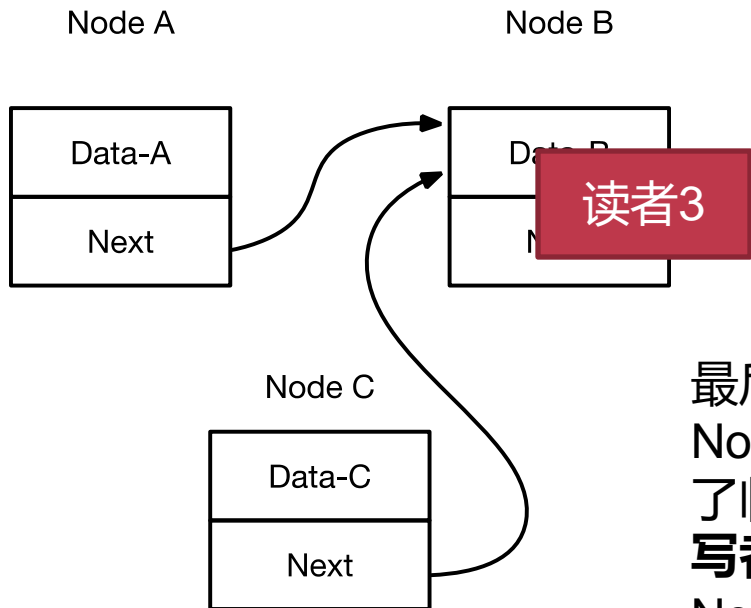
RCU 宽限期

知道读临界区什么时候开始，什么时候结束



RCU 宽限期

知道读临界区什么时候开始，什么时候结束

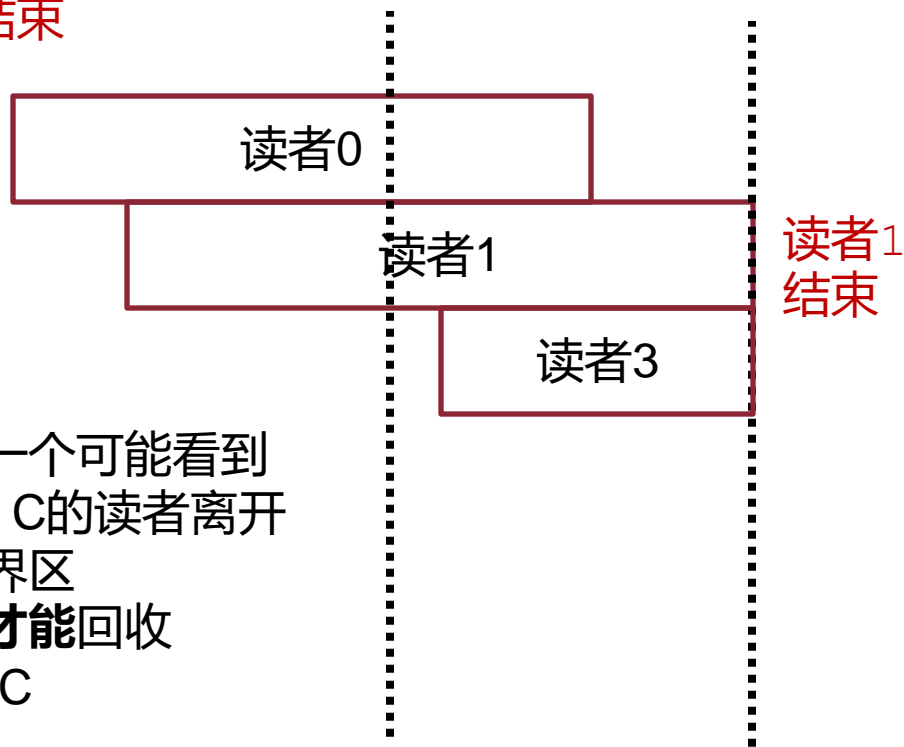


$A.Next = \&B$

最后一个可能看到
Node C的读者离开
了临界区
写者才能回收
NodeC

写者开销较大

T0 T1 T2 T3



$A.Next = \&B$ 可以回收Node C

RCU 宽限期

如何知道读临界区什么时候开始，什么时候结束？

```
void rcu_reader() {  
    RCU_READ_START();  
  
    /* Reader Critical Section */  
  
    RCU_READ_STOP();  
}
```

通知RCU，读者进临界区了

通知RCU，读者出临界区了

可以使用不同的方式实现：如计数器

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

不同同步原语之间的比较

同步原语对比：互斥锁/条件变量/信号量

只允许0与1的信号量：只有一个资源，即互斥锁



- **互斥锁与二元信号量**功能类似，但**抽象不同**：
 - 互斥锁有**拥有者**的概念，一般同一个线程拿锁/放锁
 - 信号量为资源协调，一般一个线程signal，另一个线程wait

```
sem_init(&s, 1);
```

```
sem_wait
```



```
lock
```

```
sem_signal
```



```
unlock
```

通常可直接替换

同步原语对比：互斥锁/条件变量/信号量

只允许0与1的信号量：只有一个资源，即互斥锁



- **互斥锁与二元信号量**功能类似，但**抽象不同**：
 - 互斥锁有**拥有者**的概念，一般同一个线程拿锁/放锁
 - 信号量为资源协调，一般一个线程signal，另一个线程wait

Thread 0

```
lock(&lock0);
```

Thread 0

```
sem_wait(&s0);
```

Thread 1

另一个线程

同一线程

```
unlock(&lock0);
```

```
sem_signal(&s0);
```


同步原语对比：互斥锁/条件变量/信号量

只允许0与1的信号量：只有一个资源，即互斥锁



- **互斥锁与二元信号量**功能类似，但**抽象不同**：
 - 互斥锁有**拥有者**的概念，一般同一个线程拿锁/放锁
 - 信号量为资源协调，一般一个线程signal，另一个线程wait
- **条件变量**用于解决不同问题（睡眠/唤醒），需要搭配**互斥锁**使用

```
lock(&empty_slot_lock);  
while (empty_slot == 0)  
    cond_wait(&empty_cond,  
             &empty_slot_lock);  
empty_slot--;  
unlock(&empty_slot_lock);
```

搭配**互斥锁+计数器**
可以实现与**信号量**
相同的功能

`sem_wait(&empty_slot_sem);`

同步原语对比：互斥锁 vs 读写锁

- 接口不同：读写锁区分读者与写者
- **针对场景不同**：获取**更多程序语义**，标明只读代码段，达到更好性能
- 读写锁在读多写少场景中可以显著**提升读者并行度**
 - 即允许多个读者同时执行读临界区
- 只用写者锁，则与互斥锁的语义基本相同

同步原语对比：互斥锁 vs 读写锁

Reader 0	Reader 1
<code>lock(&glock);</code>	<code>lock(&glock);</code>
<code>// Reader CS</code>	被阻塞
<code>unlock(&glock);</code>	
	<code>lock(&glock);</code>
	<code>// Reader CS</code>
	<code>unlock(&glock);</code>

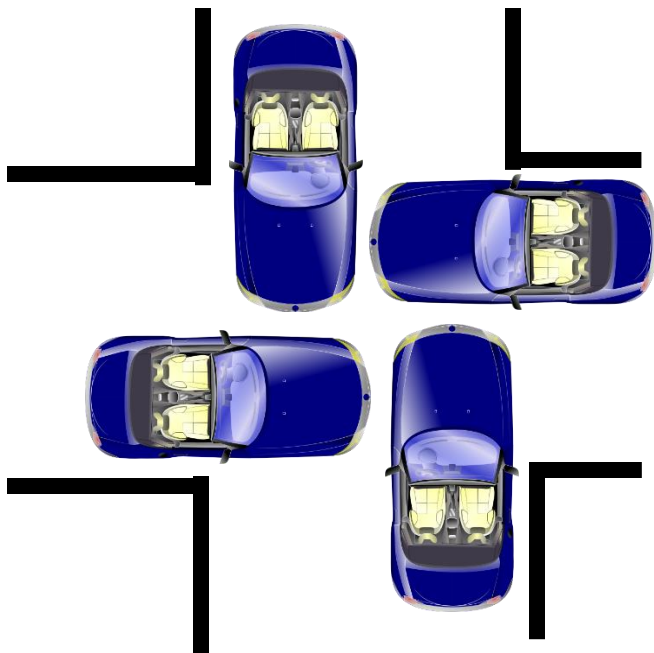
Reader 0	Reader 1
<code>reader_lock(&glock);</code>	<code>reader_lock(&glock);</code>
<code>// Reader CS</code>	<code>// Reader CS</code>
<code>reader_unlock(&glock);</code>	<code>reader_unlock(&glock);</code>

同时执行



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

同一时刻只有一个线程能够访问

```
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

一直持有一部分资源并等待另一部分
不会中途释放（如proc_A不会放锁A）

```
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

即proc_B不会抢proc_A已经持有的锁A

```
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

A等B, B等A

```
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?

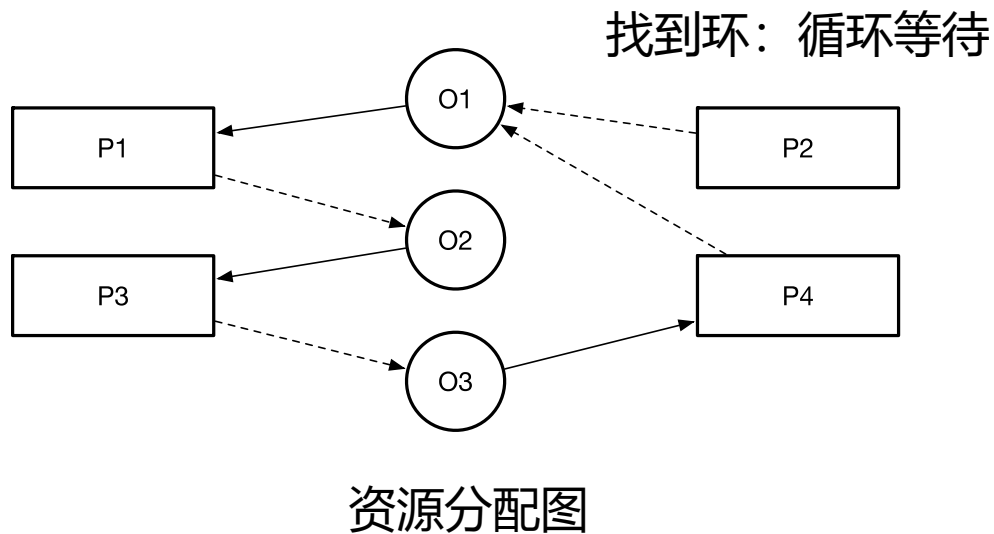
解决死锁

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所有循环中的线程
- 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

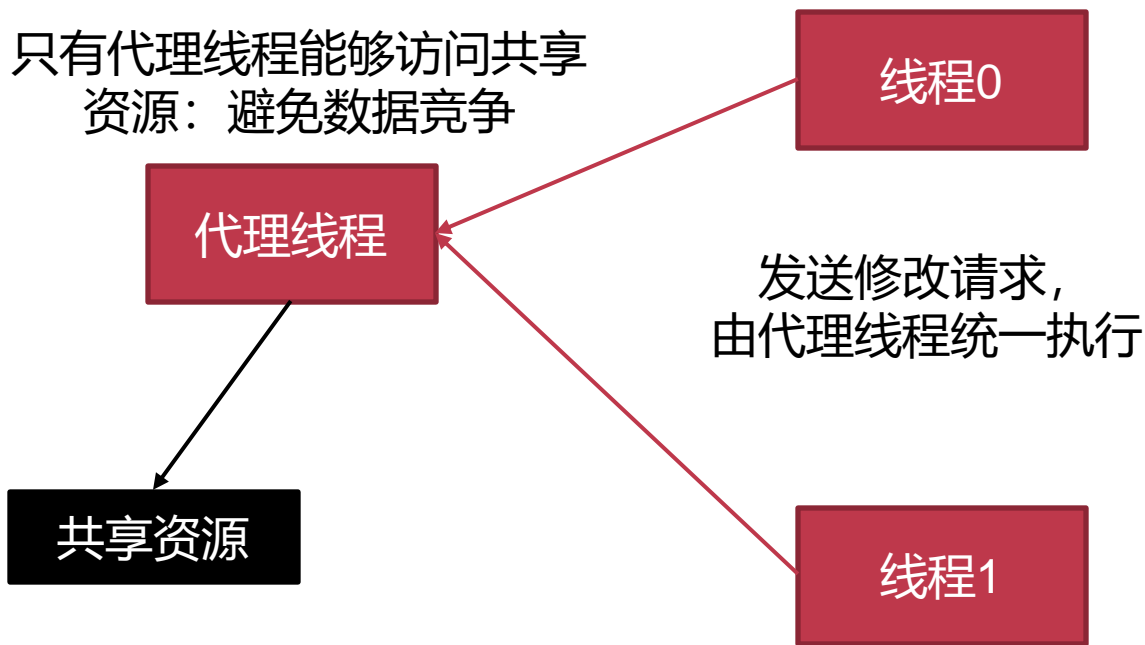
设计时避免：死锁预防

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）

只有代理线程能够访问共享
资源：避免数据竞争



*代理锁 (Delegation Lock) 实现了该功能

死锁预防：四个方向

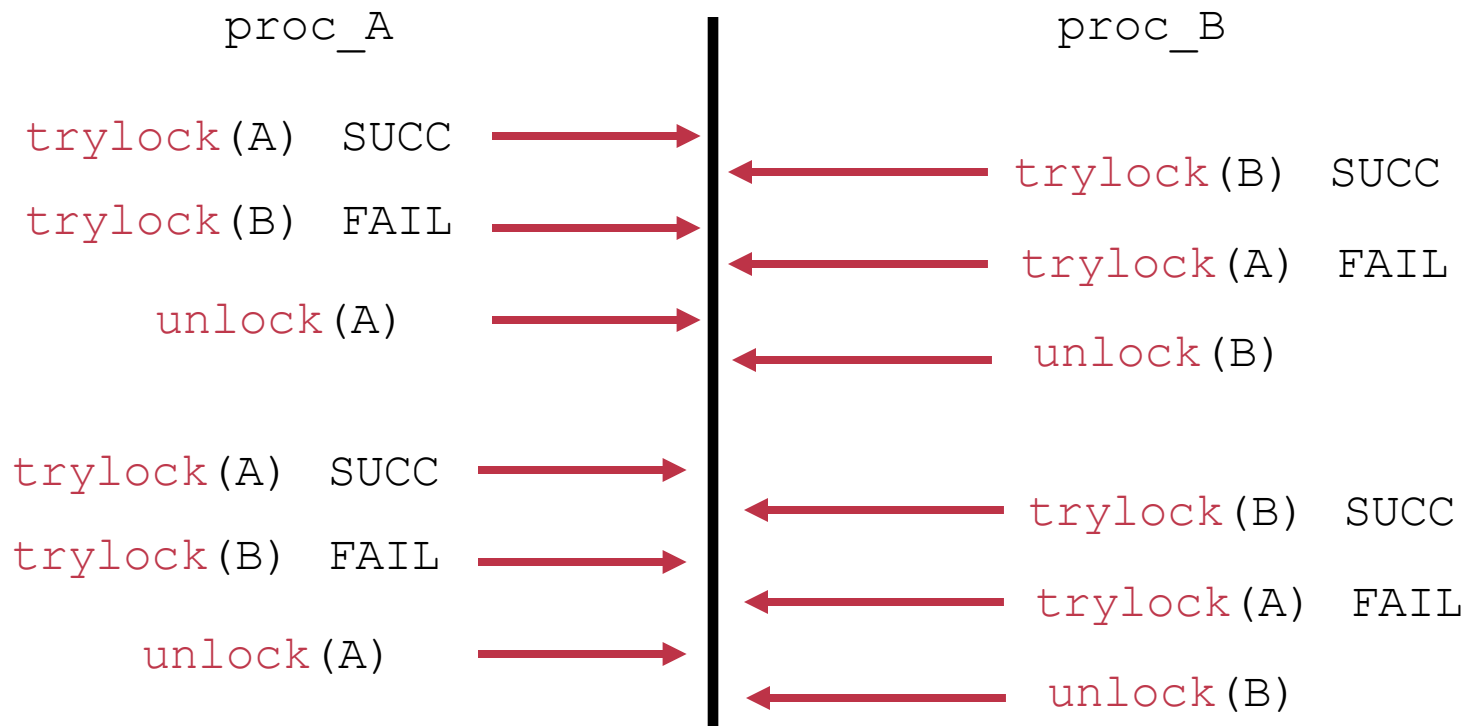
- 1、避免互斥访问：通过其他手段（如代理执行）
- 2、不允许持有并等待：一次性申请所有资源

```
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

需要让线程A正确回滚到拿锁A之前的状态

```
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);  
}
```

↑ 抢占锁A

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
 - 对所有资源进行编号
 - 让所有线程递增获取

```
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);  
}
```

A: 1号 B: 2号: 必须先拿锁A, 再拿锁B

任意时刻：获取最大资源号的线程可以继续执行，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：银行家算法

死锁避免：运行时检查是否会出现死锁

银行家算法的核心：

- 所有线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：阻塞线程，下次再给
 - 如果不会造成：给线程该资源

死锁避免：银行家算法

如何**预演判断**？将系统划分为两个状态

对于一组线程 $\{P1, P2, \dots, Pn\}$:

- 安全状态

能找出至少一个执行序列，如 $P2 \rightarrow P1 \rightarrow P5 \dots$ 让所有线程需求得到满足

- 非安全状态

不能找出这个序列，必定会导致死锁

安全性检查算法



银行家算法：保证系统一直处于**安全状态**，且按照这个序列执行

银行家算法：安全性检查

四个数据结构：

M个资源 N个线程

- 全局可利用资源：Available[M]
- 每线程最大需求量：Max[N][M]
- 已分配资源：Allocation[N][M]
- 还需要的资源：Need[N][M]

银行家算法安全性检查：一个例子

安全序列：

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

某时刻系统状态

分配给能满足其全部需求的线程

银行家算法安全性检查：一个例子

安全序列： P2 ->

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1	5	10	2	8	3	2	3	2
P2								
P3	10	11	5	1	5	10		

模拟P2执行完成

分配给能满足其全部需求的线程

银行家算法安全性检查：一个例子

安全序列： P2 -> P1 ->

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1							5	10
P2								
P3	10	11	5	1	5	10		

模拟P1执行完成

分配给能满足其全部需求的线程

银行家算法安全性检查：一个例子

安全序列： P2 -> P1 -> P3

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1							10	11
P2								
P3								

模拟P3执行完成

分配给能满足其全部需求的线程

银行家算法安全性检查：一个例子

安全序列： P2 -> P1 -> P3

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

通过安全性检查：处于安全状态！

新来请求：P1请求资源，需要A资源2份，B资源1份

银行家算法安全性检查：一个例子

安全序列： P2 -> P1 -> P3

	Max		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
P1	5	10	4	9	1	1	3→1	1→0
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

新来请求：P1请求资源，需要A资源2份，B资源1份

假设分配给它，运行安全检查：无法通过

采取行动：阻塞P1，保证系统维持在安全状态