# 同步原语:
# 条件变量与读写锁

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

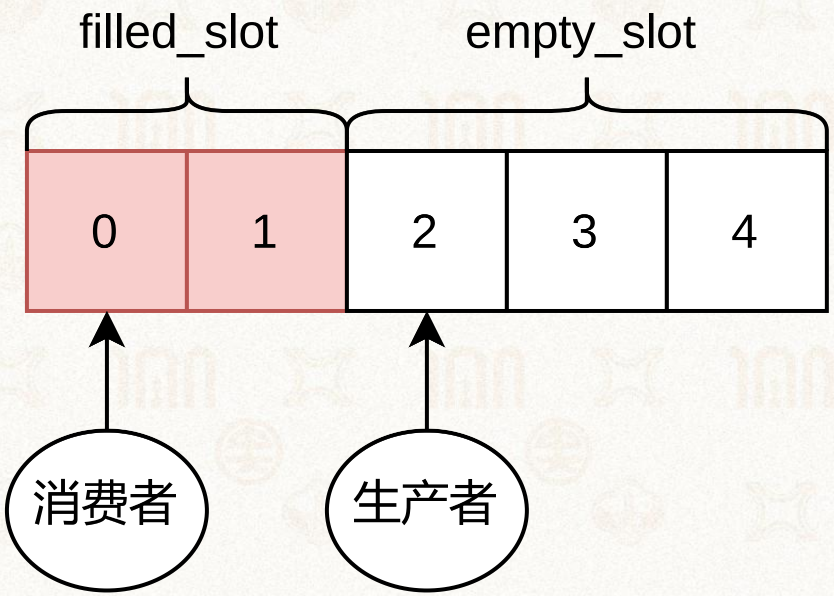
- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498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

# 大纲

- 同步问题的背景
  - 多核场景
  - 生产者消费者模型
  - 临界区问题
- 互斥锁
  - 皮特森算法
  - 原子操作
  - 互斥锁抽象
    - 自旋锁
    - 排号自旋锁
- 条件变量
- 信号量
  - PV原语
- 读写锁
- 同步原语产生的问题
  - 死锁
    - 银行家算法
  - 活锁
  - 优先级反转

# 生产者消费者问题：单生产者、单消费者

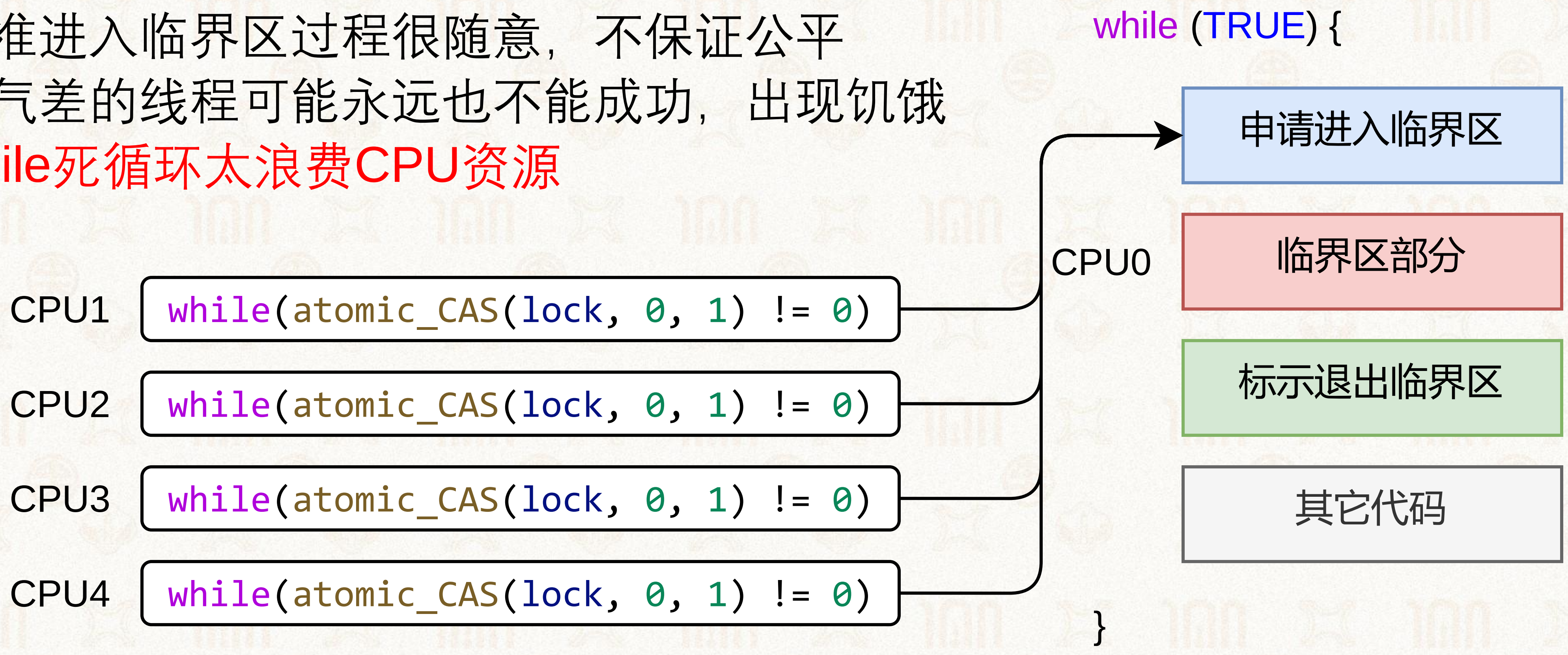
```c
volatile int empty_slot = 5; // 共享的
volatile int filled_slot = 0;// 共享的
void producer(void) {
    int new_msg;
    while (TRUE) {
        new_msg = produce_new();
        while (empty_slot == 0)
            ; // 没有空位可使用
        empty_slot--;
        buffer_add(new_msg);
        filled_slot++;
    }
}
void consumer(void) {
    int cur_msg;
    while(TRUE) {
        while(filled_slot == 0)
            ; // 没有对象可消耗
        filled_slot--;
        cur_msg = buffer_remove();
        empty_slot++;
        consume_msg(cur_msg);
    }
}
```




无谓消耗CPU资源!

# 用原子操作实现互斥锁：自旋锁(spin lock)

- 可以保证互斥访问与空闲让进
- 优点：效率高，响应快
- 缺点：不能保证有限等待
  - 批准进入临界区过程很随意，不保证公平
  - 运气差的线程可能永远也不能成功，出现饥饿
  - while死循环太浪费CPU资源

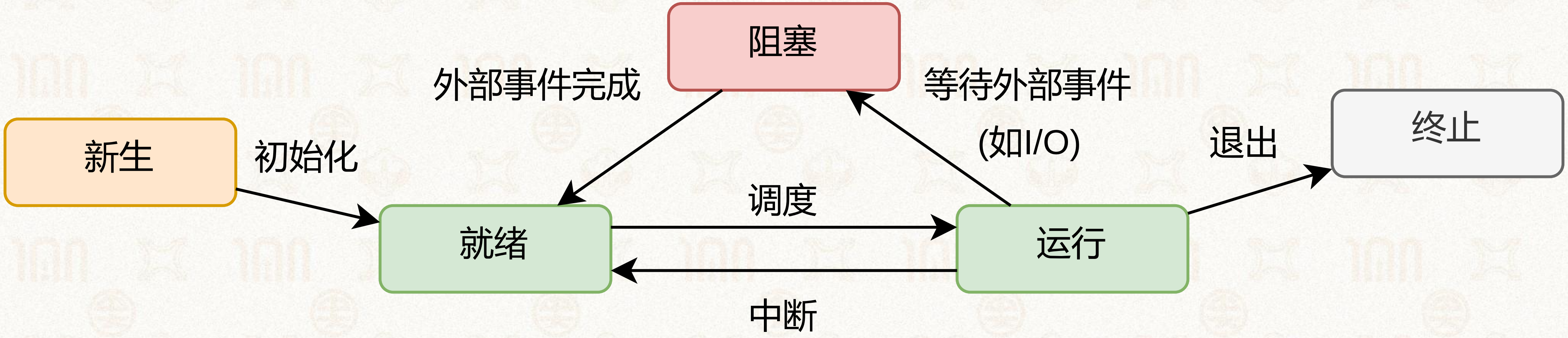


# 生产者消费者模型：状态切换



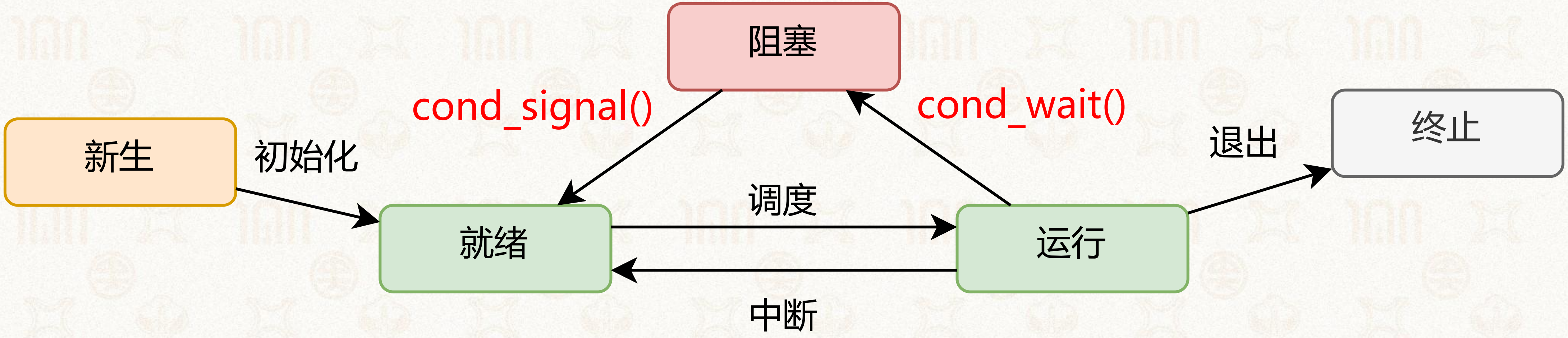
```c
void producer(void) {
    int new_msg;
    while (TRUE) {
        new_msg = produce_new();
        while (empty_slot == 0)
            ; // 没有空位可使用
        empty_slot--;
        buffer_add(new_msg);
        filled_slot++;
    }
}
```

- 用一种特殊机制将线程由运行态转化为阻塞态
  - 需要由操作系统配合
  - 怎么从阻塞态唤醒到就绪态
- 节约CPU资源，留给有需要的线程



# 条件变量

- 两个接口：
  - cond_wait() 挂起
    - 等待一个条件
  - cond_signal() 唤醒
    - 条件已满足



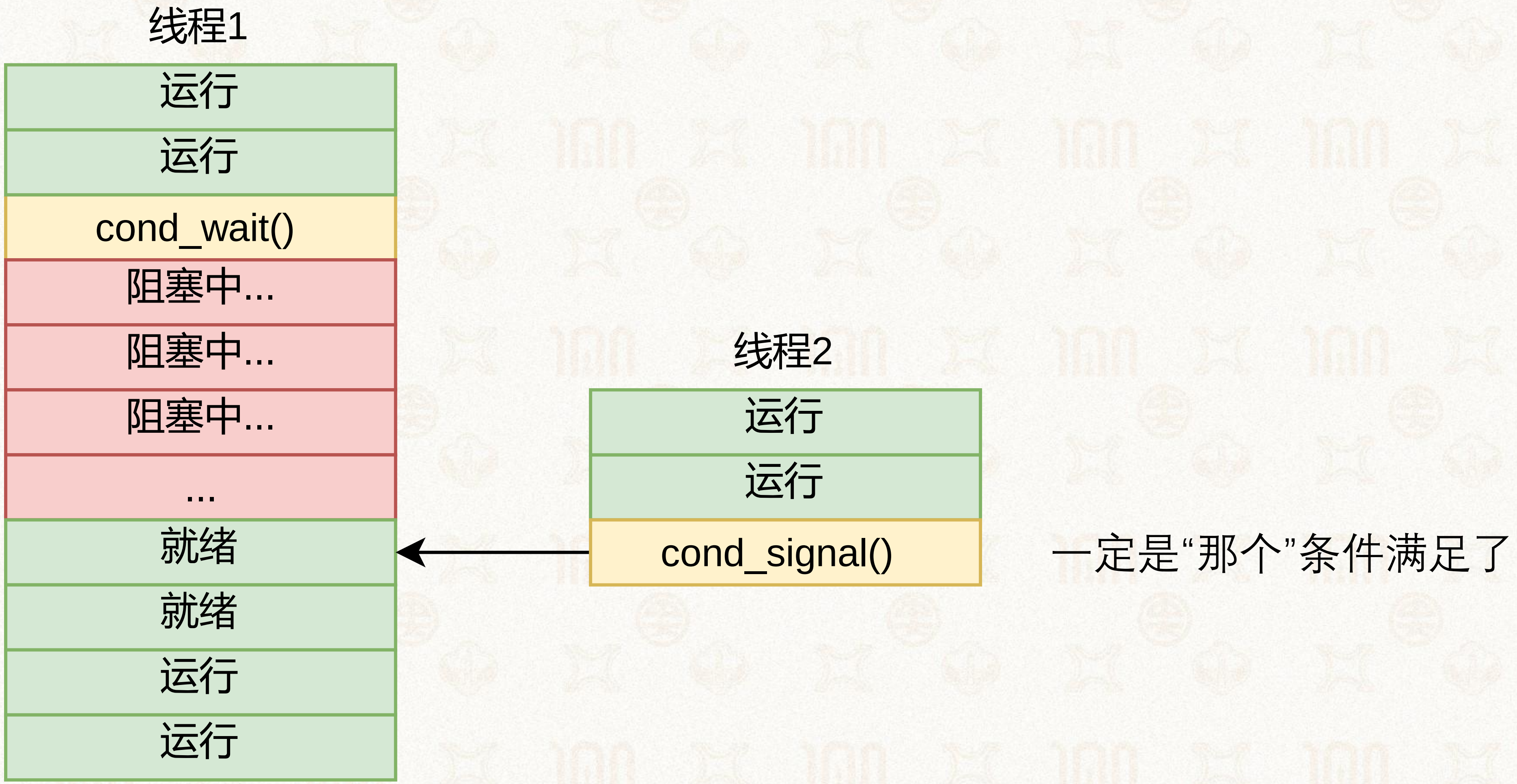
# 条件变量：线程运行状态

- 需要等待某个条件得以满足
- 自己把自己叫醒的?
- 谁睡觉时可以“眼观六路，耳听八方”?
- 重点：cond_signal()一定不是自己调用的
  - 这是和锁的最大区别
    - 锁：lock() …. unlock()

| 线程1 |
| --- |
| 运行 |
| 运行 |
| cond_wait() |
| 阻塞中… |
| 阻塞中… |
| 阻塞中… |
| … |
| cond_signal() |
| 就绪 |
| 就绪 |
| 运行 |
| 运行 |

# 条件变量：线程运行状态

- cond_wait()和cond_signal()分属于两个不同的线程



# 生产者消费者问题：条件变量

```c
int empty_slot = 5;
int filled_slot = 0;
struct cond empty_cond;
struct lock empty_cnt_lock;
struct cond filled_cond;
struct lock filled_cnt_lock;

void producer(void) {
    int new_msg;
    while(TRUE) {
        new_msg = produce_new();
        lock(&empty_cnt_lock);
        while(empty_slot == 0) {
            cond_wait(&empty_cond, &empty_cnt_lock);
        }
        empty_slot--;
        unlock(&empty_cnt_lock);
        buffer_add_safe(new_msg);
        lock(&filled_cnt_lock);
        filled_slot++;
        cond_signal(&filled_cond);
        unlock(&filled_cnt_lock);
    }
}
```

就是一个记号，可以表示任意条件

关于empty_slot的临界区

等待empty_cond被满足

在等empty_slot不为空

关于filled_slot的临界区

```c
void consumer(void) {
    int cur_msg;
    while(TRUE) {
        lock(&filled_cnt_lock);
        while(&filled_slot == 0) {
            cond_wait(&filled_cond, &filled_cnt_lock);
        }
        filled_slot--;
        unlock(&filled_cnt_lock);

        cur_msg = buffer_remove_safe();

        lock(&empty_cnt_lock);
        empty_slot++;
        cond_signal(&empty_cond);
        unlock(&empty_cnt_lock);

        consume_msg(cur_msg);
    }
}
```

条件被满足了，可以发信号了

等待empty_cond条件的线程可以被唤醒了

# 生产者消费者问题：条件变量

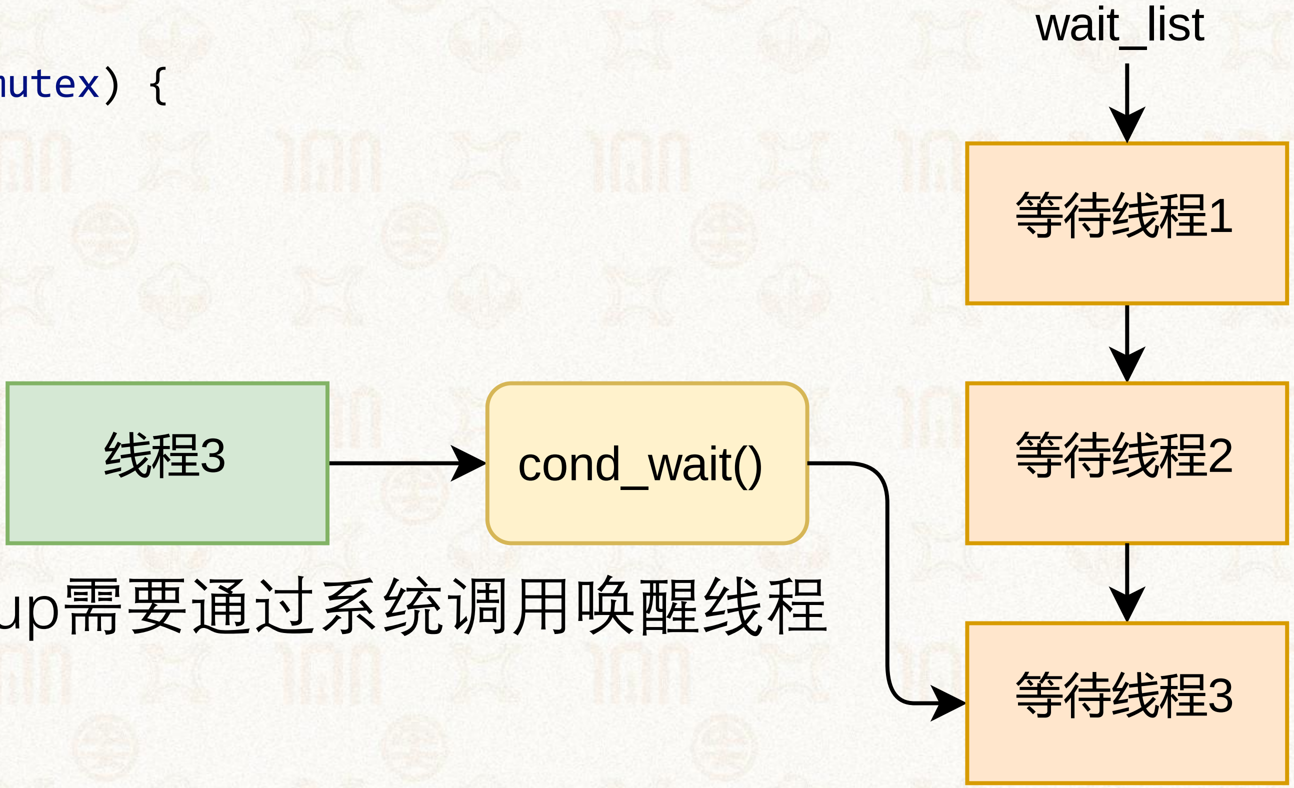
```
int empty_slot = 5;
int filled_slot = 0;
struct cond empty_cond;
struct lock empty_cnt_lock;
struct cond filled_cond;
struct lock filled_cnt_lock;

void producer(void) {
    int new_msg;
    while(TRUE) {
        new_msg = produce_new();
        lock(&empty_cnt_lock);
        while(empty_slot == 0) {
            cond_wait(&empty_cond, &empty_cnt_lock);
        }
        empty_slot--;
        unlock(&empty_cnt_lock);
        buffer_add_safe(new_msg);
        lock(&filled_cnt_lock);
        filled_slot++;
        cond_signal(&filled_cond);
        unlock(&filled_cnt_lock);
    }
}
```

```
void consumer(void) {
    int cur_msg;
    while(TRUE) {
        lock(&filled_cnt_lock);
        while(&filled_slot == 0) {
            cond_wait(&filled_cond, &filled_cnt_lock);
        }
        filled_slot--;
        unlock(&filled_cnt_lock);

        cur_msg = buffer_remove_safe();

        lock(&empty_cnt_lock);
        empty_slot++;
        cond_signal(&empty_cond);
        unlock(&empty_cnt_lock);

        consume_msg(cur_msg);
    }
}
```

# 条件变量的实现

```c
struct cond {
    struct thread *wait_list;
};

void cond_wait(struct cond *cond, struct lock *mutex) {
    list_append(cond->wait_list, thread_self());
    atomic_block_unlock(mutex); // 原子挂起并放锁
    lock(mutex); // 重新获得互斥锁
}

void cond_signal(struct cond *cond) {
    if(!list_empty(cond->wait_list)) {
        wakeup(list_remove(cond->wait_list));
    }
}

void cond_broadcast(struct cond *cond) {
    while(!list_empty(cond->wait_list)) {
        wakeup(list_remove(cond->wait_list));
    }
}
```

一次性唤醒所有等待线程




wakeup需要通过系统调用唤醒线程

# 互斥锁与条件变量



## 互斥锁:

- 保证临界区只有一个线程访问
- 参数是锁
  - lock()
  - unlock()
- 两接口在同一个线程内操作

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

}

## 条件变量

- 避免被堵在外面的线程循环等待
- 参数是cond结构体变量
  - cond_wait()
  - cond_signal()
- 两接口被不同线程调用

互斥锁与条件变量解决的不是同一个问题，条件变量需要与互斥锁配合使用

# 大纲

- 同步问题的背景
  - 多核场景
  - 生产者消费者模型
  - 临界区问题
- 互斥锁
  - 皮特森算法
  - 原子操作
  - 互斥锁抽象
    - 自旋锁
    - 排号自旋锁
- 条件变量
- 信号量
  - PV原语
- 读写锁
- 同步原语产生的问题
  - 死锁
    - 银行家算法
  - 活锁
  - 优先级反转

# 条件变量的不足

```c
int empty_slot = 5;
int filled_slot = 0;
struct cond empty_cond;
struct lock empty_cnt_lock;
struct cond filled_cond;
struct lock filled_cnt_lock;

void producer(void) {
    int new_msg;
    while(TRUE) {
        new_msg = produce_new();
        lock(&empty_cnt_lock);
        while(empty_slot == 0) {
            cond_wait(&empty_cond, &empty_cnt_lock);
        }
        empty_slot--;
        unlock(&empty_cnt_lock);
        buffer_add_safe(new_msg);
        lock(&filled_cnt_lock);
        filled_slot++;
        cond_signal(&filled_cond);
        unlock(&filled_cnt_lock);
    }
}
```

- “那个”条件是“哪个”条件？
  - 程序员心里想的条件是”empty_slot不为0”
  - 从代码里很难看出来这个假设
  - 因为cond定义与条件声明是分离的
  - 新的程序员忘了cond和谁对应怎么办？
  - 或者，不小心写错对应关系了怎么办？
- 需要简化设计，把条件和变量真正地统一起来

# 简化设计：信号量(semaphore)

```
sem_t empty_slot;
sem_t filled_slot;

void producer(void) {
    int new_msg;
    while(TRUE) {
        new_msg = produce_new();
        wait(&empty_slot); // P
        buffer_add_safe(new_msg);
        signal(&filled_slot); // V
    }
}
void consumer(void) {
    int cur_msg;
    while(TRUE) {
        wait(&filled_slot); // P
        cur_msg = buffer_remove_safe();
        signal(&empty_slot); // V
        consume_msg(cur_msg);
    }
}
```

- 根据剩余资源的数量决定线程执行或等待
- PV原语：
  - P: “检验” 代码中用wait来表示
  - V: “自增” 代码中用signal来表示

PV的逻辑含义：

```
void wait(int *S) {
    while(*S <= 0)
        ; // 循环忙等
    *S = *S - 1;
}

void signal(int *S) {
    *S = *S + 1;
}
```

# 简化设计：信号量(semaphore)

```c
sem_t empty_slot;
sem_t filled_slot;

void producer(void) {
    int new_msg;
    while(TRUE) {
        new_msg = produce_new();
        wait(&empty_slot); // P
        buffer_add_safe(new_msg);
        signal(&filled_slot); // V
    }
}
void consumer(void) {
    int cur_msg;
    while(TRUE) {
        wait(&filled_slot); // P
        cur_msg = buffer_remove_safe();
        signal(&empty_slot); // V
        consume_msg(cur_msg);
    }
}
```

- 信号量是面向多个线程访问有限数量的共享资源
- 互斥锁主要面向两个线程

PV的逻辑含义：

```c
void wait(int *S) {
    while(*S <= 0)
        ; // 循环忙等
    *S = *S - 1;
}

void signal(int *S) {
    *S = *S + 1;
}
```

# 信号量的一种实现

```c
struct sem {
    int value;  // value为正，表示剩余资源数量
                // value为负，绝对值表示正在等待的线程数量
    int wakeup; // 应当唤醒(可用资源)的资源数量
    struct lock sem_lock;
    struct cond sem_cond;
};

void wait(struct sem *S) {
    lock(&S->sem_lock);
    S->value--;
    if(S->value < 0) {
        do {
            cond_wait(&S->sem_cond, &S->sem_lock);
        } while(S->wakeup == 0);
        S->wakeup--;
    }
    unlock(&S->sem_lock);
}
```

```c
void signal(struct sem *S) {
    lock(&S->sem_lock);
    S->value++;
    if(S->value <= 0) {
        S->wakeup++;
        cond_signal(&S->sem_cond);
    }
    unlock(&S->sem_lock);
}
```

用互斥锁、条件变量实现用法简单的信号量操作

# 填空题 7分

设置

SUN YAT-SEN UNIVERSITY

```
int x = 1;
struct sem a, b, c;
void init(void) {
    a->value = [填空1];
    b->value = [填空2];
    c->value = [填空3];
}

void thread1(void) {
    while( x != 12) {
        [填空4];
        x = x * 2;
        [填空5];
    }
    exit(0);
}

void thread1(void) {
    while( x != 12) {
        [填空6];
        x = x * 3;
        [填空7];
    }
    exit(0);
}
```

- 教材P328的题目：如果需要保证两个线程都一定可以终止运行，请填写代码中空出的部分
- 在thread1和thread2的函数中，只能填写signal / wait, 例如signal(a), wait(b)，或者不填
- 每个空位中可填写的操作数量不限。

作答

# 大纲

- 同步问题的背景
  - 多核场景
  - 生产者消费者模型
  - 临界区问题
- 互斥锁
  - 皮特森算法
  - 原子操作
  - 互斥锁抽象
    - 自旋锁
    - 排号自旋锁
- 条件变量
- 信号量
  - PV原语
- 读写锁
- 同步原语产生的问题
  - 死锁
    - 银行家算法
  - 活锁
  - 优先级反转

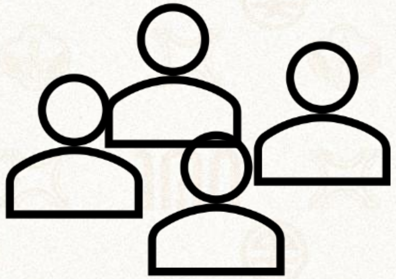
# 公告栏问题

这个公告栏
要撤走了

写者

公告栏

操作系统期末考试范围：
1、操作系统概述
2、硬件结构
3、操作系统结构
…



读者

别挤，再挤
就看不到了

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？

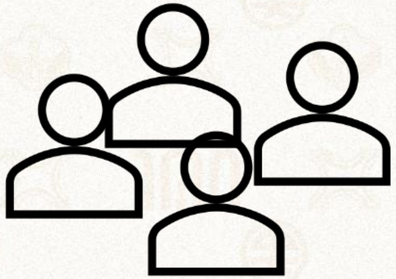
# 公告栏问题

这个公告栏
要撤走了

写者

公告栏

操作系统期中考试范围：
1、操作系统概述
2、硬件结构
3、操作系统结构
…



别挤，再挤
就看不到了

读者

思考：多个读者如果希望读公告栏，他们互斥吗？

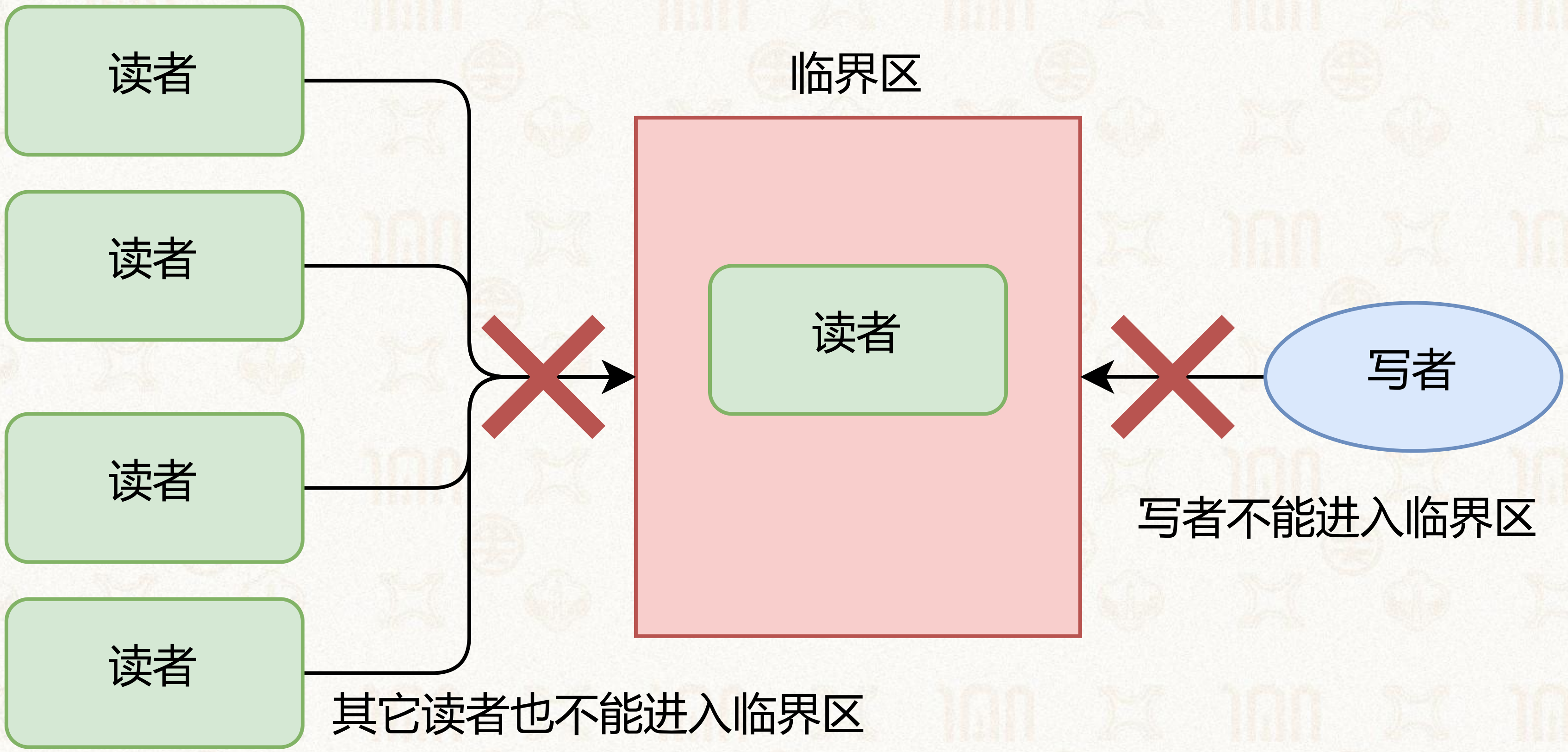
不互斥

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

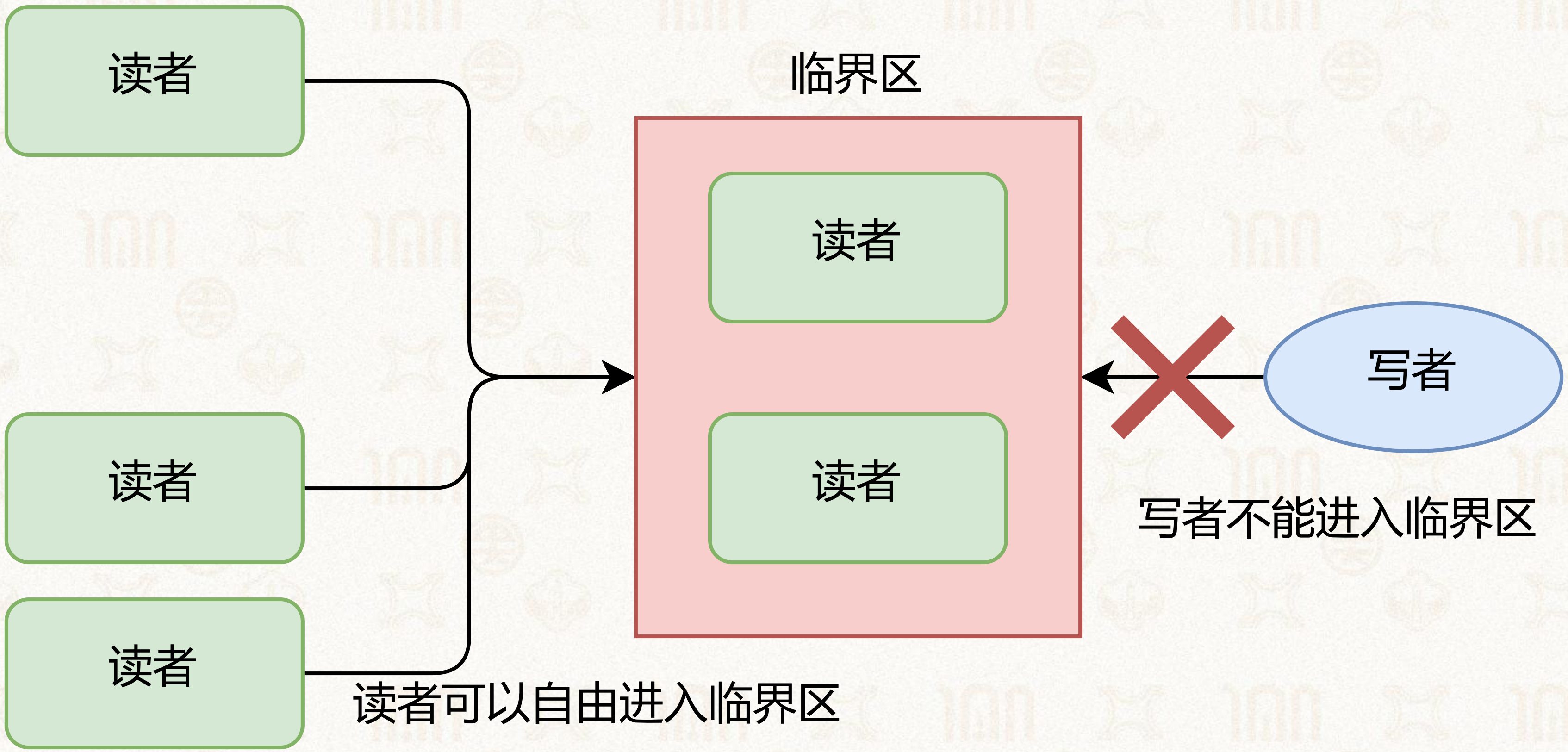
# 读写锁

- 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



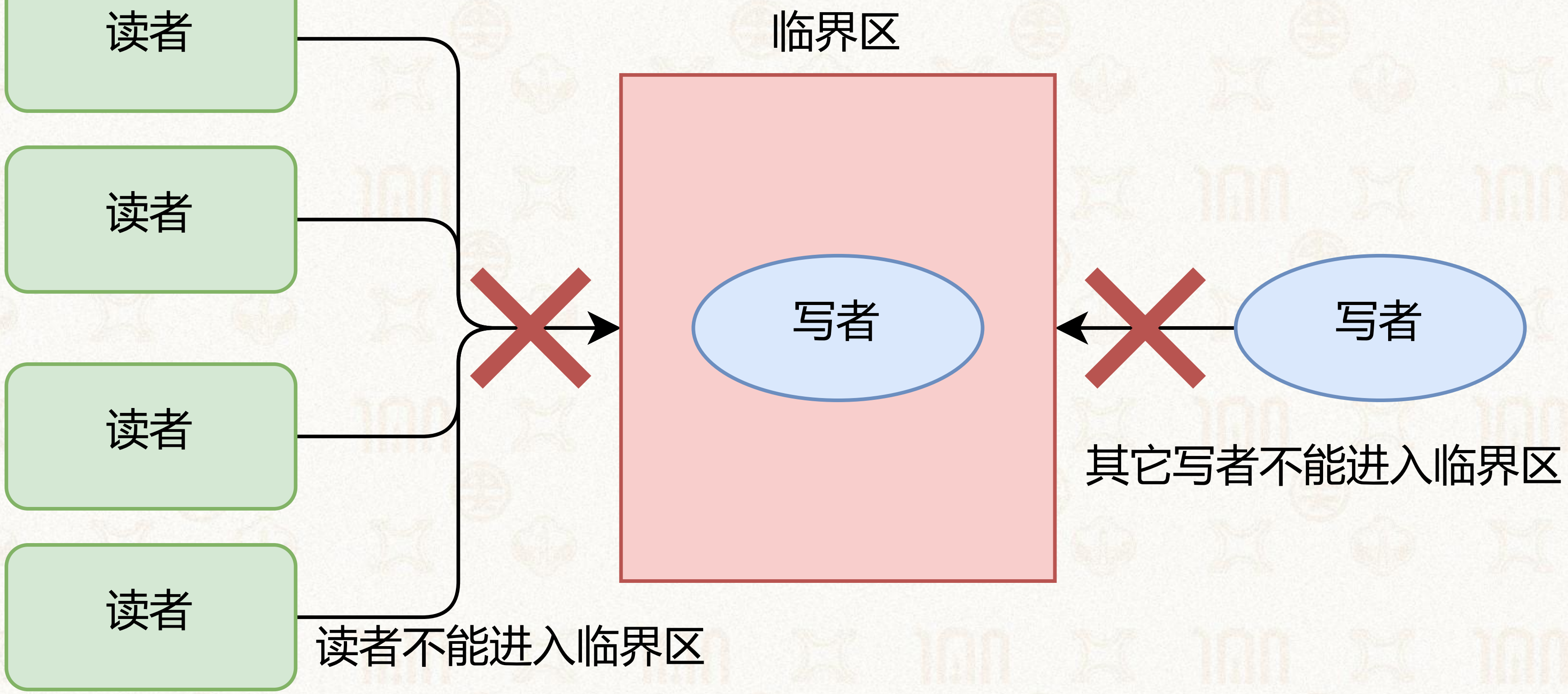
# 读写锁

- 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

- 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



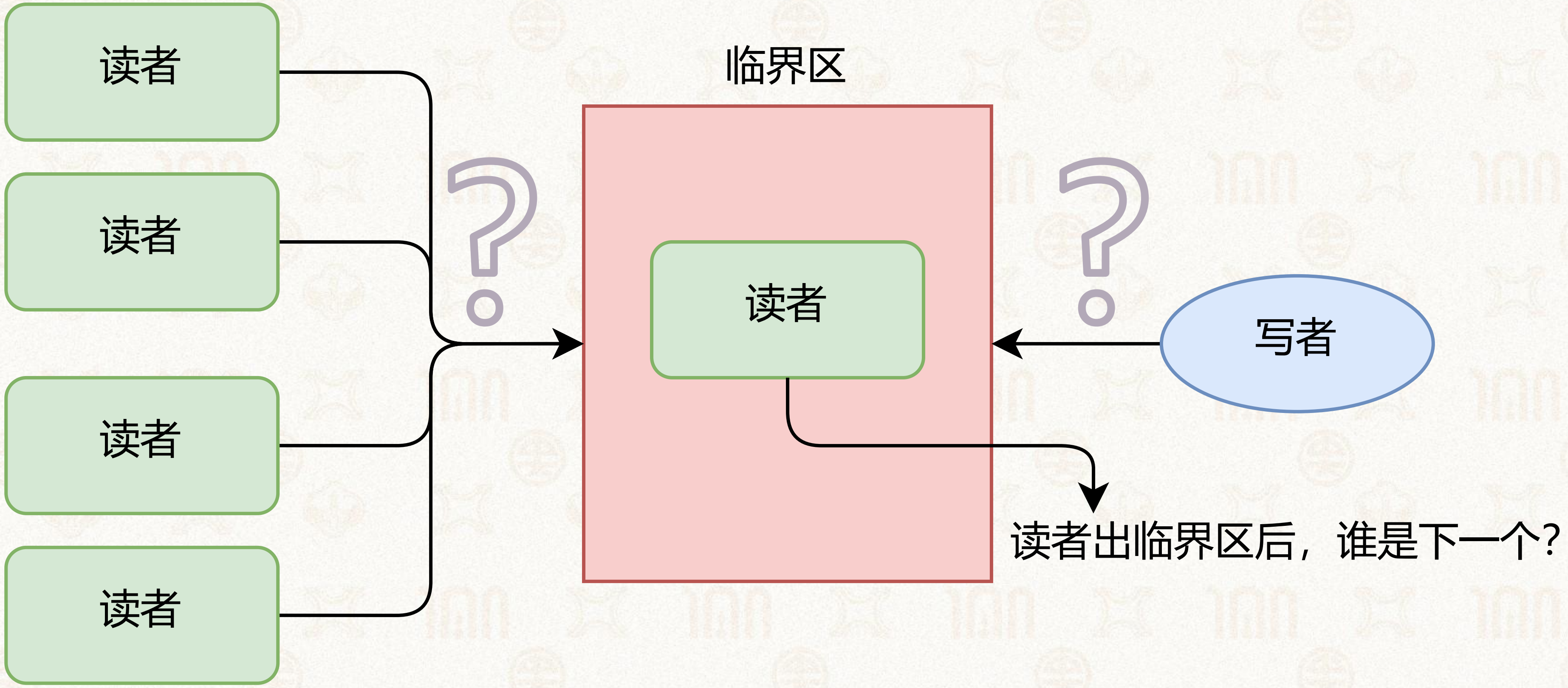
# 读写锁的使用示例

- 考虑写者较少，而读者较多的场景

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

void reader(void) {
    lock_reader(&lock);
    read_data(data); // 读临界区
    unlock_reader(&lock);
}

void writer(void) {
    lock_writer(&lock);
    update_data(data); // 写临界区
    unlock_writer(&lock);
}
```

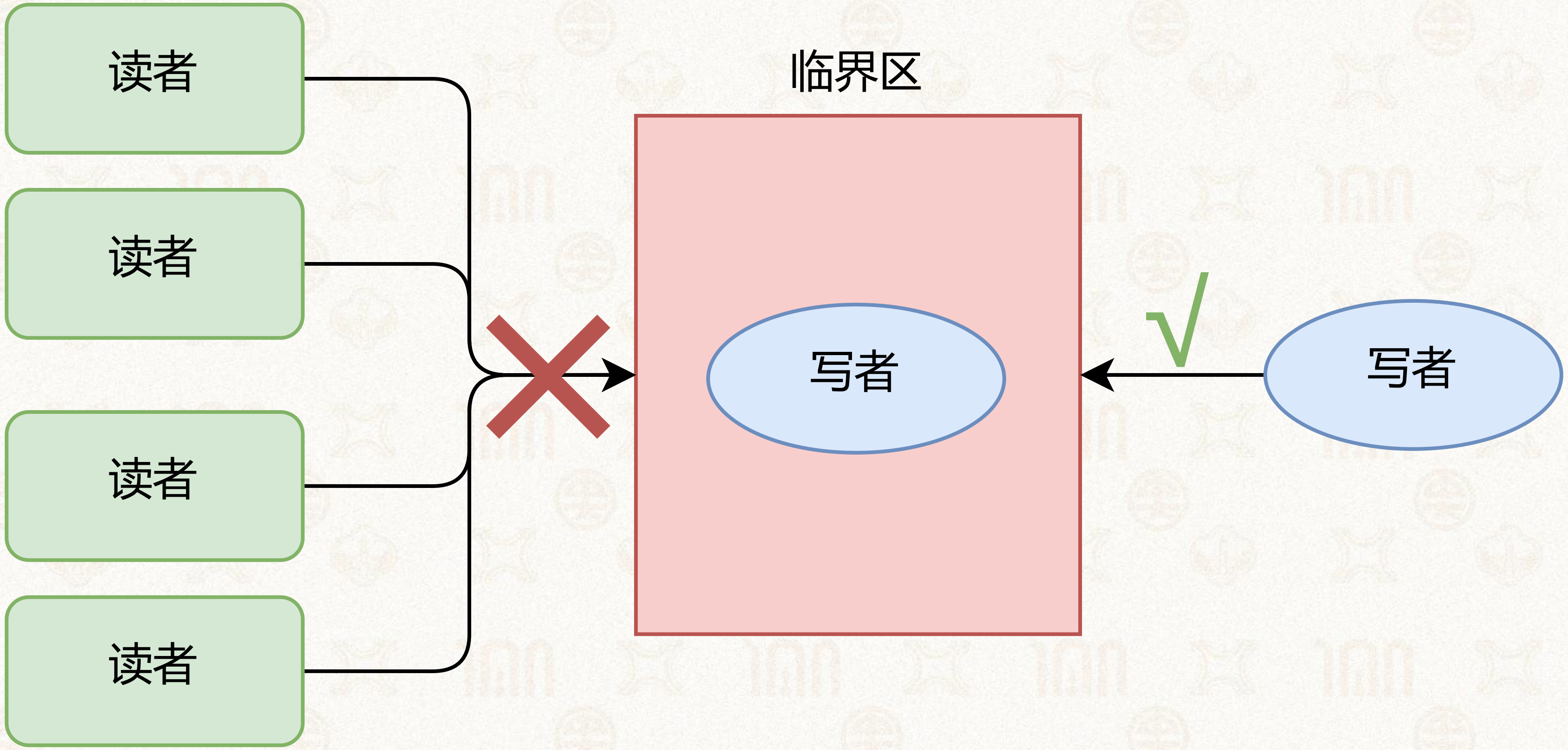
# 读写锁的偏向性

- 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？



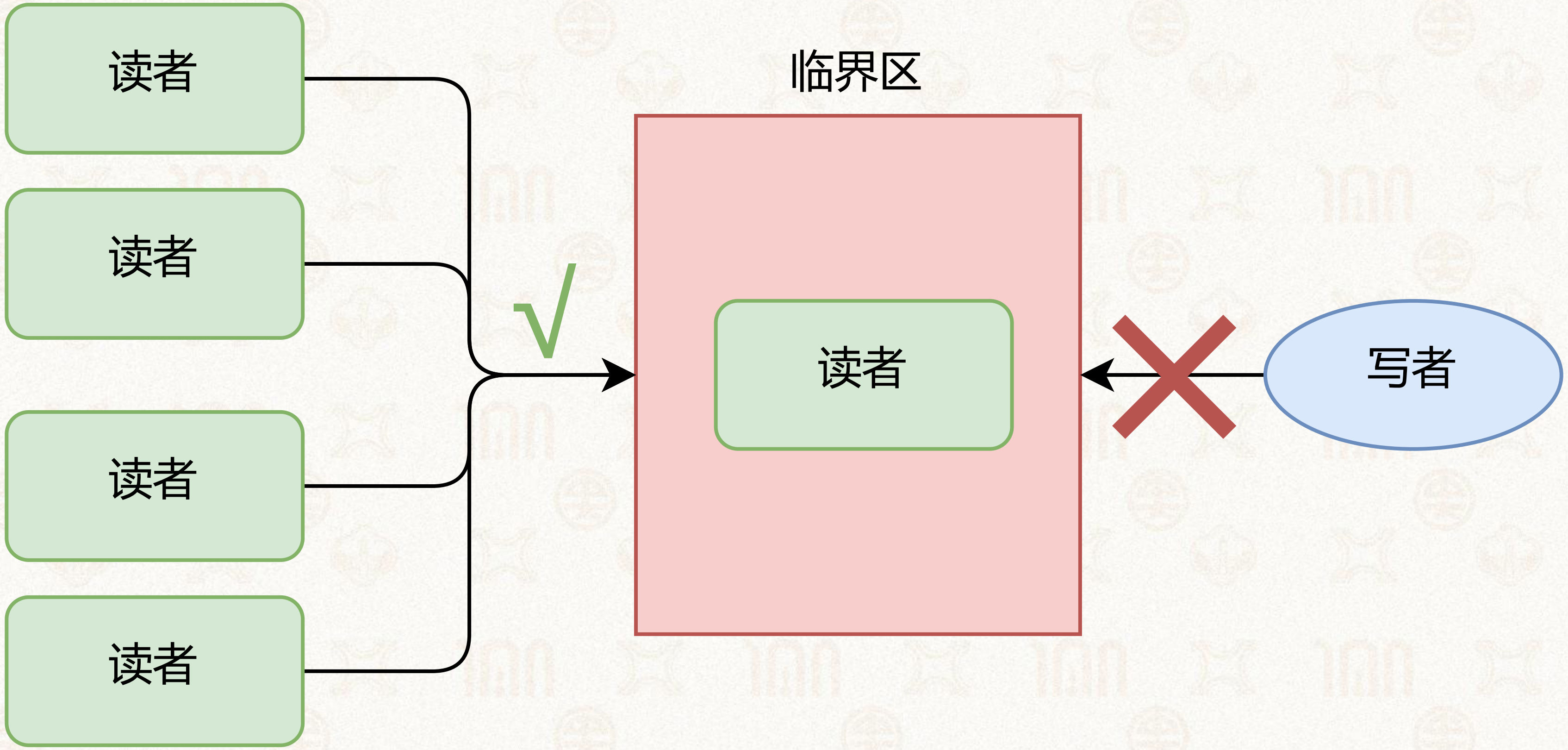
# 读写锁的偏向性

- 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并离开后才可进入 (更加公平)



# 读写锁的偏向性

- t2：另一个读者能否进入临界区？
- 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)



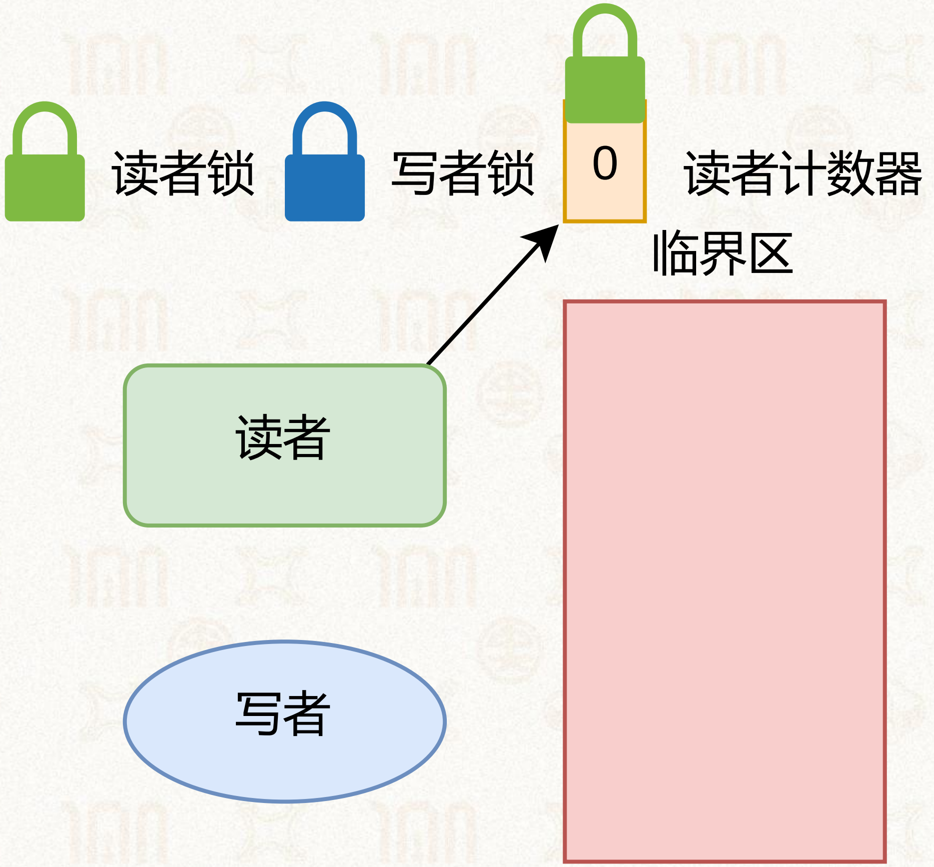
# 偏向读者的读写锁实现示例

```c
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```

```c
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

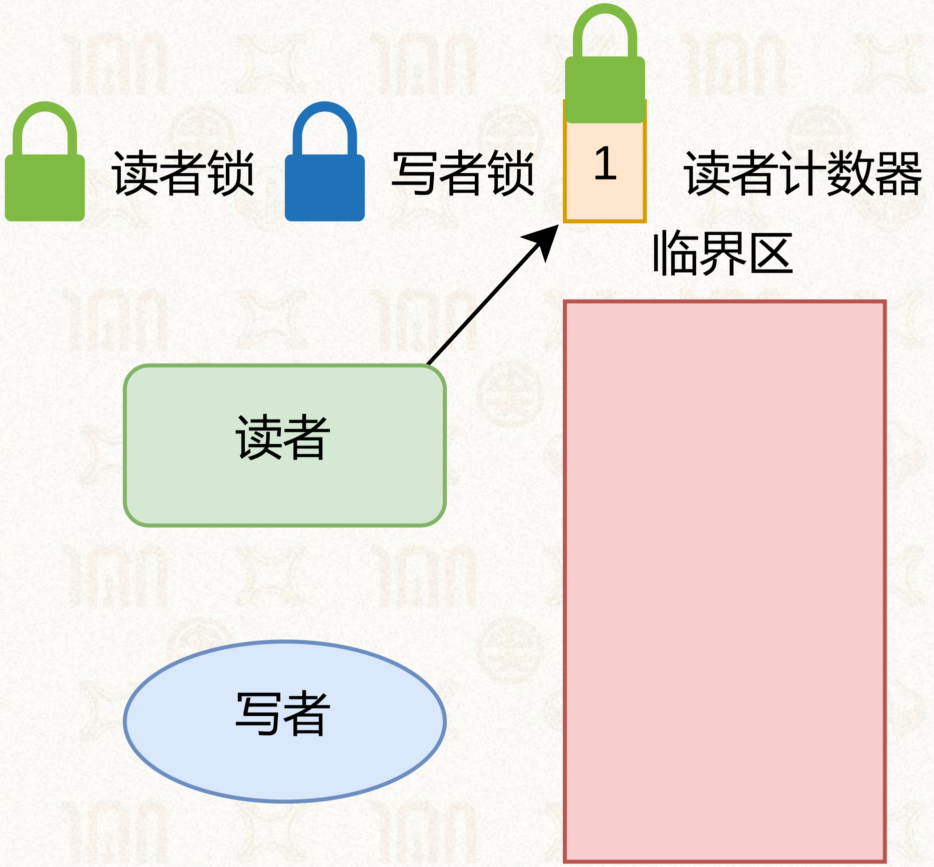
```c
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




1. 给读者计数器加个锁，再去更新读者计数器

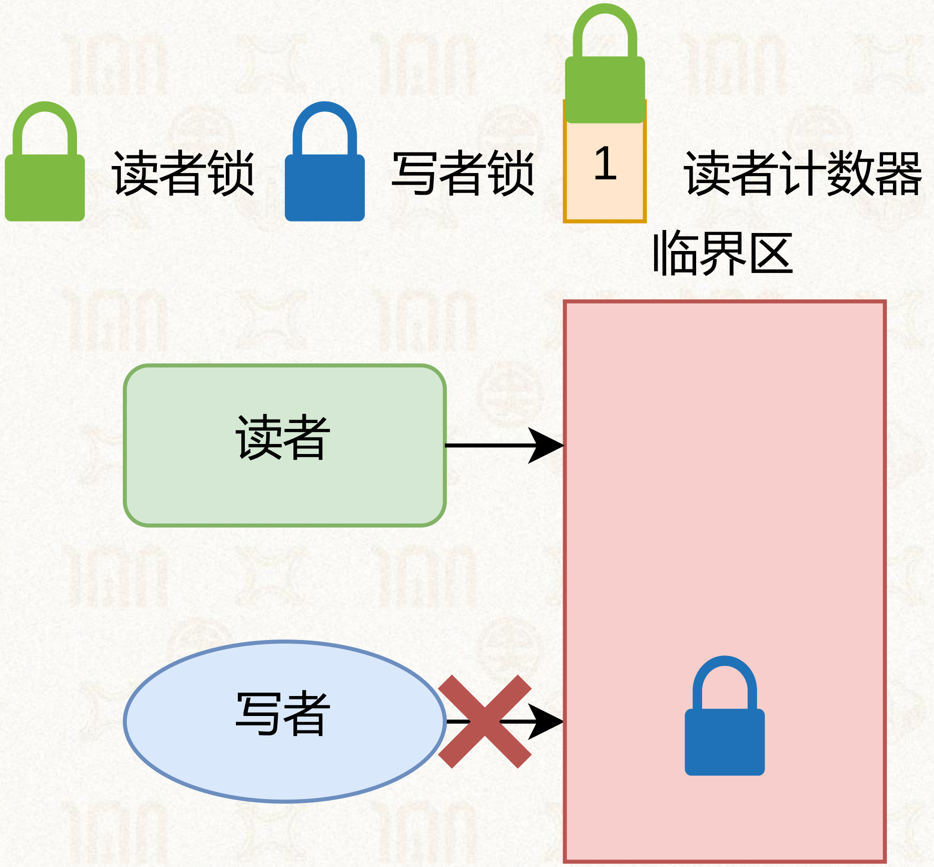
# 读写锁的实现：偏向读者为例

```c
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```



# 读写锁的实现：偏向读者为例

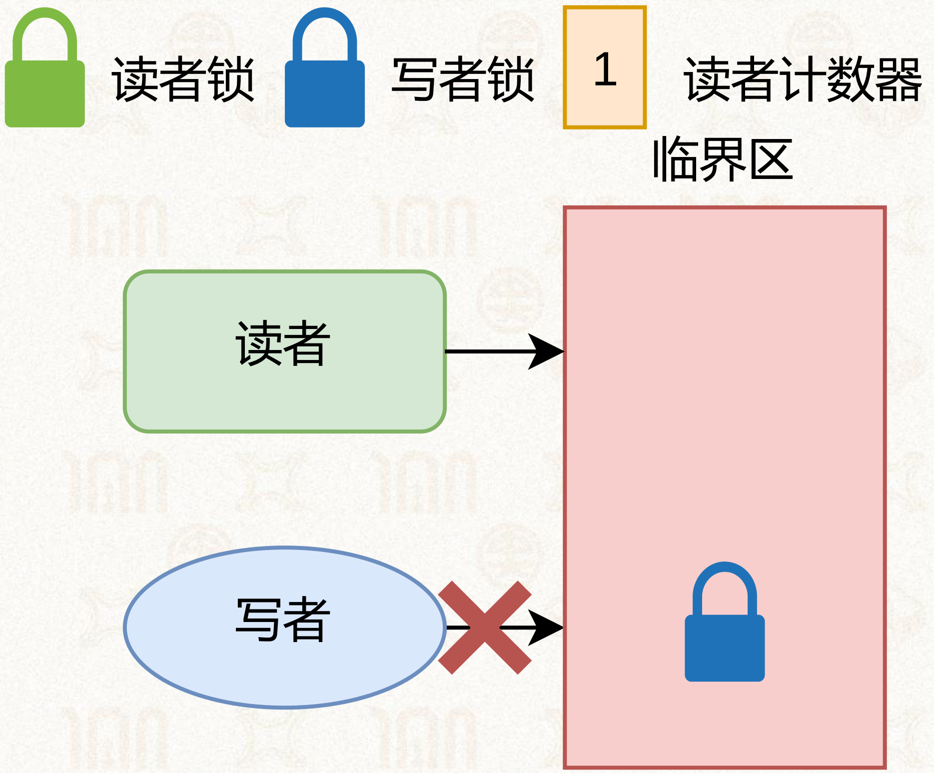
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




2. 如果没有读者在，拿写锁避免写者进入

# 读写锁的实现：偏向读者为例

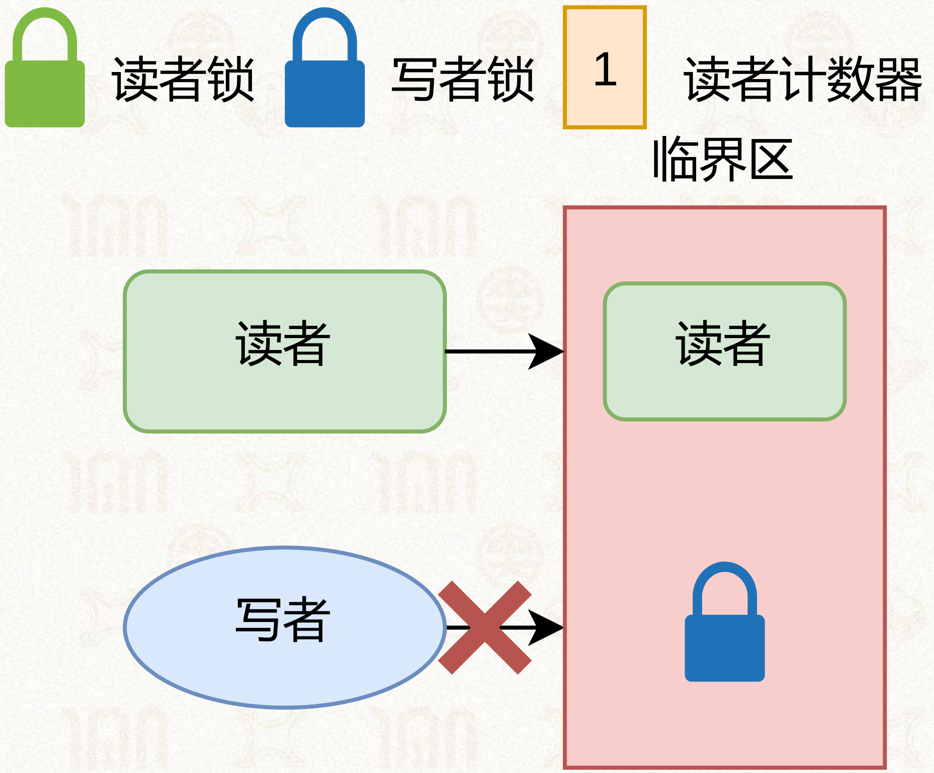
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




3. 释放读者锁

# 读写锁的实现：偏向读者为例

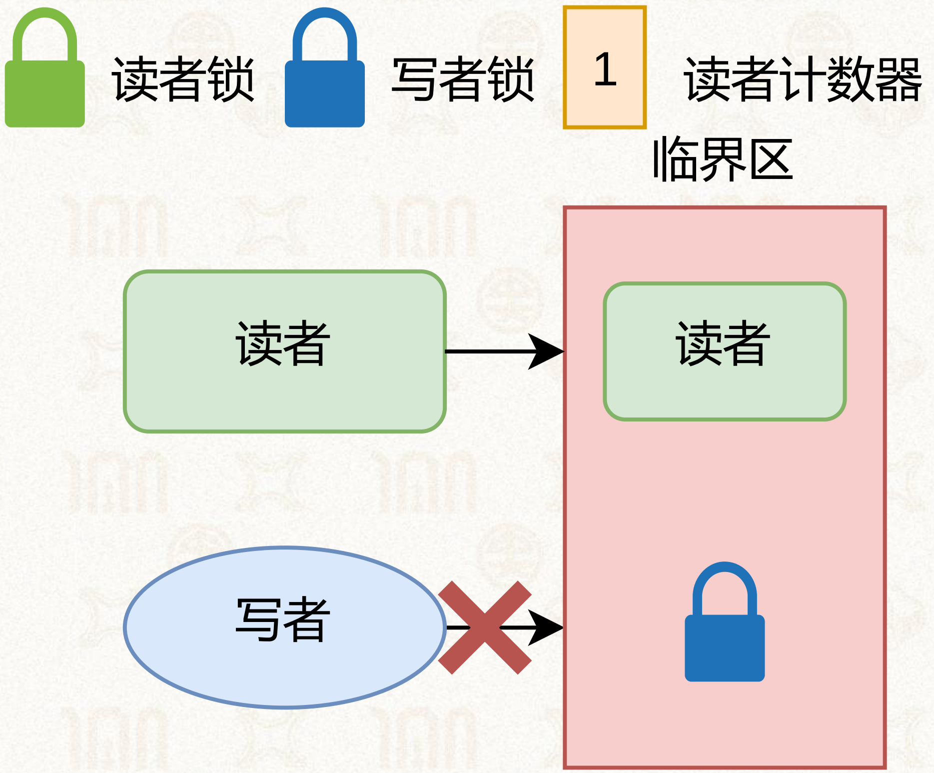
```c
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




第一个读者进入临界区

# 读写锁的实现：偏向读者为例

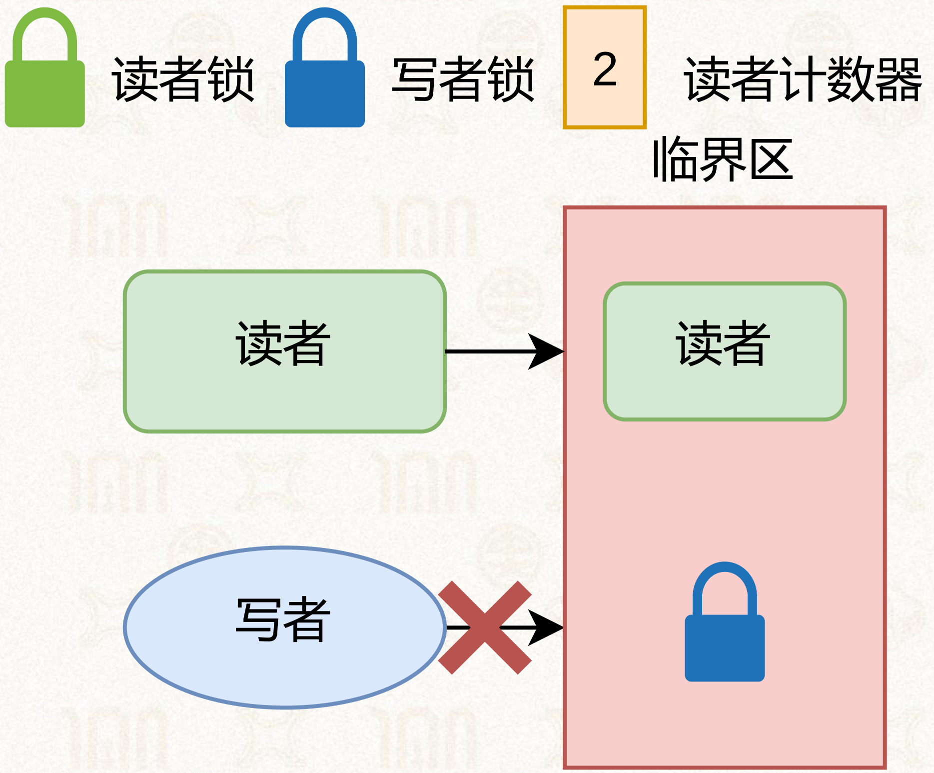
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




再来一个新的读者

# 读写锁的实现：偏向读者为例

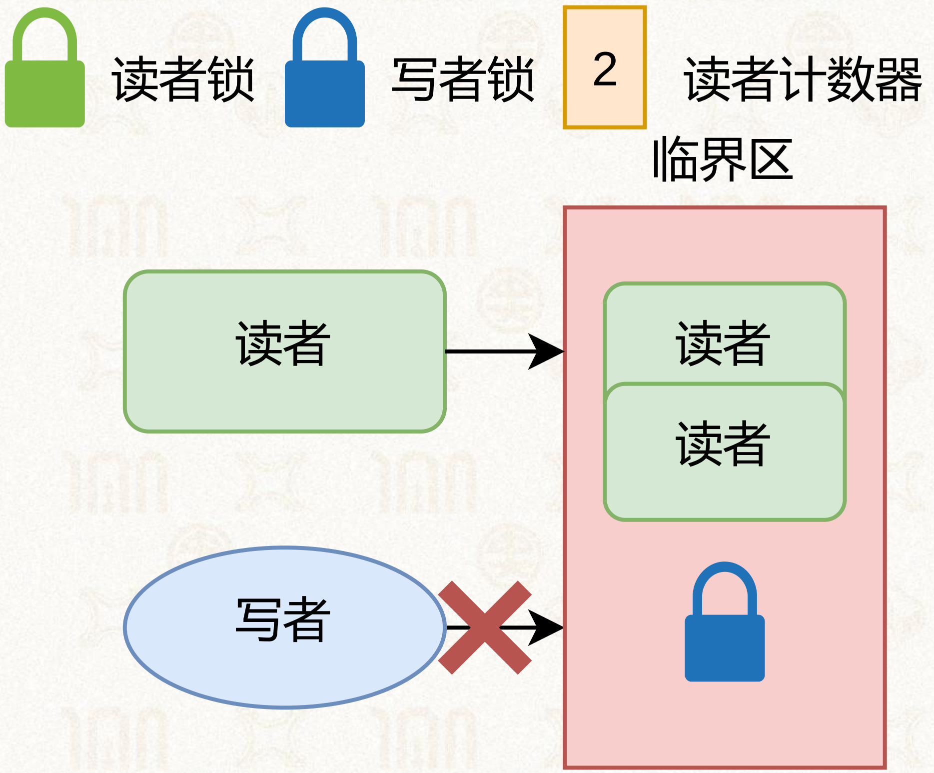
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




只需将读者计数器加1

# 读写锁的实现：偏向读者为例

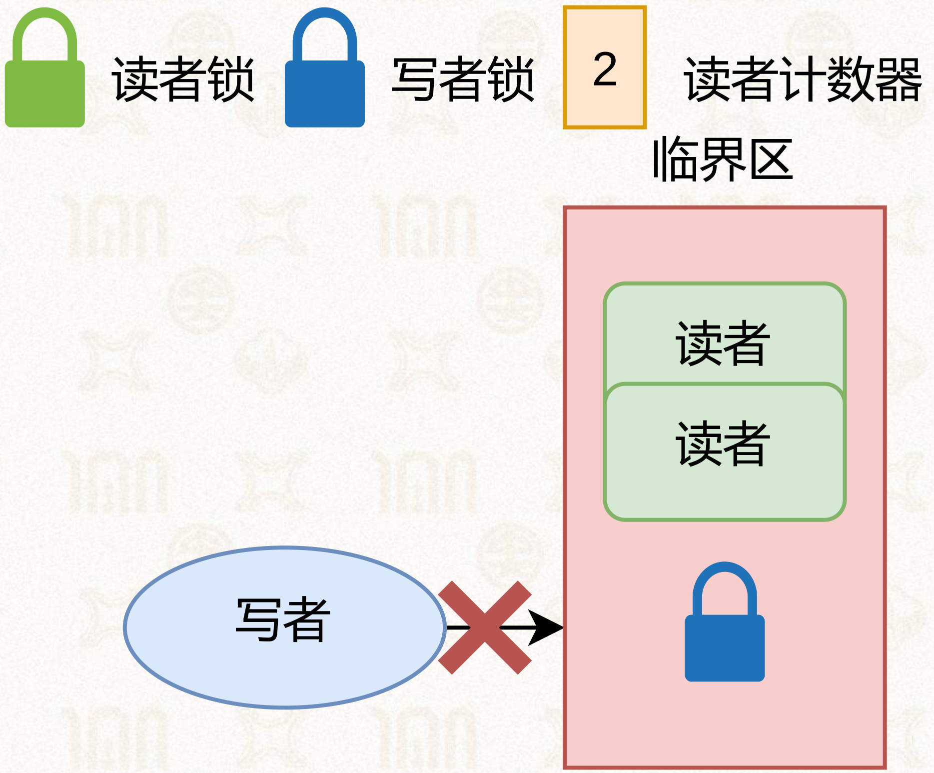
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




此时在临界区内有两个读者

# 读写锁的实现：偏向读者为例

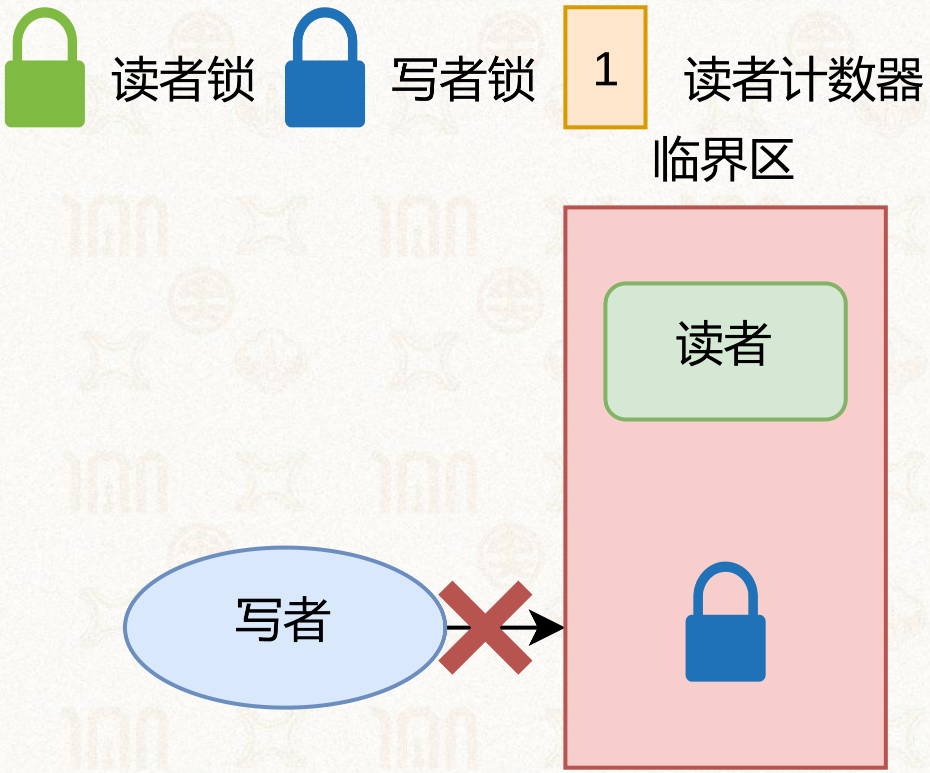
```c
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&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);
}
```




1. 写者尝试拿写锁，等待

# 读写锁的实现：偏向读者为例

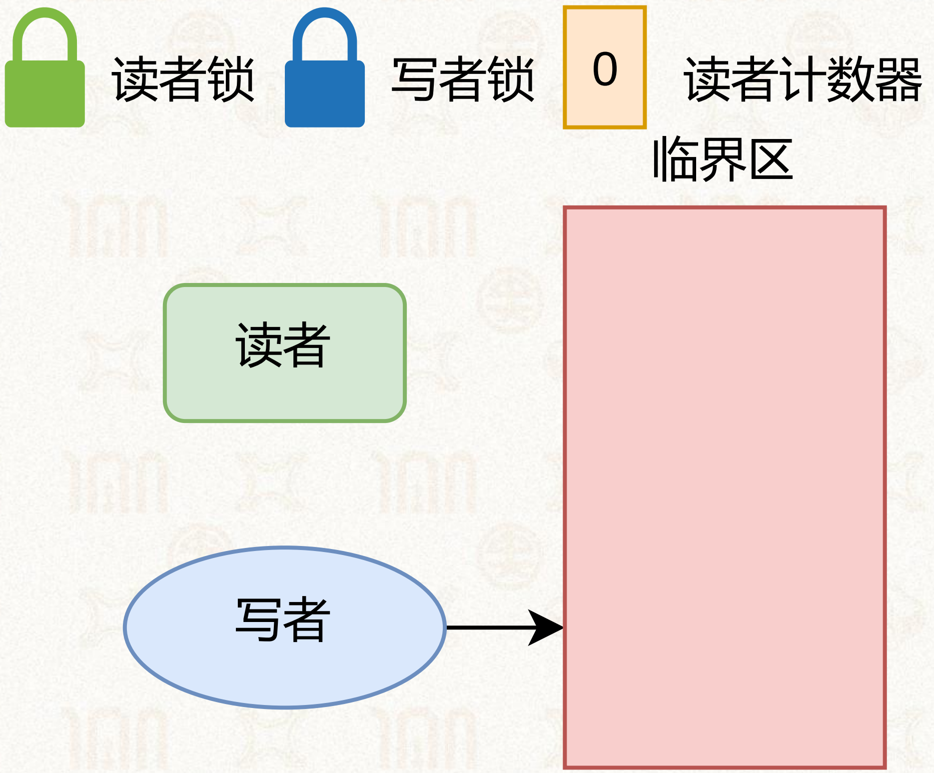
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




读者开始退出

# 读写锁的实现：偏向读者为例

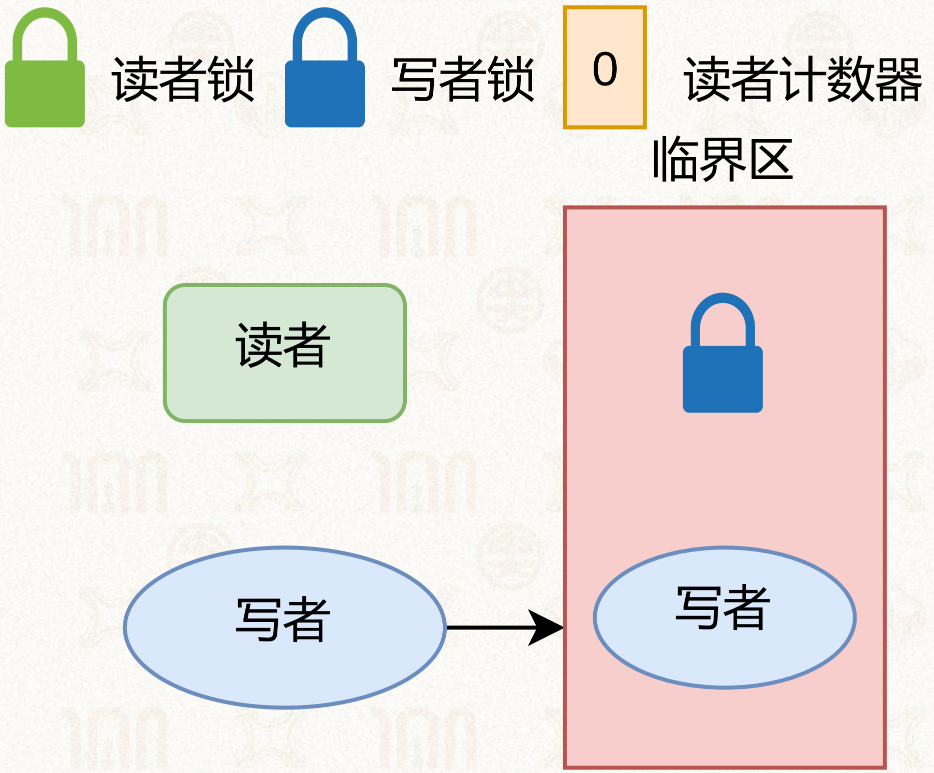
```c
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}

void unlock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt--;
    if(lock->reader_cnt == 0) {// 最后一个读者
        unlock(&lock->writer_lock);
    }
    unlock(&lock->reader_lock);
}
```




最后一个读者退出时释放写者锁

# 读写锁的实现：偏向读者为例

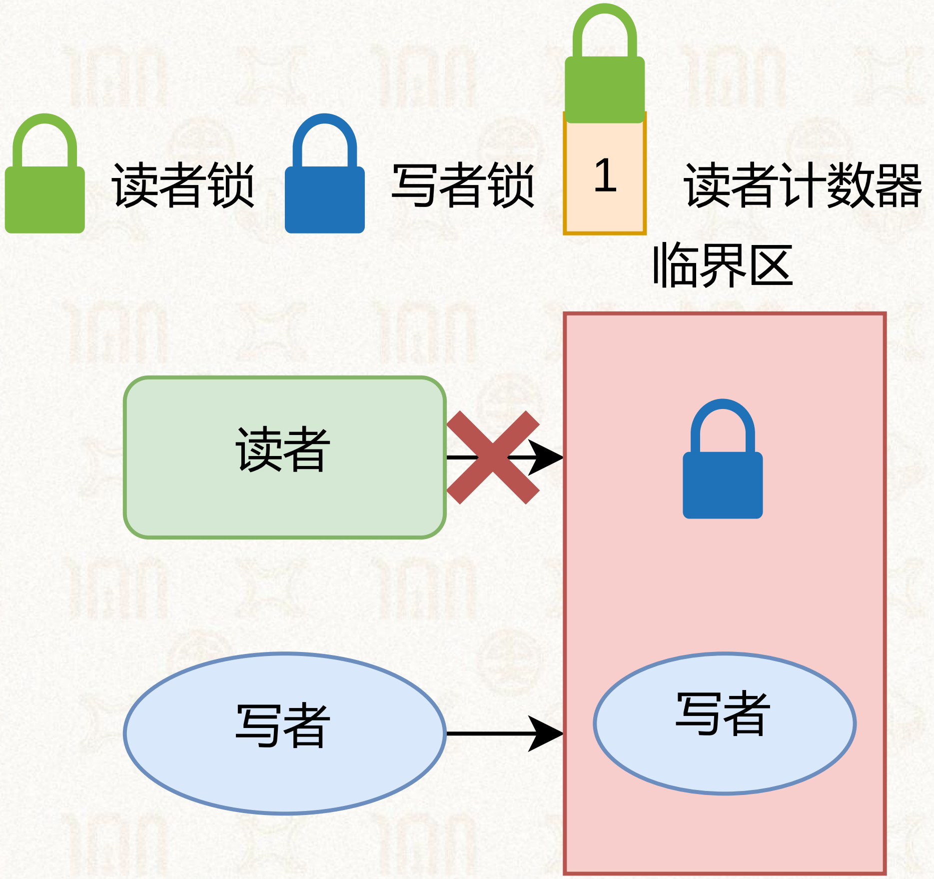
```c
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&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);
}
```




没有读者时，写者进入，并加写者锁

# 读写锁的实现：偏向读者为例

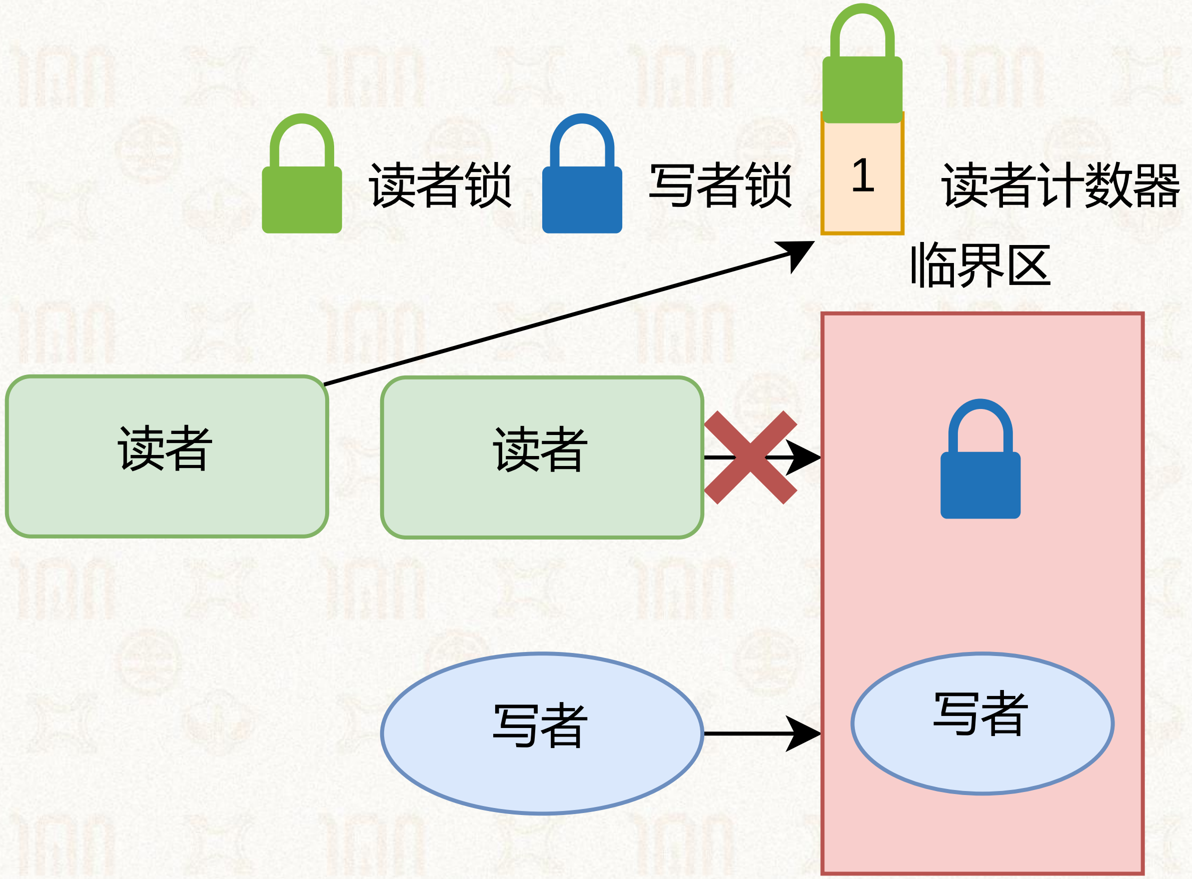
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&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);
}
```




此时读者被锁住，不能进入

# 读写锁的实现：偏向读者为例

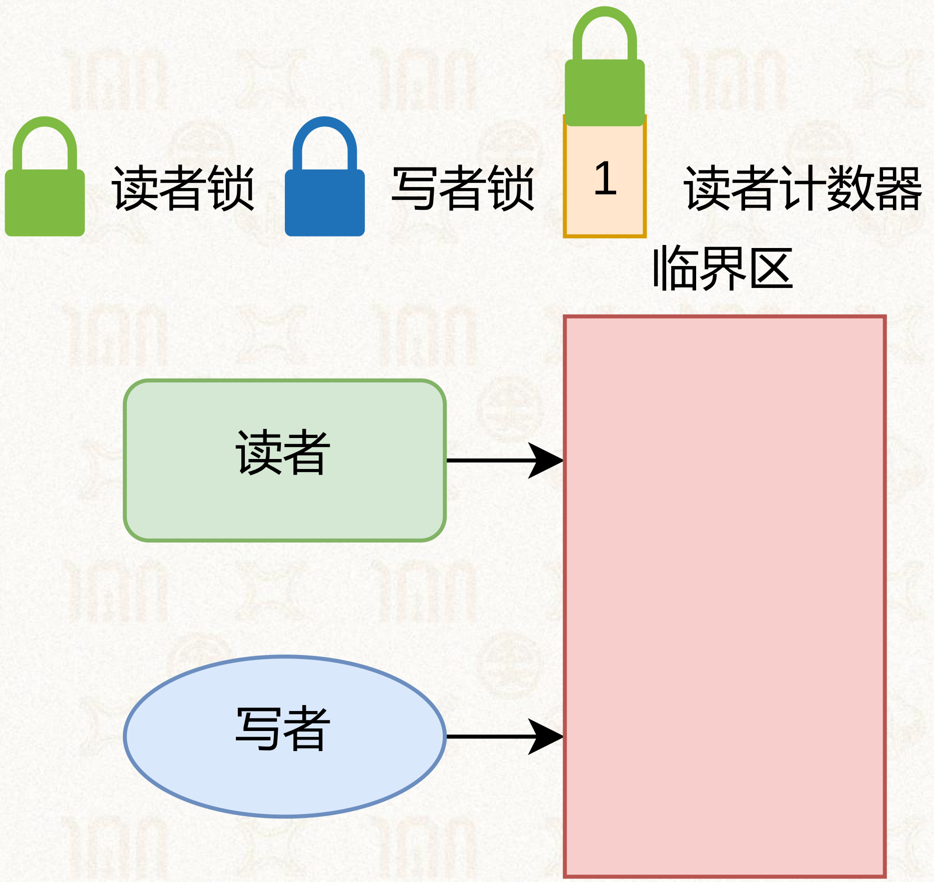
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
→   lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&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);
}
```




第二个读者也想来时，会被堵在读者锁的位置
所以读者锁也有价值，阻塞其它读者

# 读写锁的实现：偏向读者为例

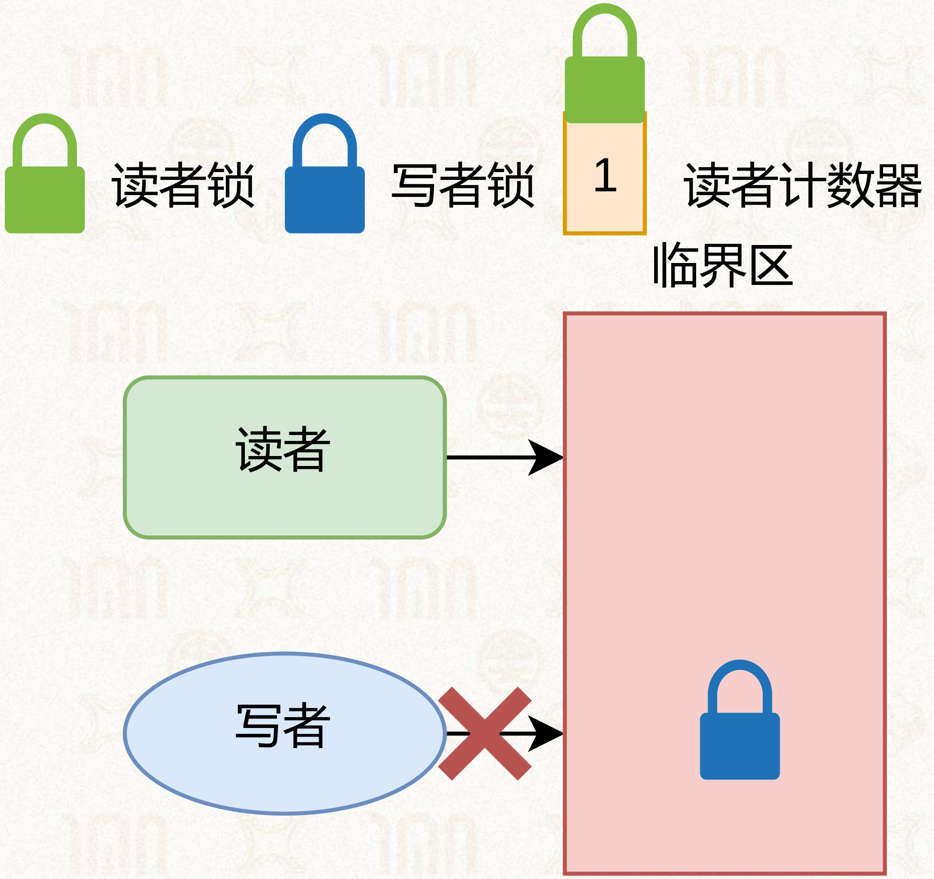
```
struct rwlock {
    int reader_cnt;
    struct lock reader_lock;
    struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&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);
}
```




写者离开，释放写者锁

# 读写锁的实现：偏向读者为例

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

void lock_reader(struct rwlock *lock) {
    lock(&lock->reader_lock);
    lock->reader_cnt++;
    if(lock->reader_cnt == 1) { // 第一个读者
        lock(&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);
}
```




读者加一个写者锁，然后正常进入

# 偏向写者的读写锁实现示例

```c
struct rwlock {
    volatile int reader_cnt;
    volatile bool has_writer;
    struct lock lock;
    struct cond reader_cond;
    struct cond writer_cond;
};
void lock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond,
                  &rwlock->lock);
    }
    rwlock->reader_cnt++;
    unlock(&rwlock->lock);
}
void unlock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    rwlock->reader_cnt--;
    if(rwlock->reader_cnt == 0) {
        cond_signal(&rwlock->reader_cond);
    }
    unlock(&rwlock->lock);
}
```

```c
void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}

void unlock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    rwlock->has_writer = FALSE;
    cond_broadcast(&rwlock->writer_cond);
    unlock(&rwlock->lock);
}
```

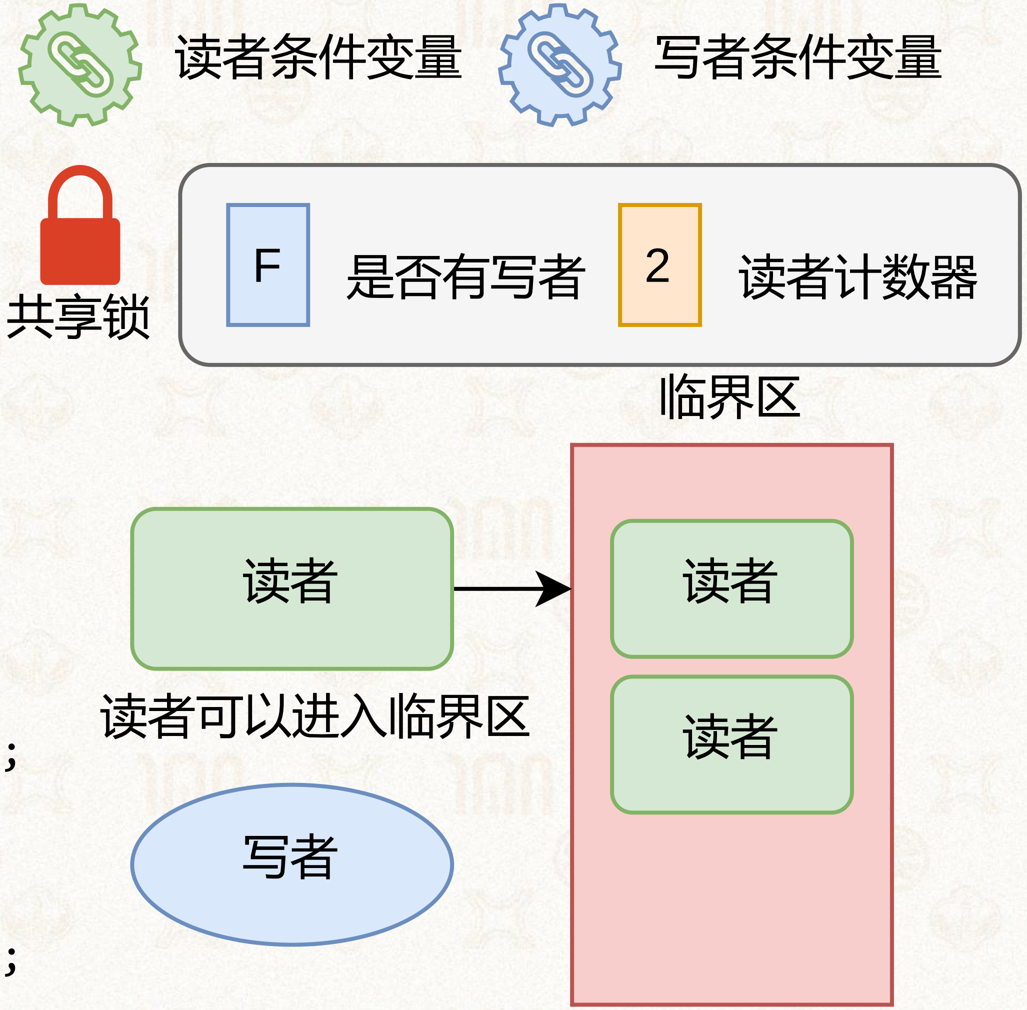
# 读写锁的实现：偏向写者为例

➢ 假设读者在临界区

```
void lock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond,
                  &rwlock->lock);
    }
    rwlock->reader_cnt++;
    unlock(&rwlock->lock);
}
```

没有写者，随便进

```
void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}
```

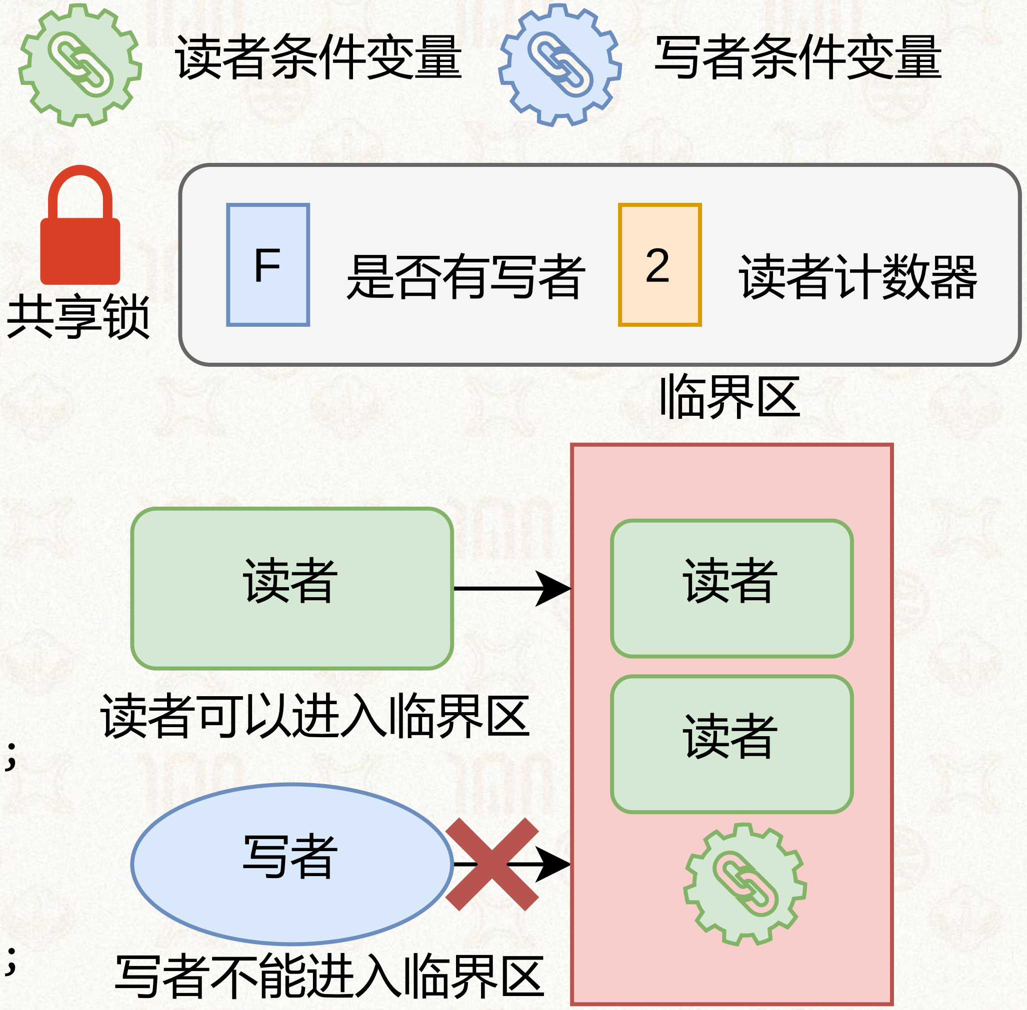


# 读写锁的实现：偏向写者为例

- 假设读者在临界区

```
void lock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond,
                  &rwlock->lock);
    }
    rwlock->reader_cnt++;
    unlock(&rwlock->lock);
}

void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}
```

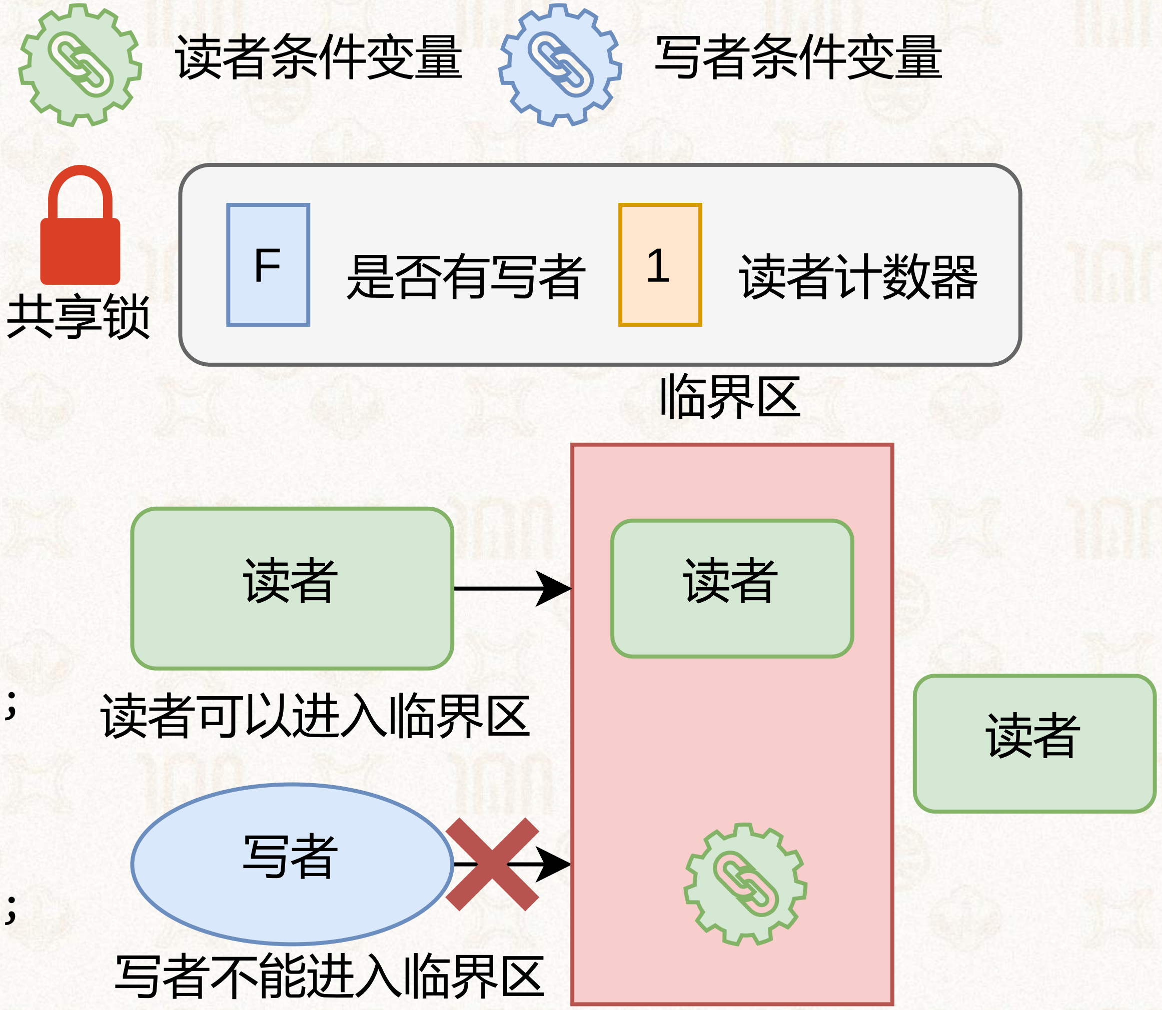
需要等读者都离开，这里和信号量等价



# 读写锁的实现：偏向写者为例

- 假设读者在临界区，但准备离开

```
void unlock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    rwlock->reader_cnt--;    读者离开，计数器减一
    if(rwlock->reader_cnt == 0) {
        cond_signal(&rwlock->reader_cond);
    }
    unlock(&rwlock->lock);
}

void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}
```



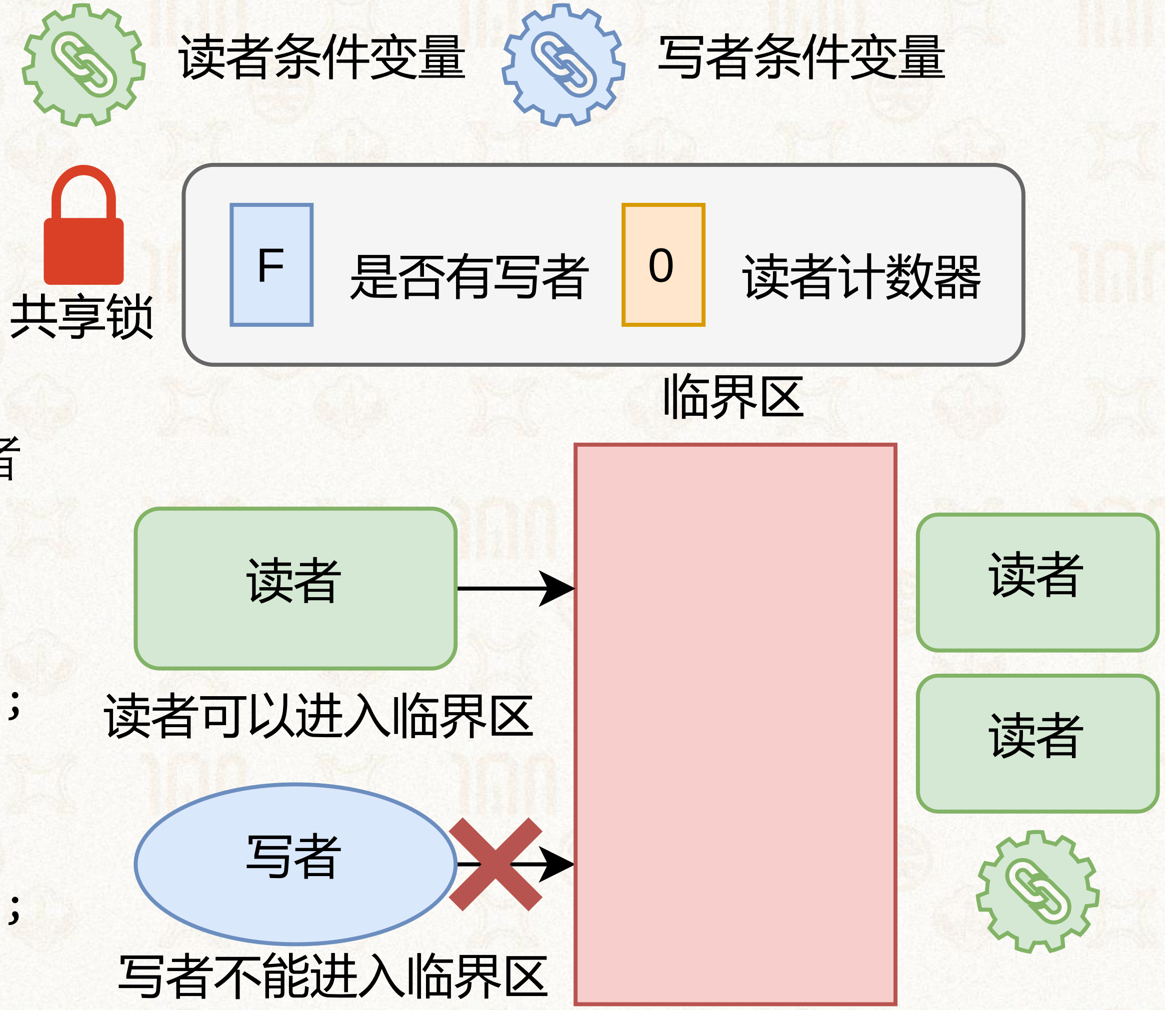
# 读写锁的实现：偏向写者为例

➢ 假设读者在临界区，但准备离开

```
void unlock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    rwlock->reader_cnt--;
    if(rwlock->reader_cnt == 0) {
        cond_signal(&rwlock->reader_cond);
    }
    unlock(&rwlock->lock);
}
```

最后一个读者离开，需要唤醒等待的写者

```
void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}
```



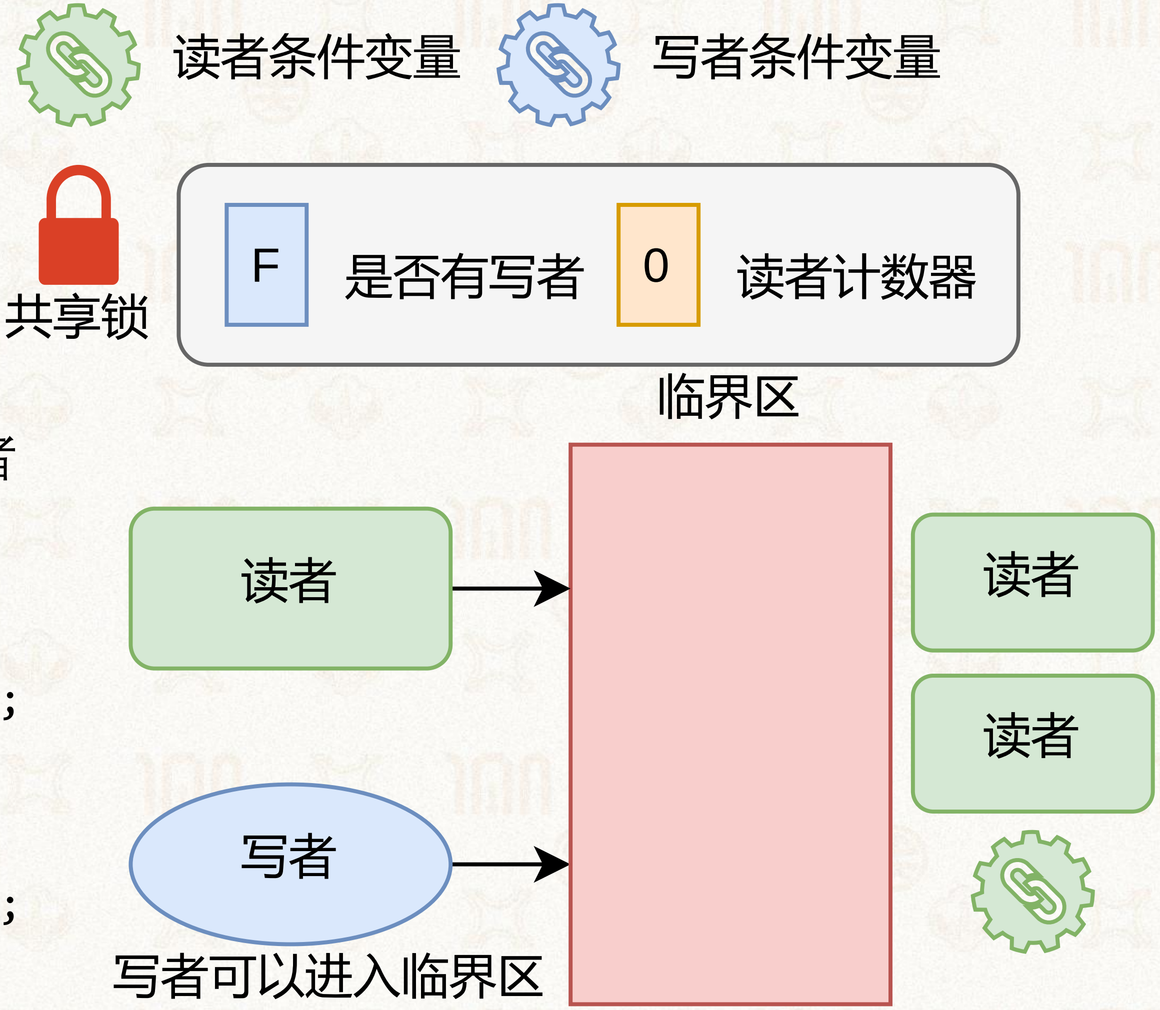
# 读写锁的实现：偏向写者为例

➢ 写者被唤醒，可以进入临界区

```
void unlock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    rwlock->reader_cnt--;
    if(rwlock->reader_cnt == 0) {
        cond_signal(&rwlock->reader_cond);
    }
    unlock(&rwlock->lock);
}
```

最后一个读者离开，需要唤醒等待的写者

```
void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}
```



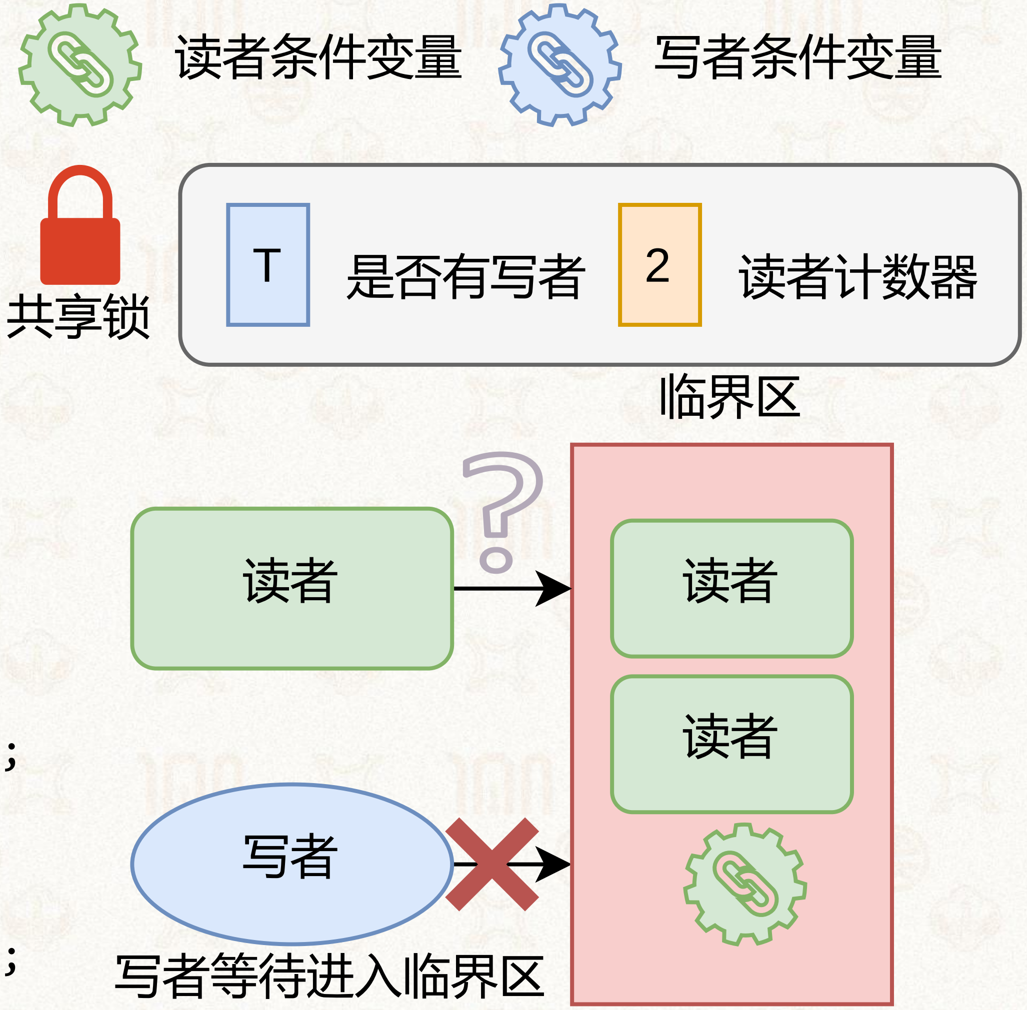
# 读写锁的实现：偏向写者为例

➢ 偏向写者体现在哪?

```
void lock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond,
                  &rwlock->lock);
    }
    rwlock->reader_cnt++;
    unlock(&rwlock->lock);
}
```

有写者在等待，新的读者能否再进?

```
void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}
```



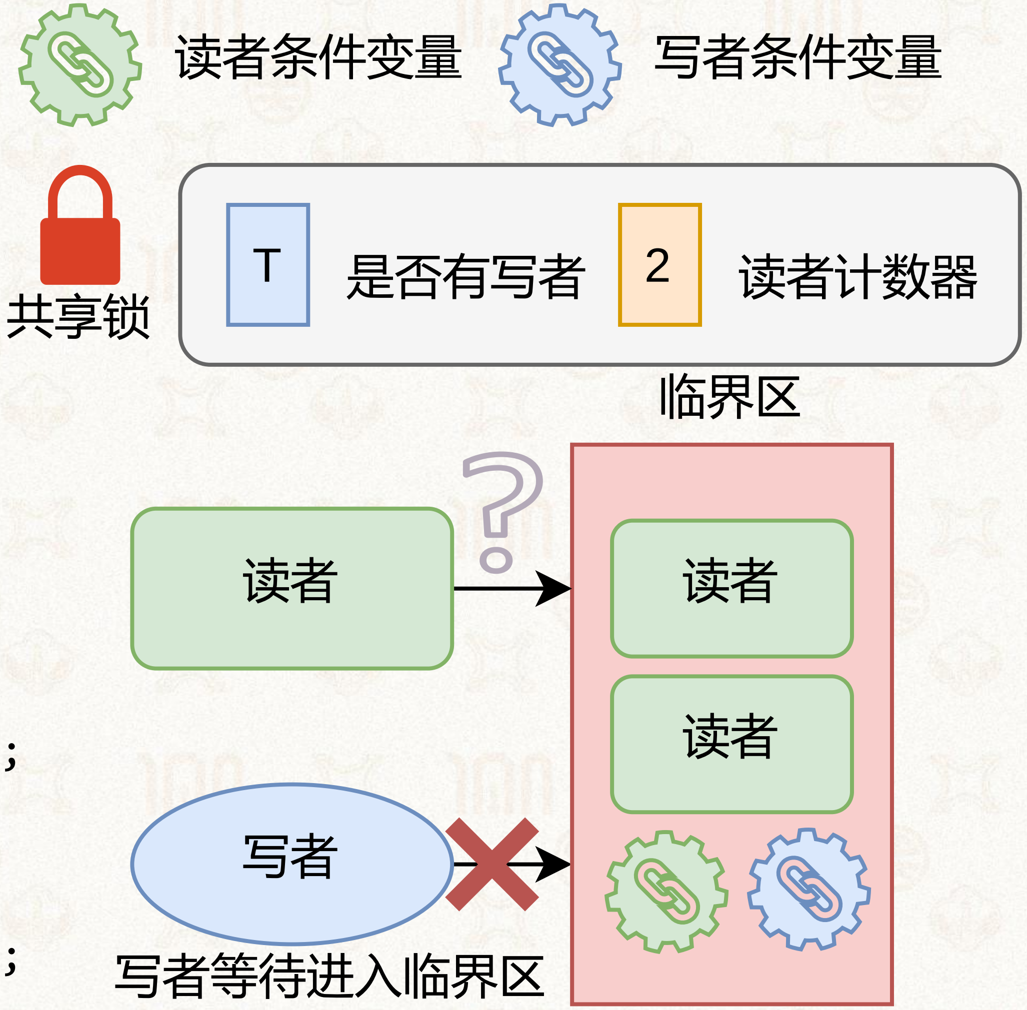
# 读写锁的实现：偏向写者为例

➢ 偏向写者体现在这里：

```
void lock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond,
                &rwlock->lock);
    }
    rwlock->reader_cnt++;
    unlock(&rwlock->lock);
}
```

有写者在等待，新的读者不能再进！

```
void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}
```



# 读写锁的实现：偏向写者为例

➢ 假设写者在临界区

```c
void lock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond,
                  &rwlock->lock);
    }
    rwlock->reader_cnt++;
    unlock(&rwlock->lock);
}
```

等待写者离开

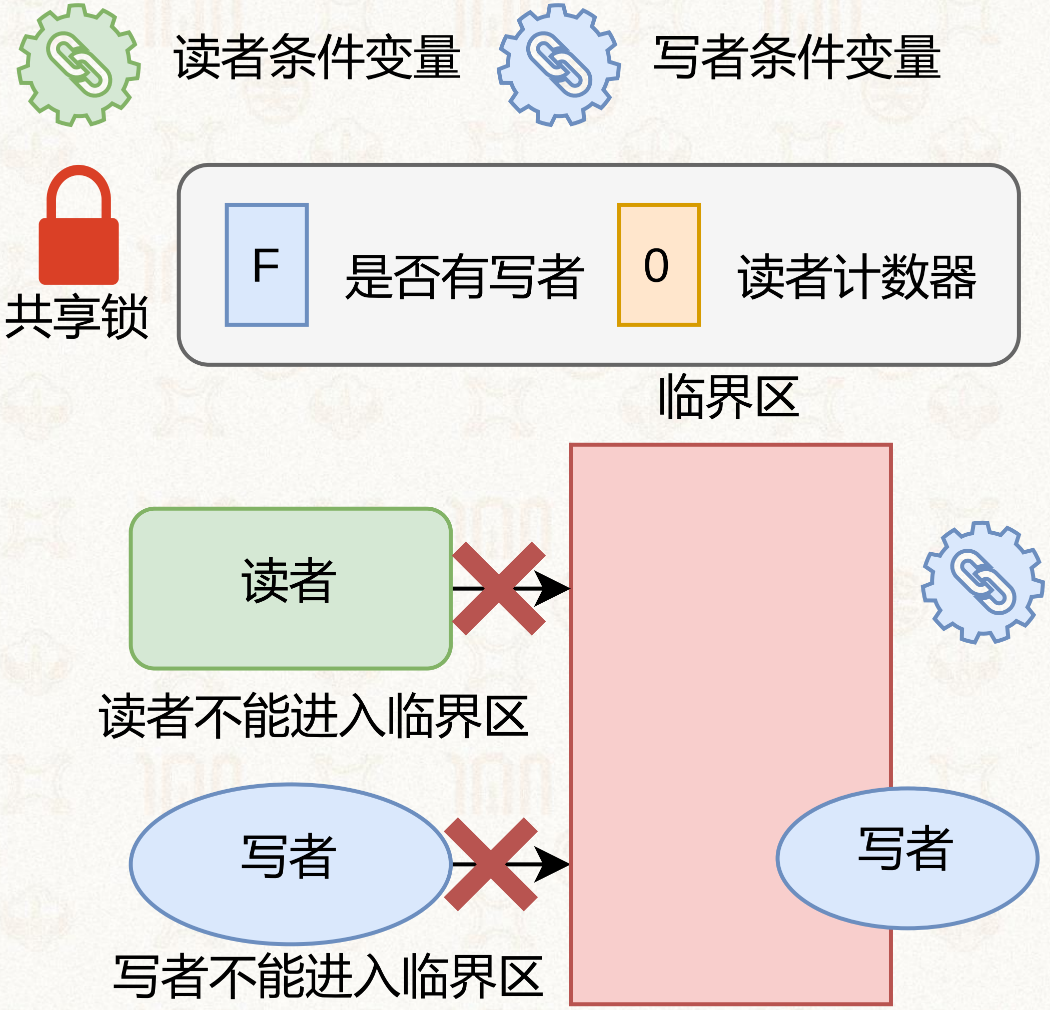
```c
void lock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond, &rwlock->lock);
    }
    rwlock->has_writer = TRUE;
    while(rwlock->reader_cnt > 0) {
        cond_wait(&rwlock->reader_cond, &rwlock->lock);
    }
    unlock(&rwlock->lock);
}
```

# 读写锁的实现：偏向写者为例

➤ 写者准备离开临界区

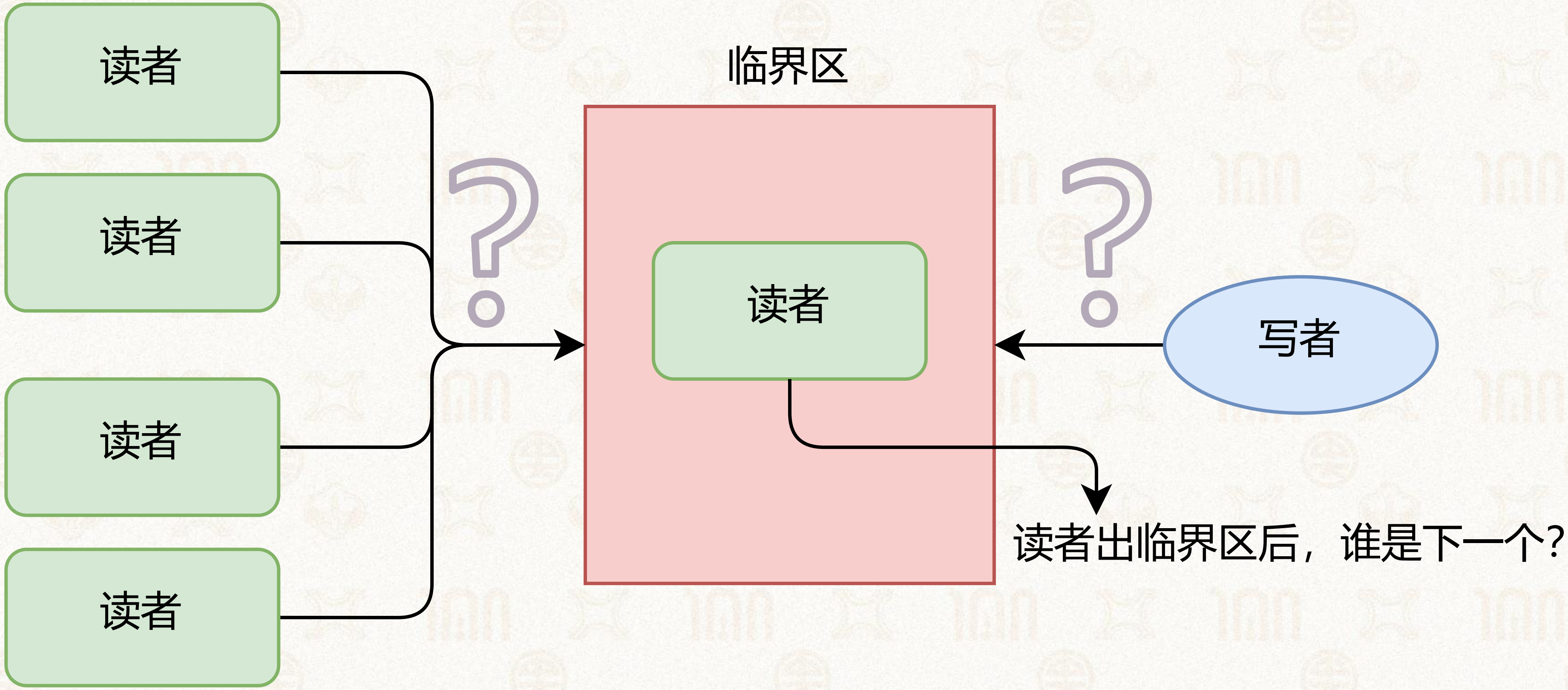
```
void lock_reader(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    while(rwlock->has_writer == TRUE) {
        cond_wait(&rwlock->writer_cond,
                  &rwlock->lock);
    }
    rwlock->reader_cnt++;
    unlock(&rwlock->lock);
}

void unlock_writer(struct rwlock *rwlock) {
    lock(&rwlock->lock);
    rwlock->has_writer = FALSE;
    cond_broadcast(&rwlock->writer_cond);
    unlock(&rwlock->lock);
}
```

通知所有等待写者条件变量的读者和写者



# 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？



# 大纲

- 同步问题的背景
  - 多核场景
  - 生产者消费者模型
  - 临界区问题
- 互斥锁
  - 皮特森算法
  - 原子操作
  - 互斥锁抽象
    - 自旋锁
    - 排号自旋锁
- 条件变量
- 信号量
  - PV原语
- 读写锁
- 同步原语产生的问题
  - 死锁
    - 银行家算法
  - 活锁
  - 优先级反转

# 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