

同步原语

上海交通大学

<https://www.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：进程间通信IPC

IPC的基本功能目标

- **数据传递**

- 发送消息：Send
- 接收消息：Recv
- 例如：pipe, msgqueue, 网络socket, 消息订阅发布publish/subscribe

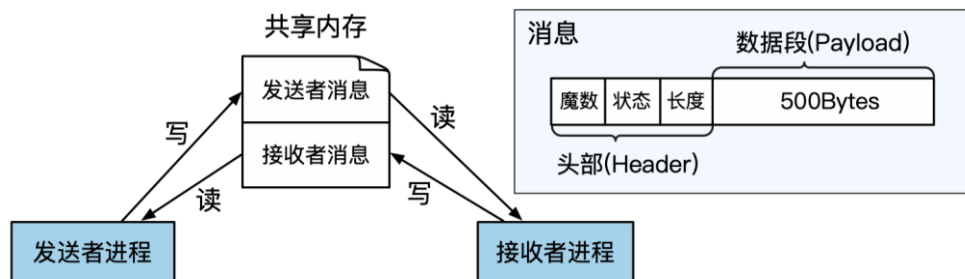
- **过程调用**

- 包括数据传递
- 远程方法调用：RPC
- 调用结果返回：Reply
- 例如：进程1调用进程2的方法（与远程过程调用RPC类似）

常见IPC的类型

IPC机制	数据抽象	参与者	方向
管道	文件接口	两个进程	单向
共享内存	内存接口	多进程	单向/双向
消息队列	消息接口	多进程	单向/双向
信号	信号接口	多进程	单向
套接字	文件接口	两个进程	单向/双向

简单IPC的两个阶段



- **阶段-1：准备阶段**

- 建立通信连接，即进程间的信道
 - 假设内核已经为两个进程映射了一段共享内存

- **阶段-2：通信阶段**

- 数据传递
 - “消息”抽象：通常包含头部（元数据）和数据内容（例如500字节）
- 通信机制
 - 两个消息保存在共享内存中：发送者消息、接收者消息
 - 发送者和接收者通过**轮询**消息的状态作为通知机制

示例：共享内存通信

- 基础实现：共享区域

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
...
```

```
} item;
```

```
item buffer[BUFFER_SIZE];
```

共享数据区域，容量为10

```
volatile int buffer_write_cnt = 0;
```

```
volatile int buffer_read_cnt = 0;
```

```
volatile int empty_slot = BUFFER_SIZE;
```

```
volatile int filled_slot = 0;
```

共享状态

volatile关键字的目的？

基于共享内存的生产者消费者问题实现

- 基础实现: 发送者 (生产者)

```
while (true) {  
    /* Produce an item/msg */
```

当没有空间时，发送者盲等

```
    while (empty_slot == 0)  
        ; /* do nothing -- no free buffers */
```

```
    empty_slot --;
```

```
    buffer[buffer_write_cnt] = msg;
```

发送者放置消息

```
    buffer_write_cnt = (buffer_write_cnt + 1) % BUFFER_SIZE;
```

```
    filled_slot ++;
```

```
    ...
```

```
}
```


基于共享内存的生产者消费者问题实现

- 基础实现: 接收者

当没有新消息时, 接收者盲目等待

```
while (true) {
```

```
    while (filled_slot == 0)
```

```
        ; // do nothing -- nothing to consume
```

```
    filled_slot--; // remove an item from the buffer
```

```
    item = buffer[buffer_read_cnt];
```

----- 接收者获取消息

```
    buffer_read_cnt = (buffer_read_cnt + 1) % BUFFER_SIZE;
```

```
    empty_slot++;
```

```
    return item;
```

```
}
```

简单IPC数据传递的两种方法

- **方法-1：通过共享内存的数据传递**
 - 操作系统在通信过程中不干预数据传输
 - 操作系统仅负责准备阶段的映射
- **方法-2：通过操作系统内核的数据传递**
 - 操作系统提供接口（系统调用）
 - 通过内核态内存来传递数据，无需在用户态建立共享内存

两种数据传递方法的对比

- **基于共享内存的优势**

- 完全由用户态控制，定制能力更强
- 无需内核进行额外的内存拷贝

思考：如何避免TOCTOU？

思考：如何避免内存拷贝？

- **基于系统调用的优势**

- 抽象更简单，用户态直接调用接口，使用更方便
- 安全性保证更强，发送者在消息被接收时通常无法修改消息

简单IPC的通知机制

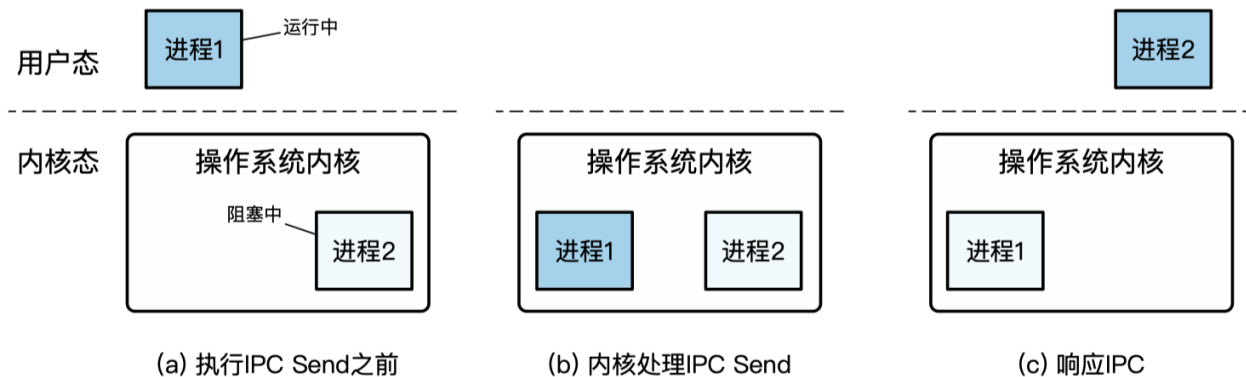
- **方法-1：基于轮询（消息头部的状态信息）**

- 缺点：大量CPU计算资源的浪费

- **方法-2：基于控制流转移**

思考：相比于方法2，方法-1有什么优势？

- 由内核控制进程的运行状态
- 优点：进程只有在条件满足的情况下才运行，避免CPU浪费



大纲

- 多线程问题：竞争条件
- 四种同步原语
 - 互斥锁
 - 条件变量
 - 信号量
 - 读写锁
- 死锁问题

思考：多个生产者会怎么样？

- 基础实现：发送者 (生产者)

```
while (true) {  
    /* Produce an item/msg */
```

当没有空间时，发送者盲等

```
    while (empty_slot == 0)  
        ; /* do nothing -- no free buffers */
```

```
    empty_slot --;
```

```
    buffer[buffer_write_cnt] = msg;
```

发送者放置消息

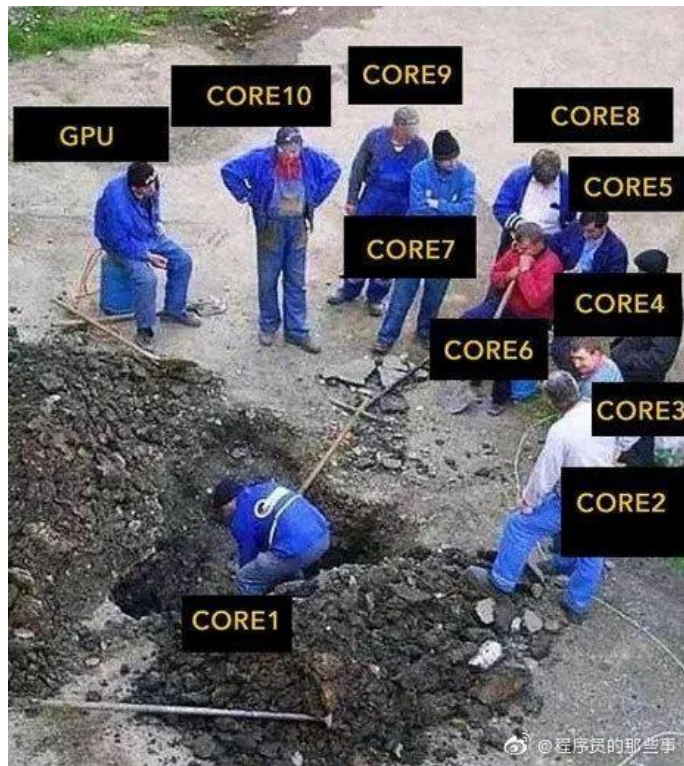
```
    buffer_write_cnt = (buffer_write_cnt + 1) % BUFFER_SIZE;
```

```
    filled_slot ++;
```

```
    ...
```

```
}
```

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）

▶ **并发带来的同步问题：竞争条件**

多线程计数实例

注意：多个**进程**操作**共享内存**中的变量，同样存在该问题

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++) {
        a++;
    }
    return NULL;
}
```

3个线程同时执行

输出结果是多少？

理论上的结果： $1000000000 * 3 = 3000000000$

实际在Intel 10代6核i7中： 1040186238 （结果不唯一）

多线程计数中的数据竞争

线程1

a++;

线程2

a++;

a的初始值为3, 结果应当为5

可以看汇编代码

T0 reg_a = a; (3)

T1 reg_a = reg_a + 1;

reg_a = a; (3)

T2 a = reg_a; (4)

reg_a = reg_a + 1;

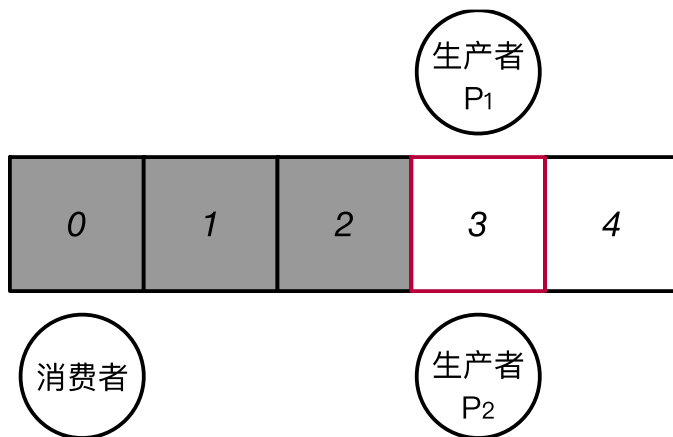
a = reg_a; (4)

最后结果a为4: 线程1的"+1"操作丢失了

生产者消费者编程

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**（又称竞争冒险、竞态条件）：

- 当2个或以上线程同时对共享的数据进行操作，其中至少有一个写操作
- 该共享数据最后的结果**依赖于这些线程特定的执行顺序**

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;  
}
```

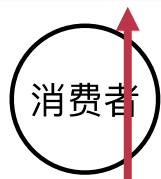
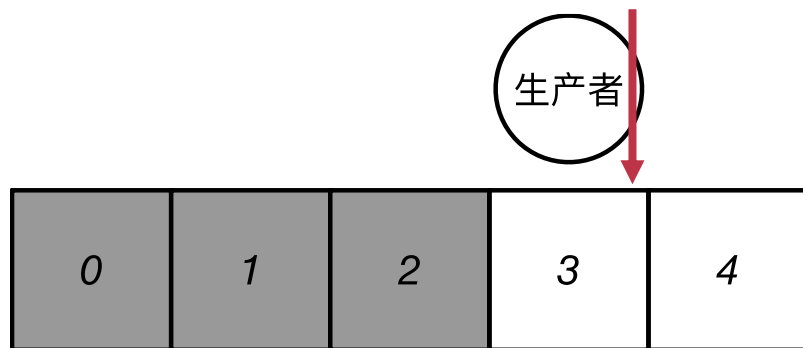
生产者消费者问题的基础实现

- 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

$\text{prodCnt} \% \text{BUFFER_SIZE}$



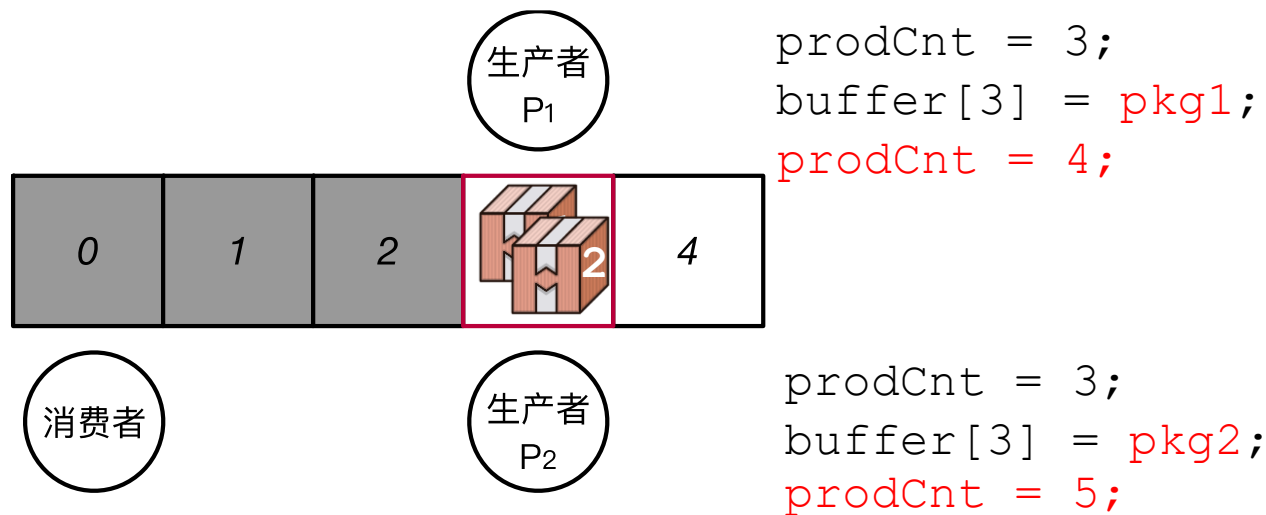
共享缓冲区

$\text{consCnt} \% \text{BUFFER_SIZE}$

通过两个计数器来协调
单个生产者与单个消费者

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;*
```



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**？

编程抽象：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

通过同步原语实现临界区抽象

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

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

```
begin:
lock(&buffer_lock); // 申请进入临界区
while (prodCnt - consCnt == BUFFER_SIZE)
    unlock(&buffer_lock);
    goto begin; /* do nothing -- no free buffers */
```

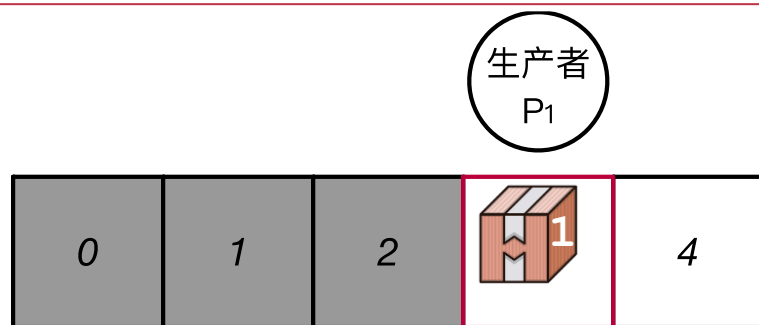
```
buffer[prodCnt % BUFFER_SIZE] = item;
prodCnt = prodCnt + 1;
```

临界区

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

用互斥锁解决多生产者消费者问题

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



生产者
P₁

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

获取互斥锁
进入临界区

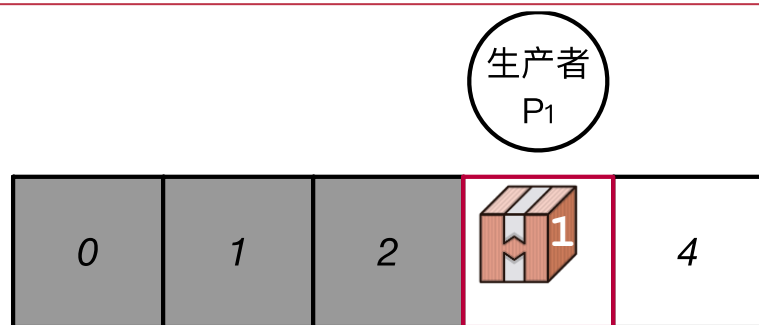
消费者

```
(prodCnt = 3)  
lock(&buffer_lock); ...;
```

没有获取互斥锁,
在原地等待

用互斥锁解决多生产者消费者问题

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



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

获取互斥锁
进入临界区

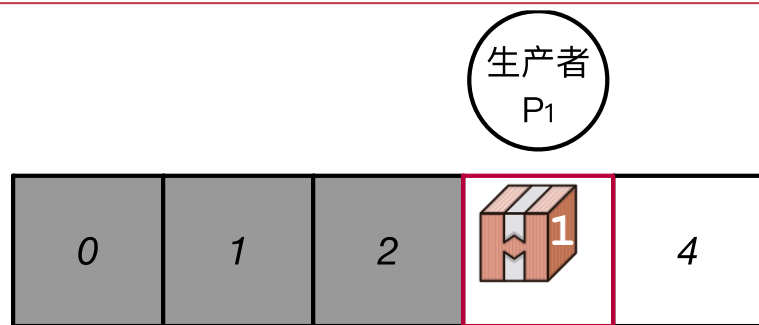


```
(prodCnt = 4)  
lock(&buffer_lock); ...;
```

没有获取互斥锁,
在原地等待

用互斥锁解决多生产者消费者问题

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



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



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

获取互斥锁
进入临界区

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

思考题：这样改写代码正确吗？

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

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

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

临界区

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

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

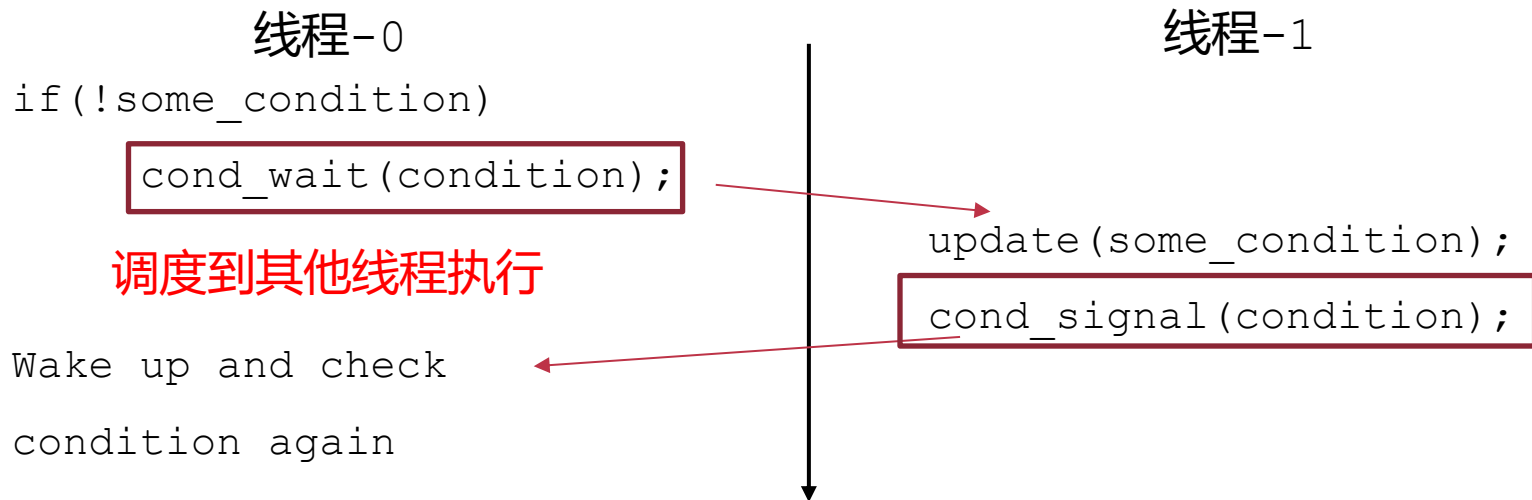
条件变量：利用睡眠/唤醒机制，避免无意义的等待

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. ...
```

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**：调度器可以调度到其他线程
3. 被唤醒后重新**获取锁**

思考：为什么阻塞和放锁两个操作需要由OS完成？

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

思考：为什么阻塞和放锁两个操作需要由OS完成？

若5-6行之间收到信号通知，
则存在信号丢失问题！

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. ...
```

```
1. ...
2. /* Wait empty slot */
3. lock(empty_cnt_lock);
4. while (empty_slot == 0)
5.     unlock(empty_cnt_lock);
6.     cond_wait_nolock(empty_cond);
7.     lock(empty_cnt_lock);
8. empty_slot--;
9. unlock(empty_cnt_lock);
10. ....
```


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)  
    时刻1 cond_wait(empty_cond,  
                    empty_cnt_lock);
```

时刻4 错误唤醒!

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

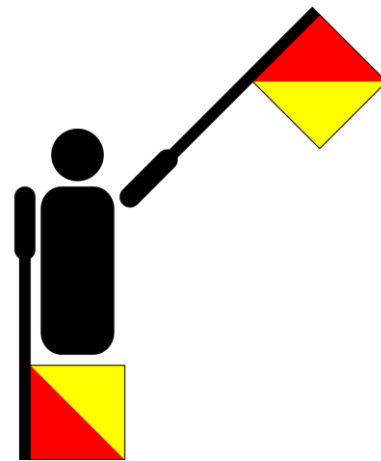
线程 2

时刻2

有新的空位, 唤醒
empty_slot = 1

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

时刻3 重新拿到锁



信号量 (SEMAPHORE)

生产者消费者问题的当前实现

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
有没有可以自动管理资源数量的同步原语？

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`cnt`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

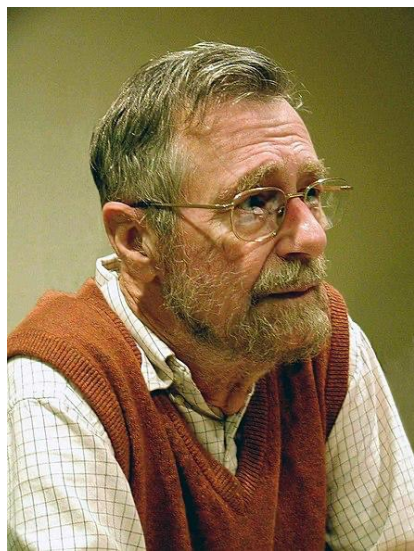
```
void sem_wait(sem_t *sem) {  
    while(sem->cnt <= 0)  
        /* Waiting */;  
    sem->cnt--;  
}
```

`cnt`代表剩余资源数量

V操作: 增加资源

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

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_cnt) {  
    sem->cnt = init_cnt;  
}
```



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

读写锁

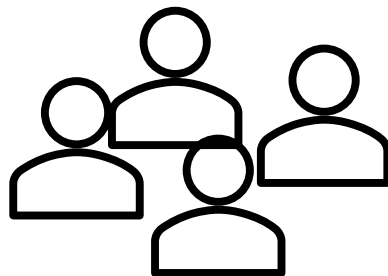
公告栏问题



写者



公告栏



读者



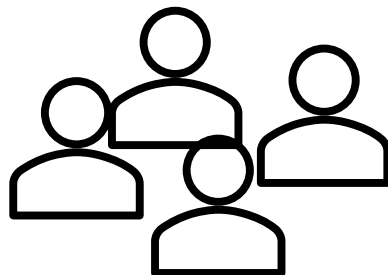
思考：多个读者如果希望读公告栏，他们互斥吗？

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？

可以使用互斥锁，但读者就不能一起看了

读写锁的使用示例

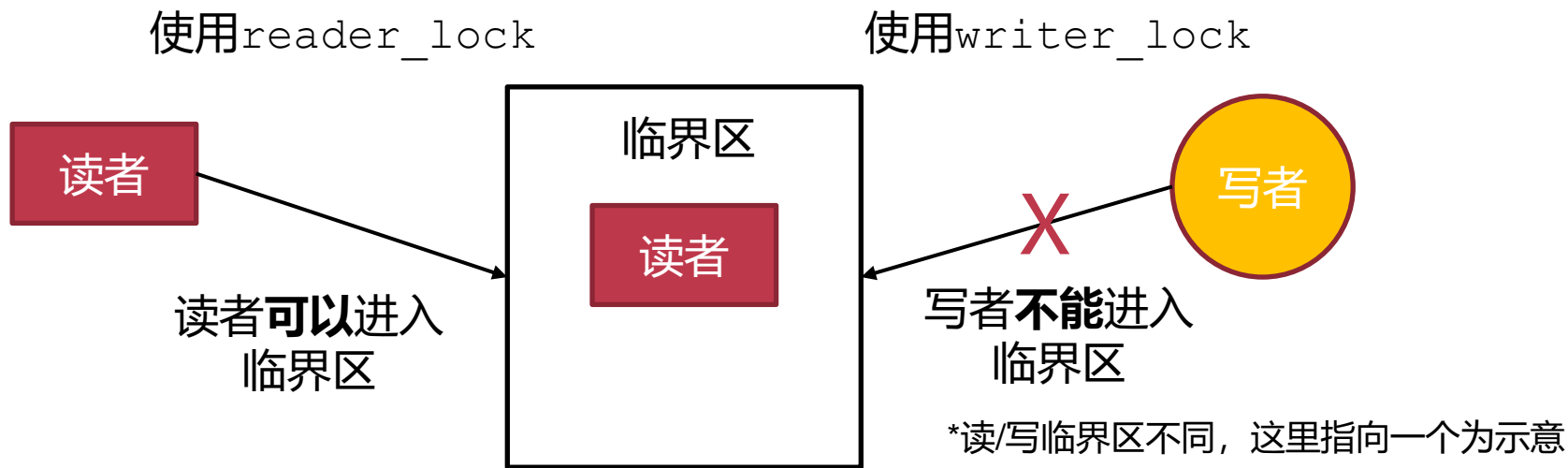
```
struct rwlock lock;  
char data[SIZE];  
  
void reader(void)  
{  
    lock_reader(&lock);  
    read_data(data); // 读临界区  
    unlock_reader(&lock);  
}  
  
void writer(void)  
{  
    lock_writer(&lock);  
    update_data(data); // 写临界区  
    unlock_writer(&lock);  
}
```

读写锁

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

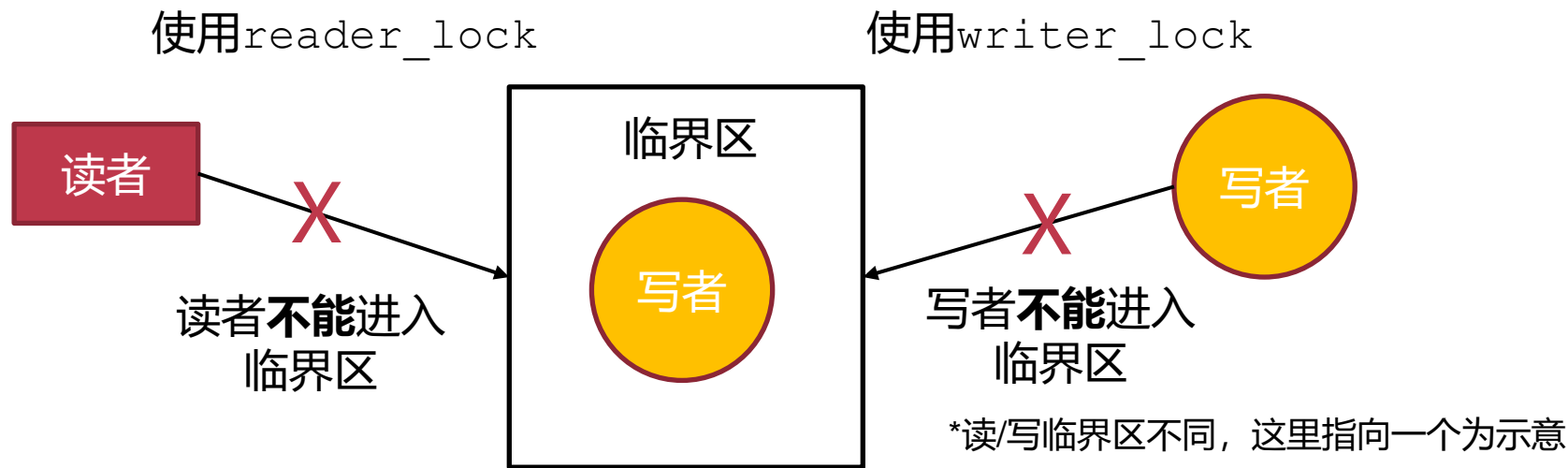


读写锁

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**



小结：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);`

`unlock(&lock0);`

Thread 0

`sem_wait(&s0);`

Thread 1

另一个线程

`sem_signal(&s0);`

同一线程

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**用于实现“有条件地”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

以锁为例看同步原语的实现

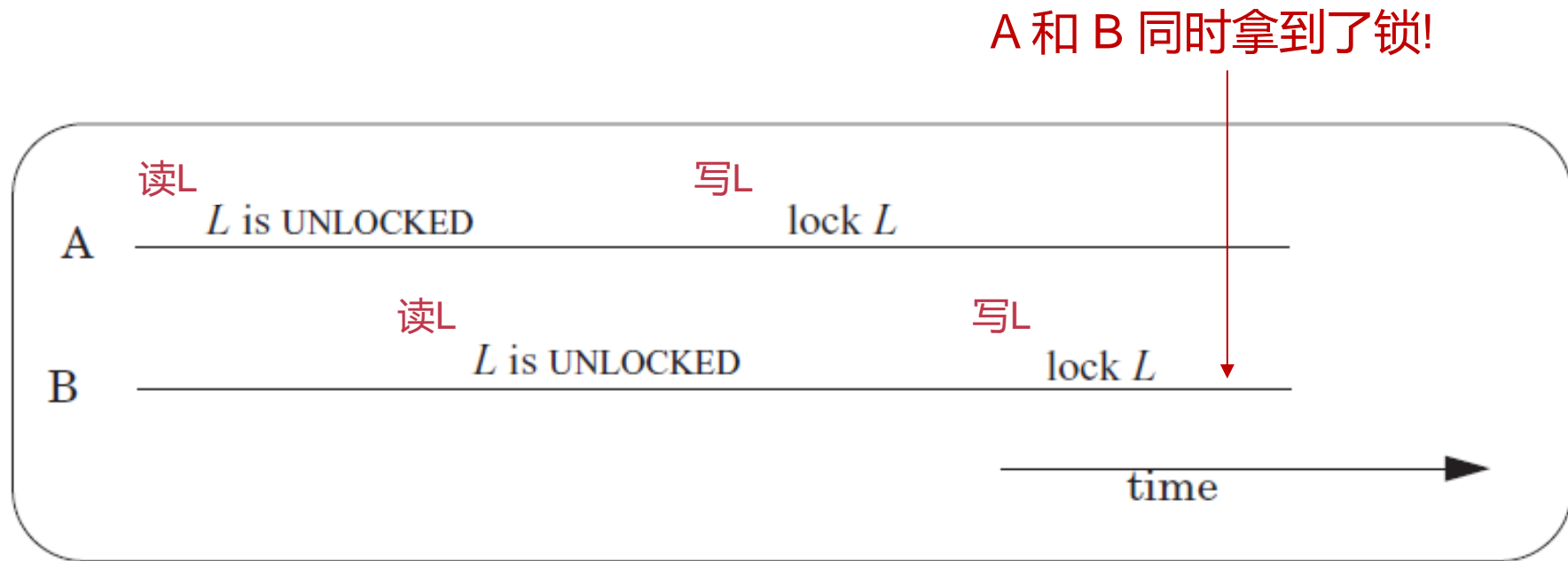
锁的实现

Lock: 一种直观但错误的实现

```
struct lock { int state; };
```

```
void lock (lock *L) {  
    while L->state == LOCKED  
        ; // spin until L is UNLOCKED  
    L->state = LOCKED; // the while test failed, got the lock  
}  
  
void unlock (lock *L) {  
    L->state = UNLOCKED;  
}
```

竞争条件依然存在!



操作-1: 读L, 检查状态是否为LOCKED

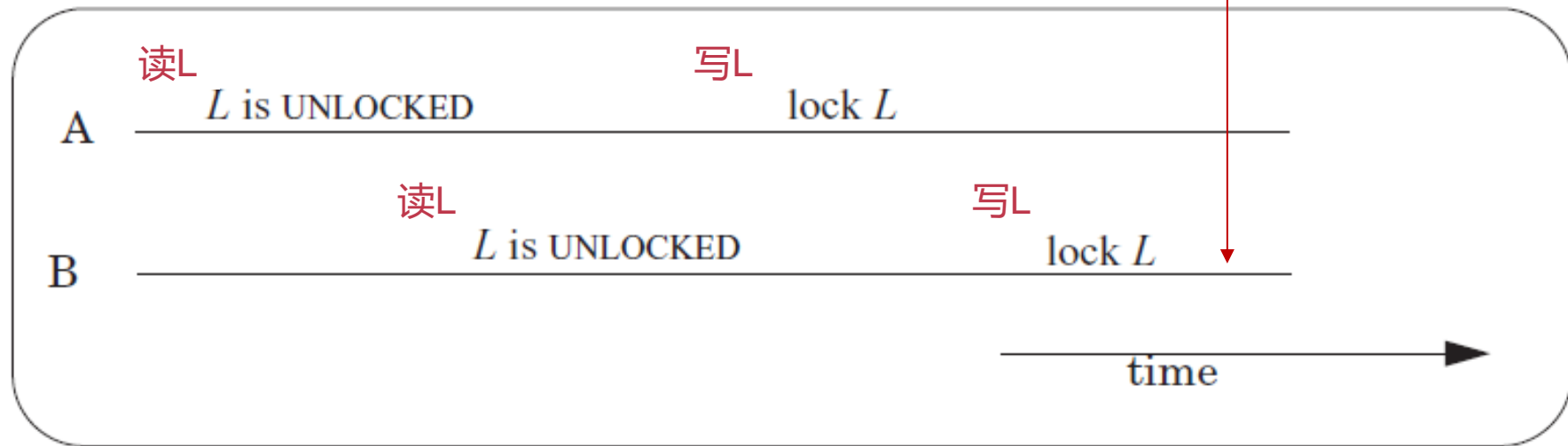
操作-2: 写L, 将其状态设置为LOCKED

这两步并非原子完成!

需要用另一个lock来保证原子性?

互斥锁实现：基于硬件原子指令

A 和 B 同时拿到了锁!



操作-1: 读L, 检查状态是否为LOCKED

操作-2: 写L, 将其状态设置为LOCKED

根本原因: 这两步并非原子完成!

硬件方法: 用原子指令来保证两步是原子的!

新指令-1: Test-and-Set

- 历史

- 1960年代初期, Burroughs B5000首先引入

```
1 int TestAndSet(int *old_ptr, int new) {  
2     int old = *old_ptr; // fetch old value at old_ptr  
3     *old_ptr = new; // store 'new' into old_ptr  
4     return old; // return the old value  
5 }
```

注意: TestAndSet仅为单条指令, C代码仅用于表示语义

使用 Test-and-Set 实现 Spin Lock

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

新指令-2: Compare-and-swap

- 另一个原子的硬件原语
 - Compare-and-swap (on SPARC)
 - Compare-and-exchange (on x86)

```
1 int CompareAndSwap(int *ptr, int expected, int new) {  
2     int actual = *ptr;  
3     if (actual == expected)  
4         *ptr = new;  
5     return actual;  
6 }
```

注意: C代码仅用于表示语义

用 Compare-and-swap 实现 Spin Lock

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (CompareAndSwap(&lock->flag, 0, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

新指令-3: Load-linked & Store-conditional

ARM架构

```
1 int LoadLinked(int *ptr) {  
2     return *ptr;  
3 }  
4  
5 int StoreConditional(int *ptr, int value) {  
6     if (no one has updated *ptr since the LoadLinked to this address) {  
7         *ptr = value;  
8         return 1; // success!  
9     } else {  
10        return 0; // failed to update  
11    }  
12 }
```

注意: C代码仅用于表示语义

用 LL/SC 来实现 Spinlock

```
1 void lock(lock_t *lock) {
2     while (1) {
3         while (LoadLinked(&lock->flag) == 1)
4             ; // spin until it's zero
5         if (StoreConditional(&lock->flag, 1) == 1)
6             return; // if set-it-to-1 was a success: all done
7                     // otherwise: try it all over again
8     }
9 }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

新指令-4: Fetch-and-add

```
1 int FetchAndAdd(int *ptr) {  
2     int old = *ptr;  
3     *ptr = old + 1;  
4     return old;  
5 }
```

注意: C代码仅用于表示语义

用 Fetch-and-add 实现 Ticket Lock

```
1 typedef struct __lock_t {
2     int ticket;
3     int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7     lock->ticket = 0;
8     lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15 }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```



和Spin Lock相比, Ticket Lock具有公平性

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
```



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

```
owner = 3  
next = 7
```

2. 等待叫号

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

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...



2. 叫下个人进来

```
owner += 1
```

1. 吃完了，买单



```
owner = 4
```

```
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

unlock操作

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

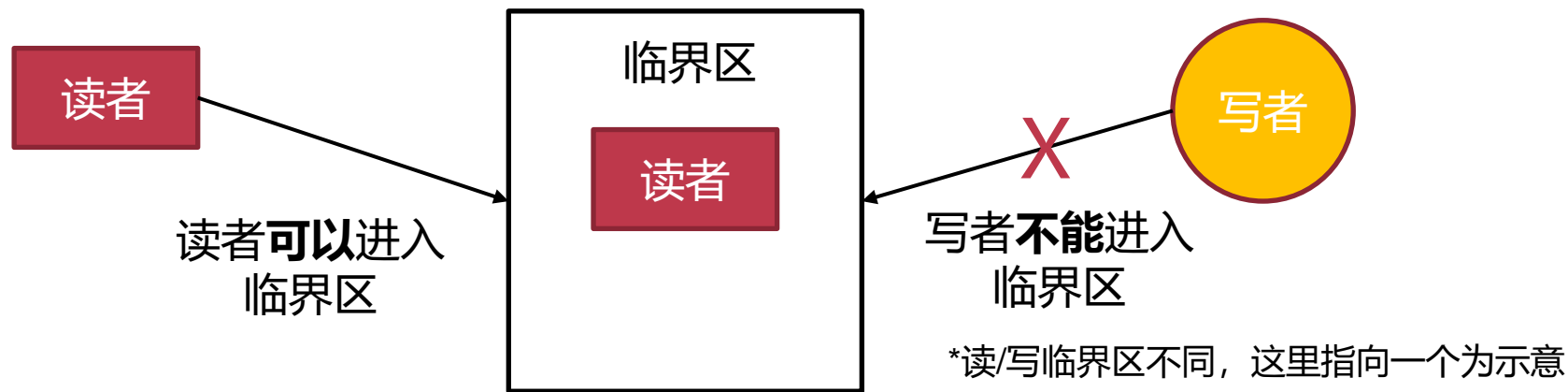
读写锁的实现

回顾：读写锁的基础语义

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

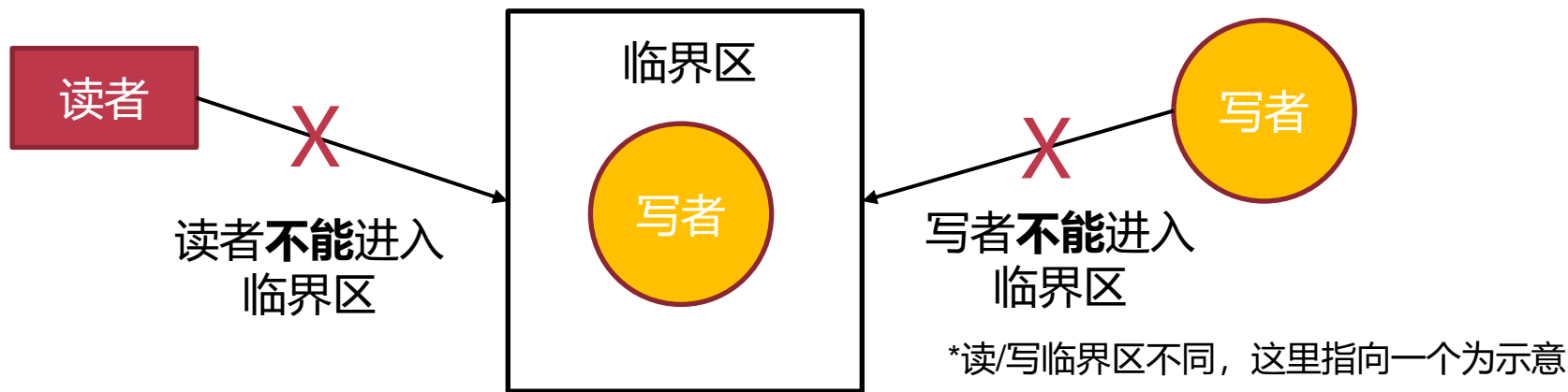


回顾：读写锁的基础语义

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：**

- 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

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

读者




临界区



写者锁



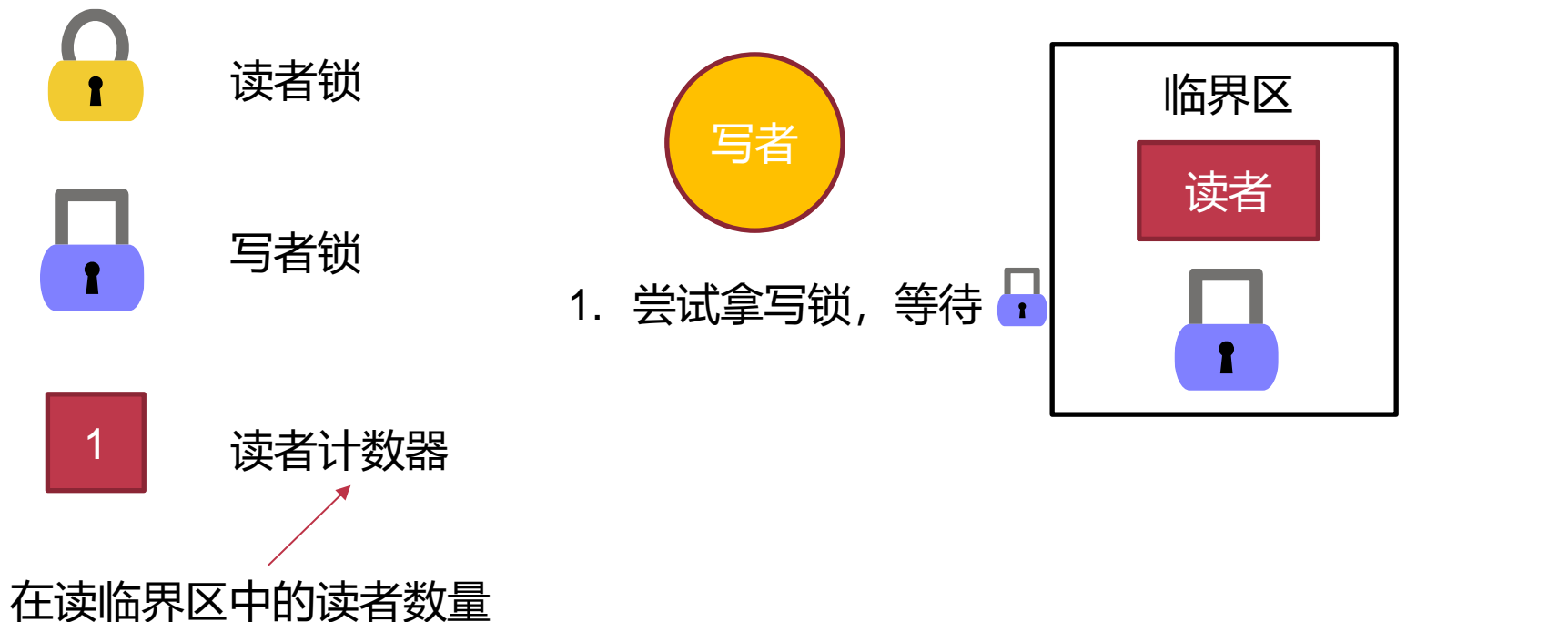
读者计数器

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

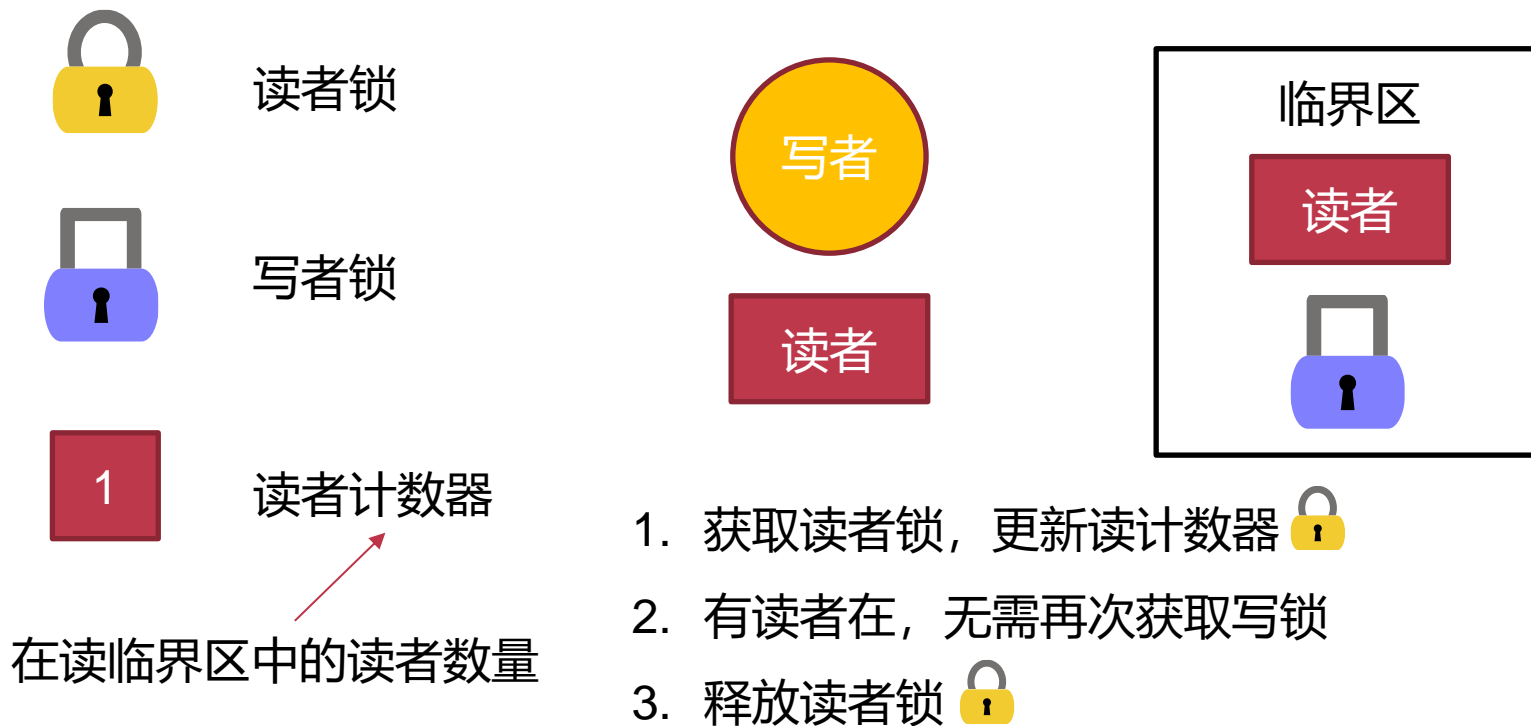
*读/写临界区不同，这里指向一个为示意

读写锁的实现：偏向读者为例



*读/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/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



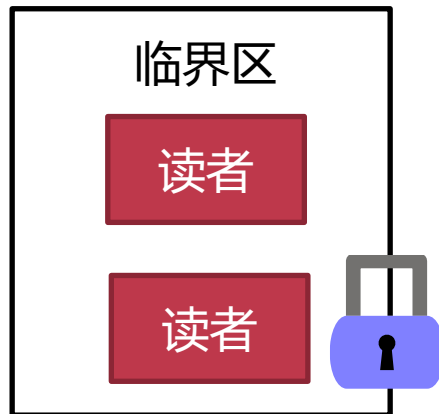
读者锁



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

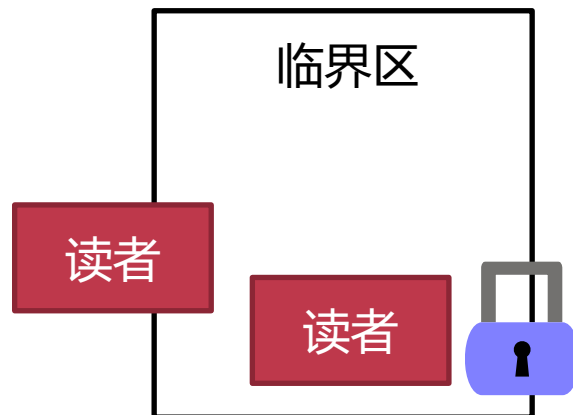




写者锁



读者计数器

在读临界区中的读者数量

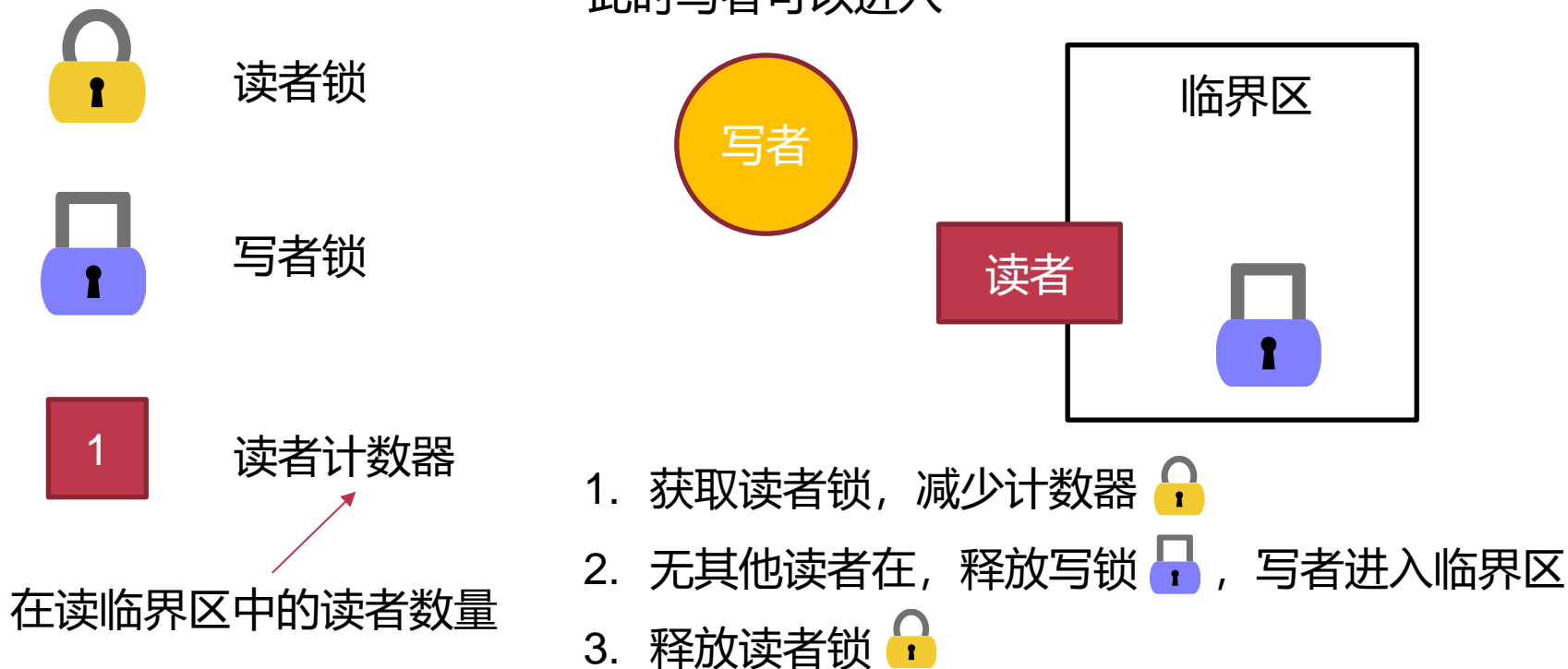


1. 获取读者锁，减少计数器 
2. 还有其他读者在，无需释放写锁
3. 释放读者锁 

*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

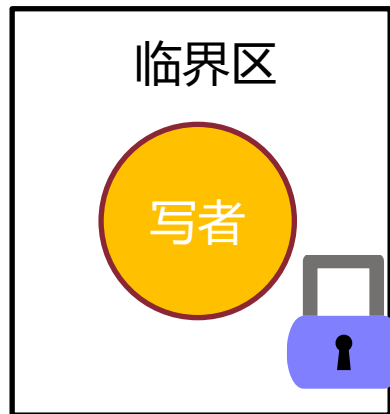


读者计数器

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



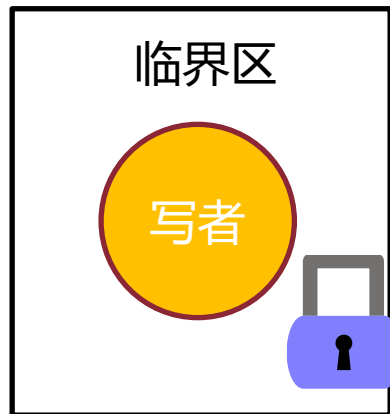
读者锁




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

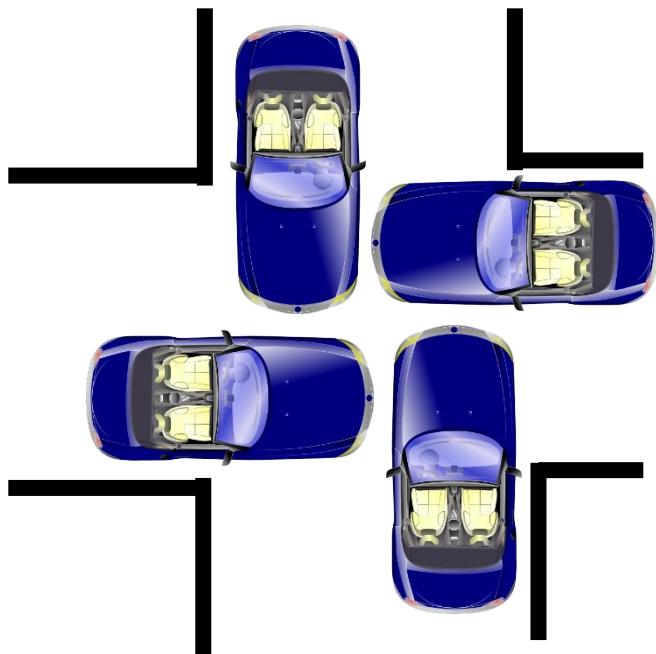
注意：读者锁还有阻塞其他读者的语义，因此不能用原子操作来替代

*读/写临界区不同，这里指向一个为示意



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

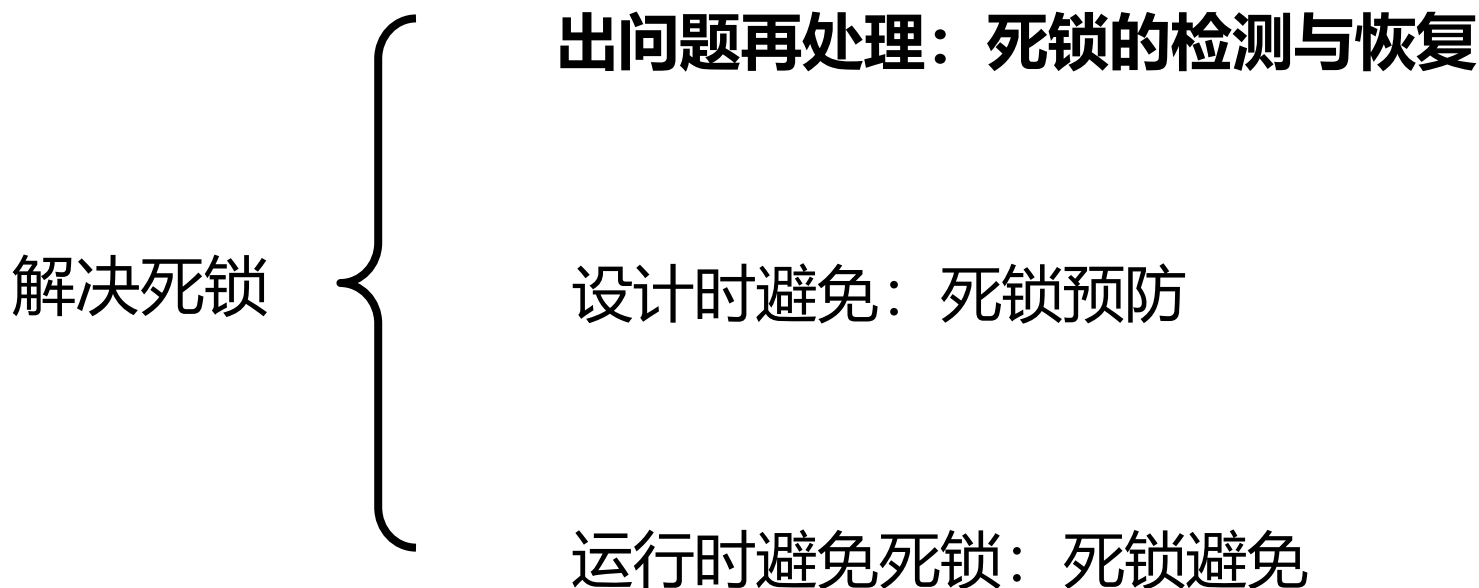
- 互斥访问
- 持有并等待
- 资源非抢占
- 循环等待

proc_A等proc_B, proc_B等proc_A

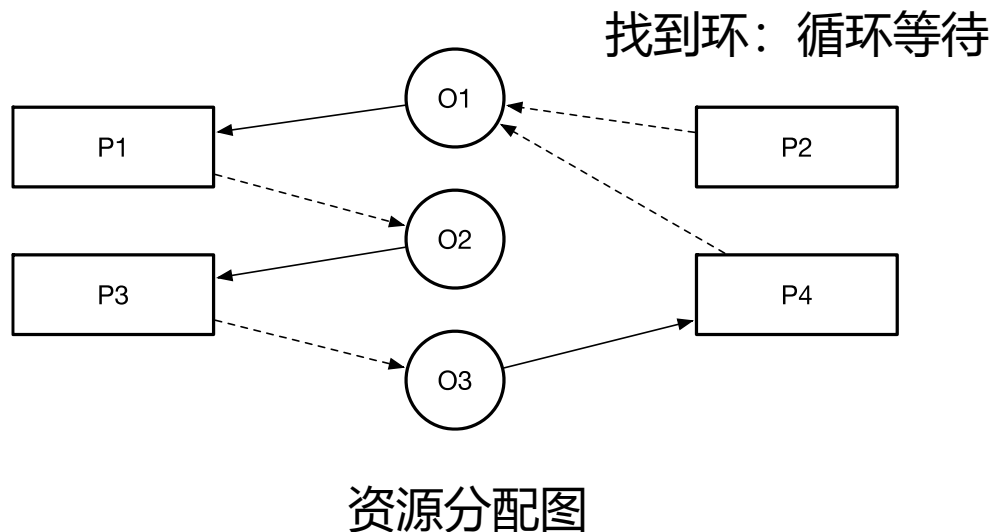
```
void proc_A(void) {  
    lock(A);  
    /* Time T1 */  
    lock(B);  
    /* Critical Section */  
    unlock(B);  
    unlock(A);  
}  
  
void proc_B(void) {  
    lock(B);  
    /* Time T1 */  
    lock(A);  
    /* Critical Section */  
    unlock(A);  
    unlock(B);  
}
```

T1时刻的死锁

如何解决死锁?



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

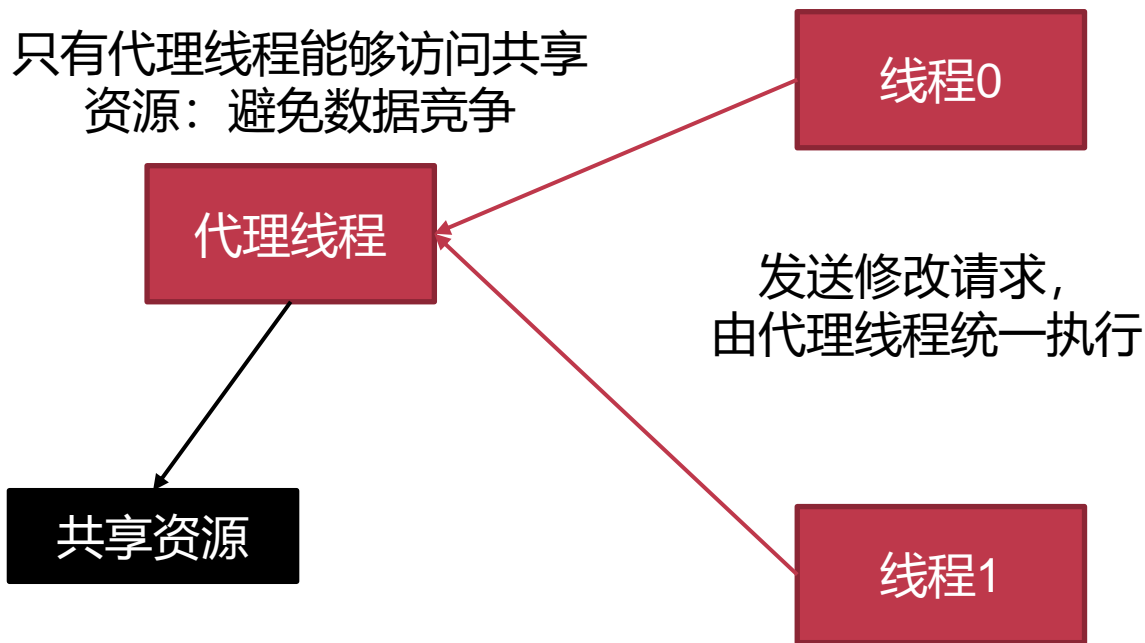
设计时避免：死锁预防

运行时避免死锁：死锁避免

死锁预防：方法一

死锁条件：互斥访问

避免互斥访问：通过其他手段（如代理执行）



*代理锁 (Delegation Lock) 实现了该功能

死锁预防：方法二

死锁条件：持有并等待

不允许持有锁并等待拿其他锁：若拿不到某把锁，则把已获取的锁也放了

```
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

死锁预防：方法三

死锁条件：循环等待

打破循环等待：按照特定顺序获取锁

- 对锁进行编号
- 让所有线程递增获取

A: 1号 B: 2号: 必须先拿锁A, 再拿锁B

```
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);  
}
```

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

死锁预防：方法四

死锁条件：资源非抢占

允许资源抢占：需要考虑如何恢复

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

抢占锁A

需要让执行proc_A的线程
回滚到拿锁A之前的状态

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

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 (P2持有的资源释放)

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

小结

- **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
- **锁的实现**
 - 自旋锁
 - 排号锁
 - 读写锁
- **死锁问题与预防**



回顾



同步原语的不同应用场景

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;
}
```

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

回顾：多线程做累加操作

使用**互斥锁**保证**互斥访问**

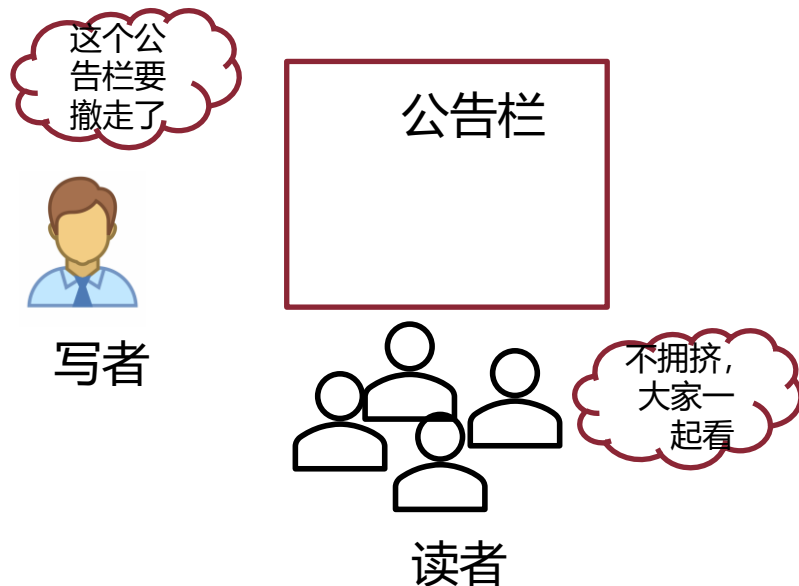
衍生场景一：读写场景并发读取

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++;  
}
```

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

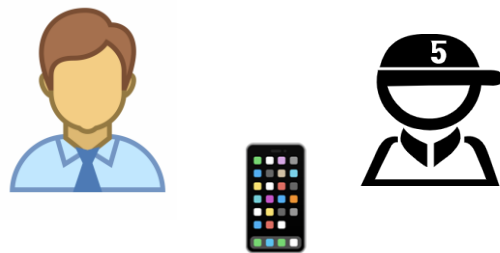
线程等待某条件时**睡眠**

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



举例：快递员 (thread_2)等取件电话，
客户 (thread_1)打包后通知快递员。

场景三：多资源协调管理

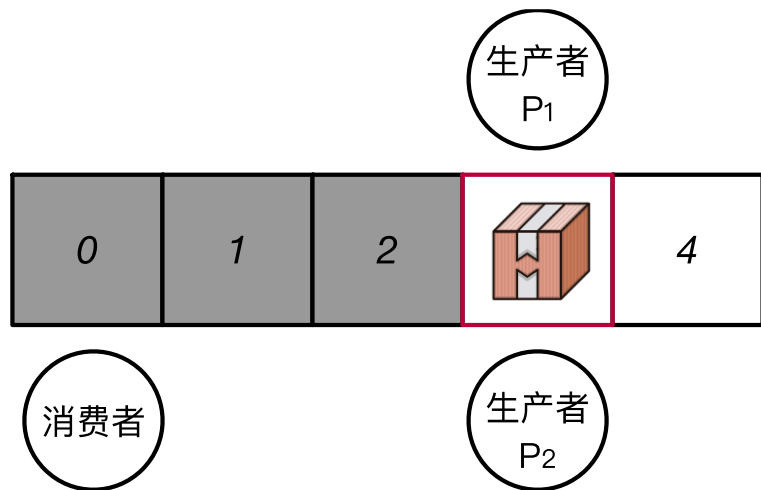
多个资源可以被多个线程**消耗或释放**

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

场景与同步原语总结

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

互斥锁 vs 读写锁

- **读写锁为特定场景（衍生场景一）下提升读者并行度**
- **衍生场景一直接使用互斥锁**
 - 也可以保证正确性
 - 读者之间不能并发执行
- **特定场景下的性能优化**

互斥锁 vs 条件变量

- **面向不同场景（正交）**
 - 互斥锁：互斥访问
 - 条件变量：条件等待与唤醒
- **条件变量通常需要搭配互斥锁使用**
 - 互斥锁中也可以使用条件变量避免循环等待

互斥锁 vs 信号量

- 互斥锁与二元信号量

- 功能基本一致，二元信号量可以实现互斥
- 语义差别：二元信号量可以由**不同的线程**获取/释放。互斥锁语义上只能由**同一个线程**获取与释放
- 保护资源互斥场景，推荐使用互斥锁（方便优化）

- 互斥锁与计数信号量

- 应对不同场景
 - 互斥锁控制对唯一资源的互斥访问（即临界区）
 - 计数信号量控制多个线程对多个资源的获取与释放

条件变量 vs 信号量

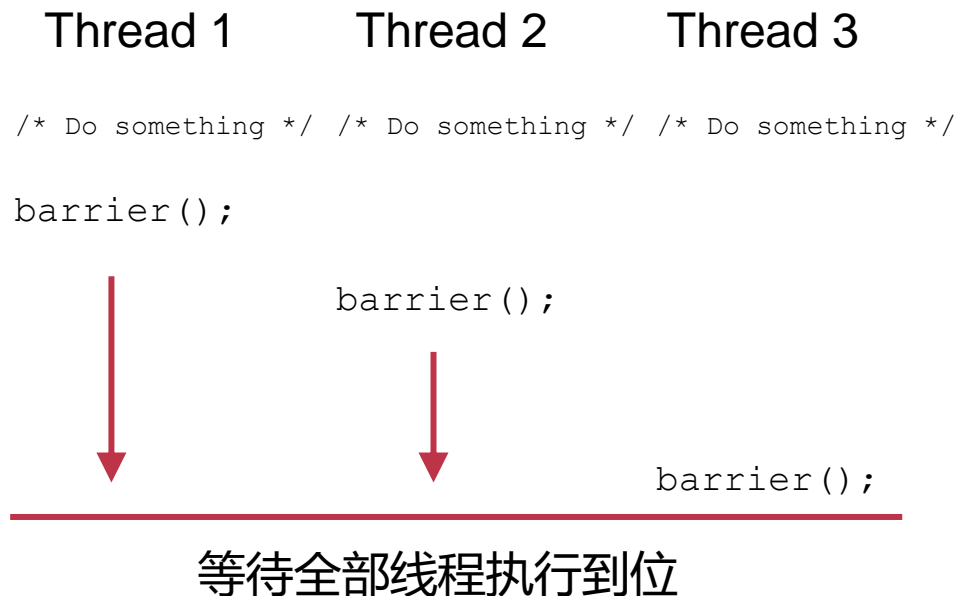
- 提供了类似的接口
- 抽象层级的区别
 - 条件变量
 - 更底层，提供睡眠唤醒机制
 - 适用范围更广
 - 信号量
 - 针对具体的场景：提供对**有限资源**的管理
 - 可以使用**条件变量+互斥锁+计数器**实现信号量



案例分析

同步案例-1：多线程执行屏障

- 多线程执行屏障
- 等待全部执行到屏障后再继续执行

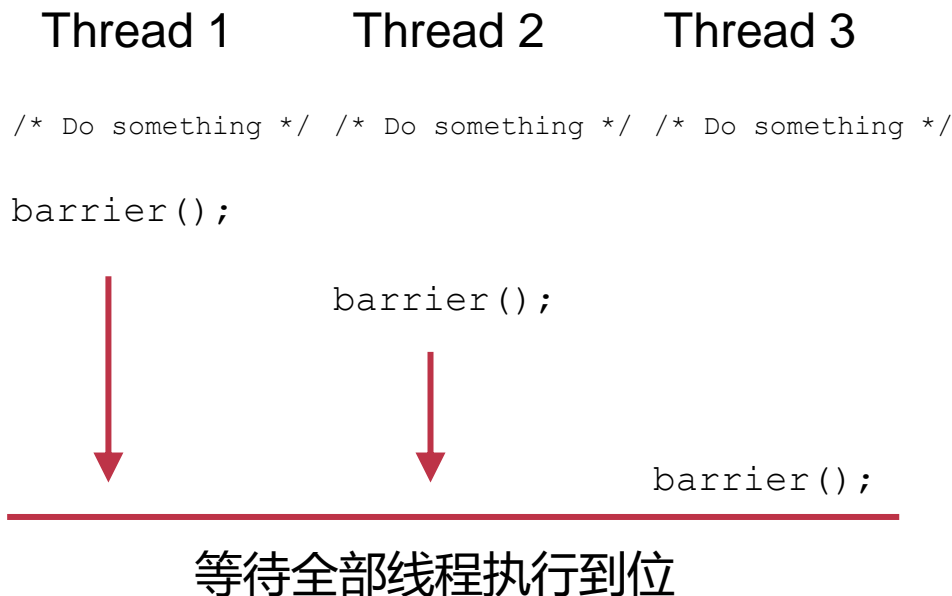


同步案例-1：多线程执行屏障

- 多线程执行屏障
- 等待全部执行到屏障后再继续执行

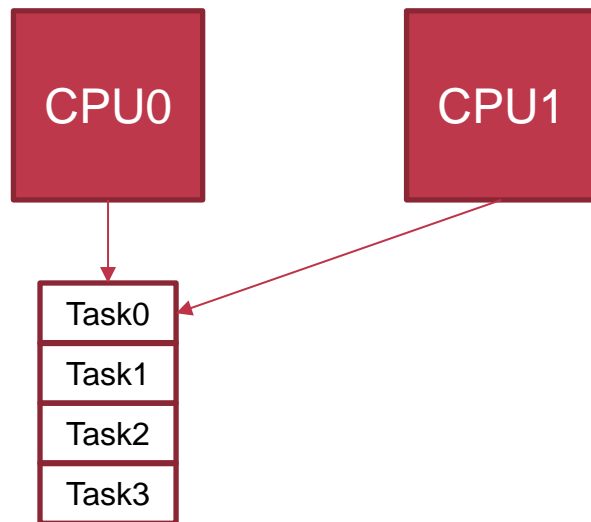
符合场景2：线程等待/唤醒

```
lock(&thread_cnt_lock);  
thread_cnt--;  
if (thread_cnt == 0)  
    cond_broadcast(cond);  
while(thread_cnt != 0)  
    cond_wait(&cond, &thread_cnt_lock);  
unlock(&thread_cnt_lock);
```



同步案例-2：负载均衡

- 每核心等待队列
- 空闲时允许窃取其他核心的任务

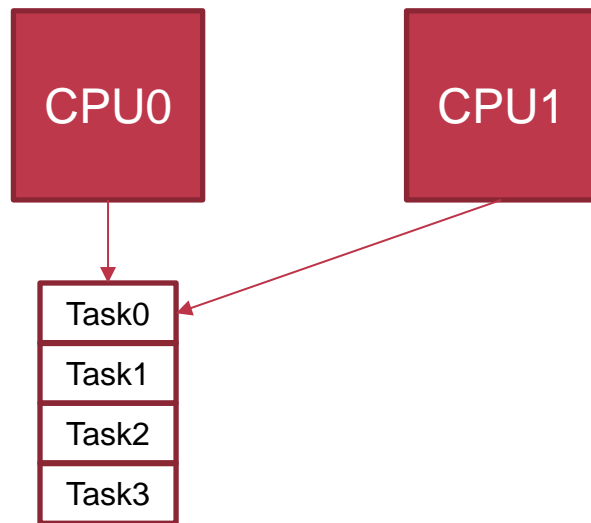


同步案例-2：负载均衡

- 每核心等待队列
- 空闲时允许窃取其他核心的任务

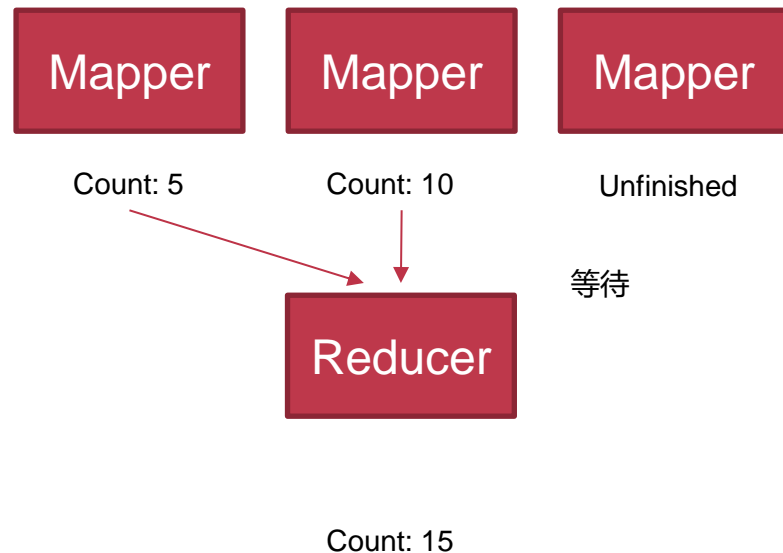
符合场景1: 共享资源互斥访问

```
lock(ready_queue_lock[0]);
```



同步案例-3: map-reduce

- Word-count: 大文本拆分字数统计
- Mapper: 统计一部分文本自述
- Reducer: 一旦其中任意数量的 Mapper 结束, 就累加其结果



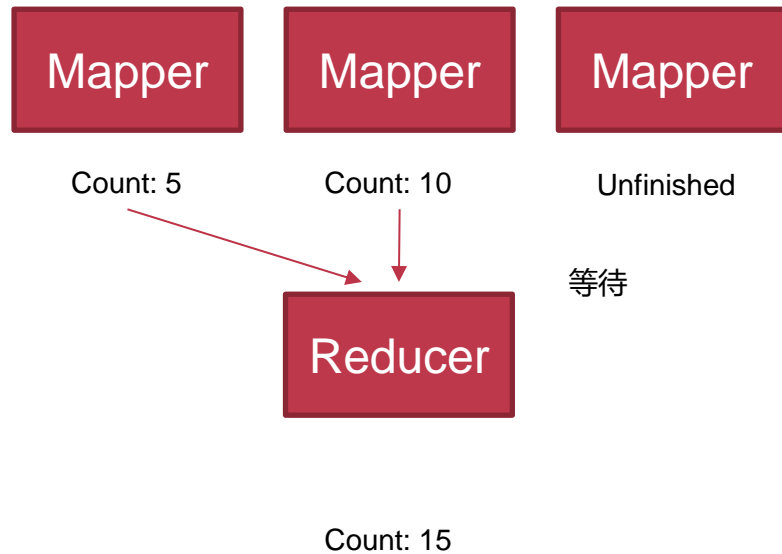
同步案例-3: map-reduce

- Word-count: 大文本拆分字数统计
- Mapper: 统计一部分文本自述
- Reducer: 一旦其中任意数量的Mapper结束, 就累加其结果

符合场景2: 线程等待/唤醒

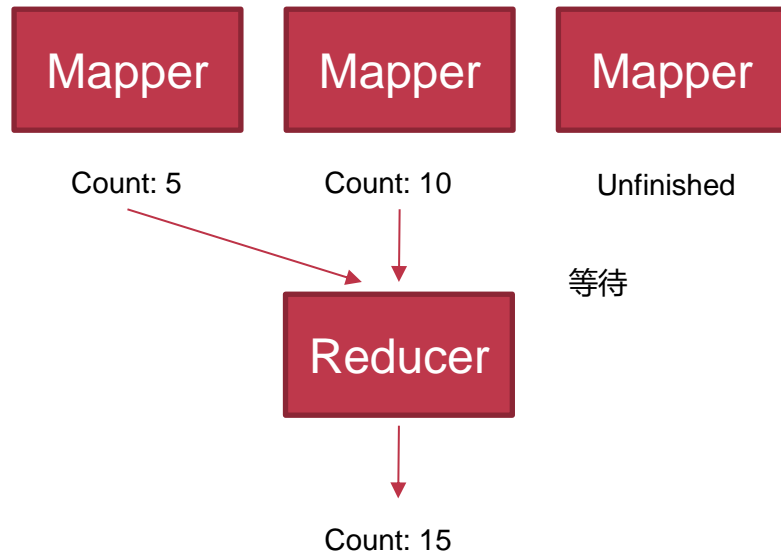
Mapper

```
lock(&finished_cnt_lock);  
finished_cnt ++;  
cond_signal(&cond);  
unlock(&thread_cnt_lock);
```



同步案例-3: map-reduce

- Word-count: 大文本拆分字数统计
- Mapper: 统计一部分文本自述
- Reducer: 一旦其中任意数量的Mapper结束, 就累加其结果



符合场景2: 线程等待/唤醒

Reducer

```
lock(&finished_cnt_lock);  
while(finished_cnt == 0)  
    cond_wait(&cond, &finished_cnt_lock);  
/* collect result */  
finished_cnt = 0;  
unlock(&thread_cnt_lock);
```

一次性拿走所有的finished
的Mapper的结果

同步案例-4：网页渲染

- 网页等待所有的请求均完成后再进行渲染

request_
cb_1

request_
cb_2

request_
cb_3

渲染线程

同步案例-4：网页渲染

- 网页等待所有的请求均完成后
再进行渲染

request_
cb_1

request_
cb_2

request_
cb_3

场景2: 等待/唤醒

Request_cb

```
lock(&glock);  
finished_cnt ++;  
if (finished_cnt == req_cnt)  
    cond_signal(&gcond);  
unlock(&glock);
```

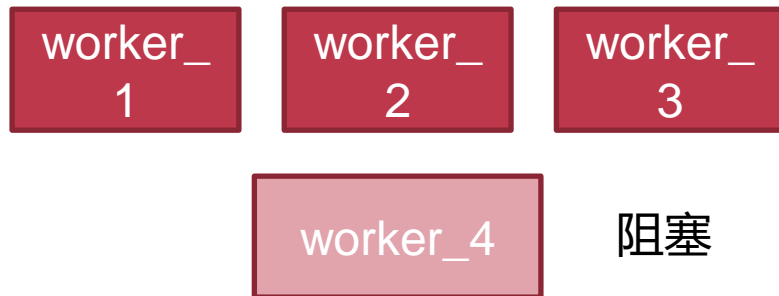
渲染线程

渲染线程

```
lock(&glock);  
while (finished_cnt != req_cnt)  
    cond_wait(&gcond, &glock);  
unlock(&glock);
```

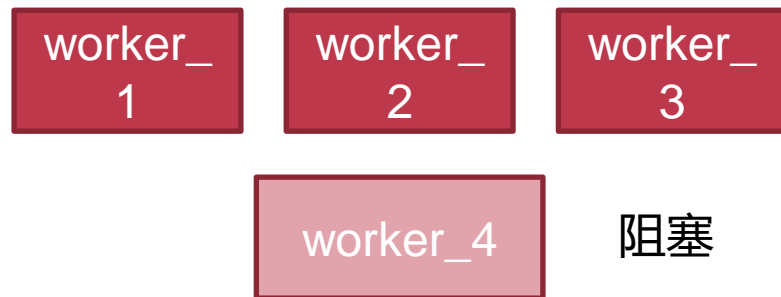
同步案例-5：线程池并发控制

- 控制同一时刻可以执行的线程数量
- 原因：有的线程阻塞时允许新的线程替上
- 例子：允许同时三个线程执行



同步案例-5：线程池并发控制

- 控制同一时刻可以执行的线程数量
- 原因：有的线程阻塞时允许新的线程替上
- 例子：允许同时三个线程执行

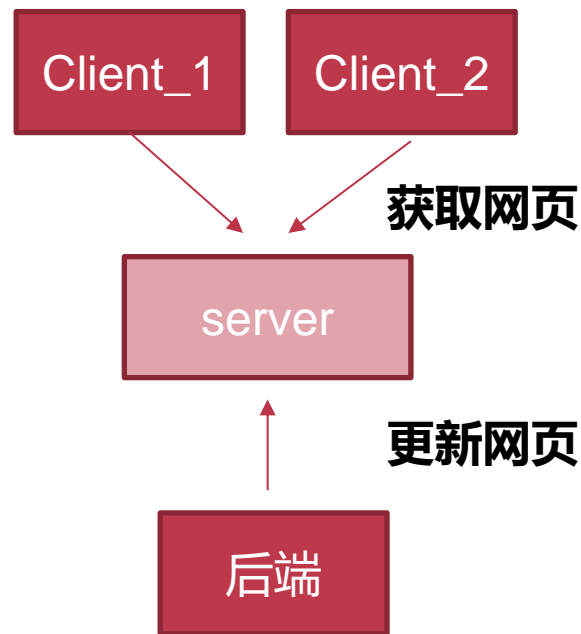


场景3: 视剩余可并行执行线程数量为有限资源

```
thread_routine () {  
    wait(&thread_cnt_sem);  
    /* doing something */  
    signal(&thread_cnt_sem);  
}
```


同步案例-6：网页服务器

- 处理响应客户端获取静态网页需求
- 处理后端更新静态网页需求
- 不允许读取更新到一半的页面



同步案例-6：网页服务器

- 处理响应客户端获取静态网页需求
- 处理后端更新静态网页需求
- 不允许读取更新到一半的页面

衍生场景1: 读写场景，可以使用读写锁

- client用读锁
- 后端用写锁

