IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

同步原语

董明凯

IPADS · 上海交通大学

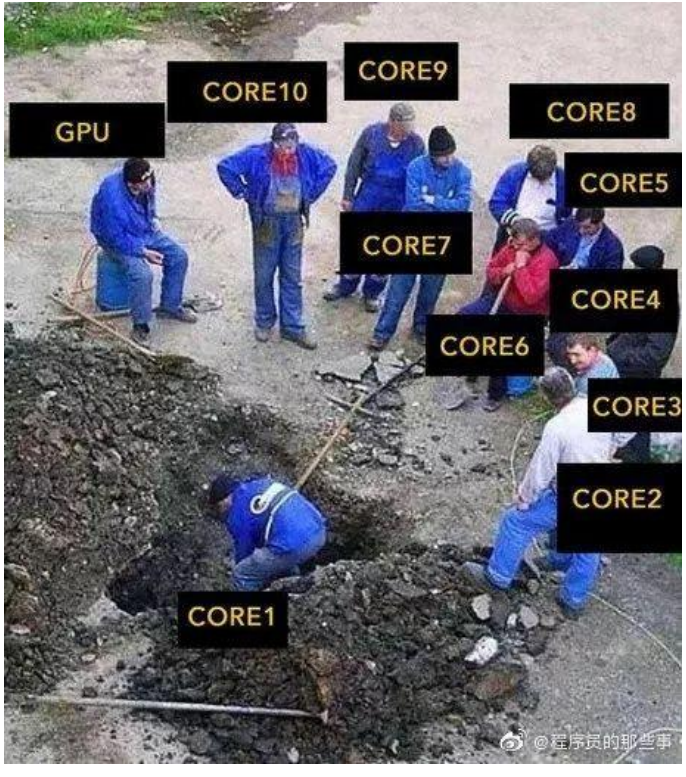
版权声明

- 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

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

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 (RACE CONDITION)

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	线程2
	a++;	a++;

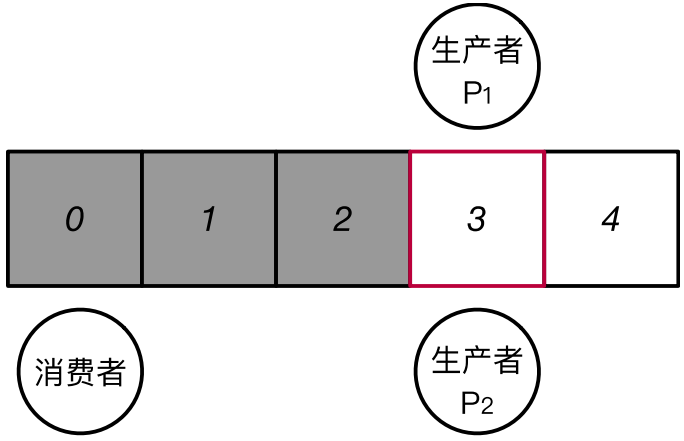
a的初始值为3，结果应当为5

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

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 → 多生产者消费者

生产者消费者问题的基础实现

- **基础实现: 生产者**

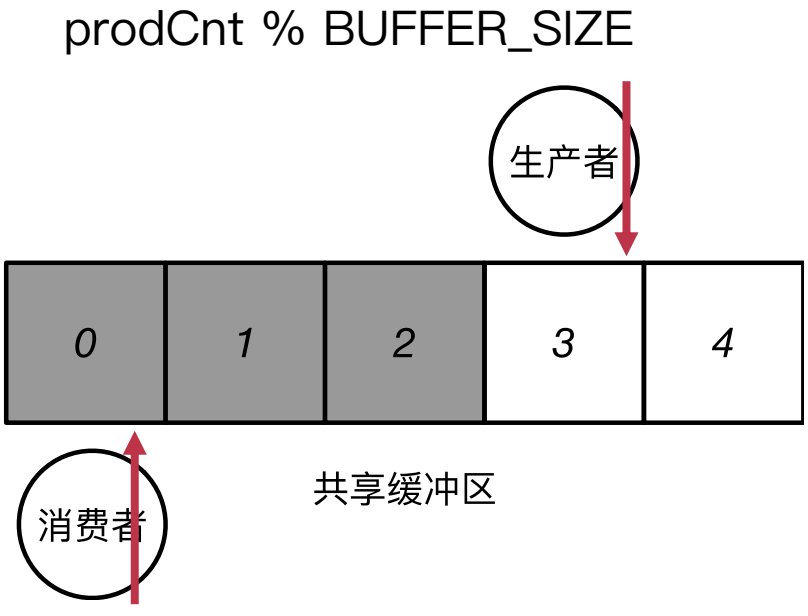
```
while (true) {
        /* Produce an item */
        while (prodCnt – consCnt == BUFFER_SIZE)
                ;   /* do nothing –– no free buffers */
        buffer[prodCnt % BUFFER_SIZE] = item;
        prodCnt = prodCnt + 1;
}
```

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

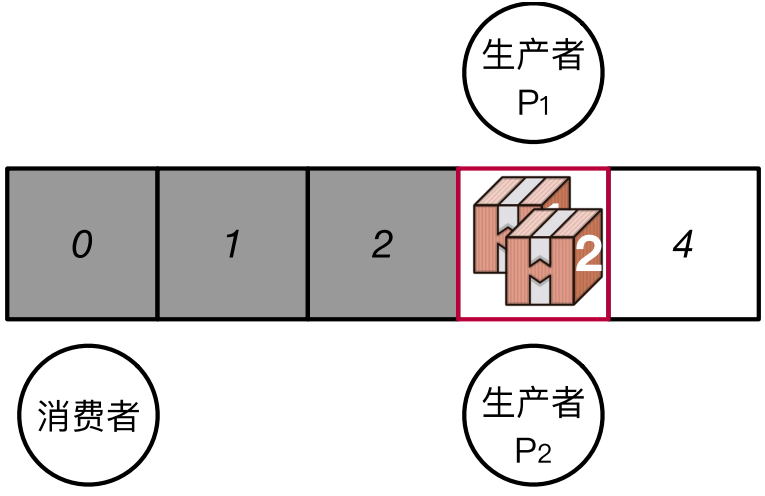


通过两个计数器来协调生产者与消费者，

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;*
```



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

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

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**

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

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

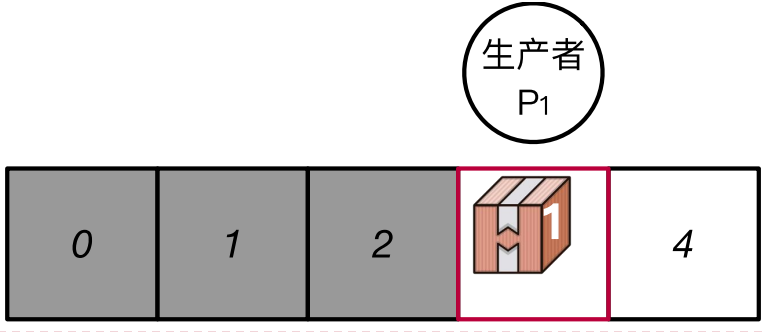
```
while (prodCnt – consCnt == BUFFER_SIZE)
            ;   /* do nothing –– no free buffers */

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

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

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

prodCnt = prodCnt + 1;*
```

临界区

*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;

获取互斥锁
进入临界区

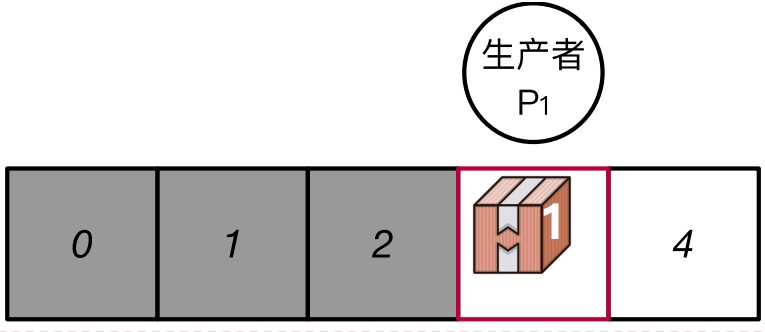
(bufCnt = 3)
lock(&buffer_lock);

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

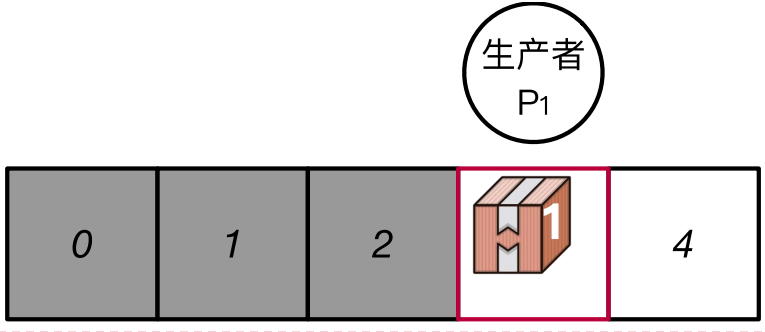
(bufCnt = 4)
lock(&buffer_lock);

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

条件变量

条件变量

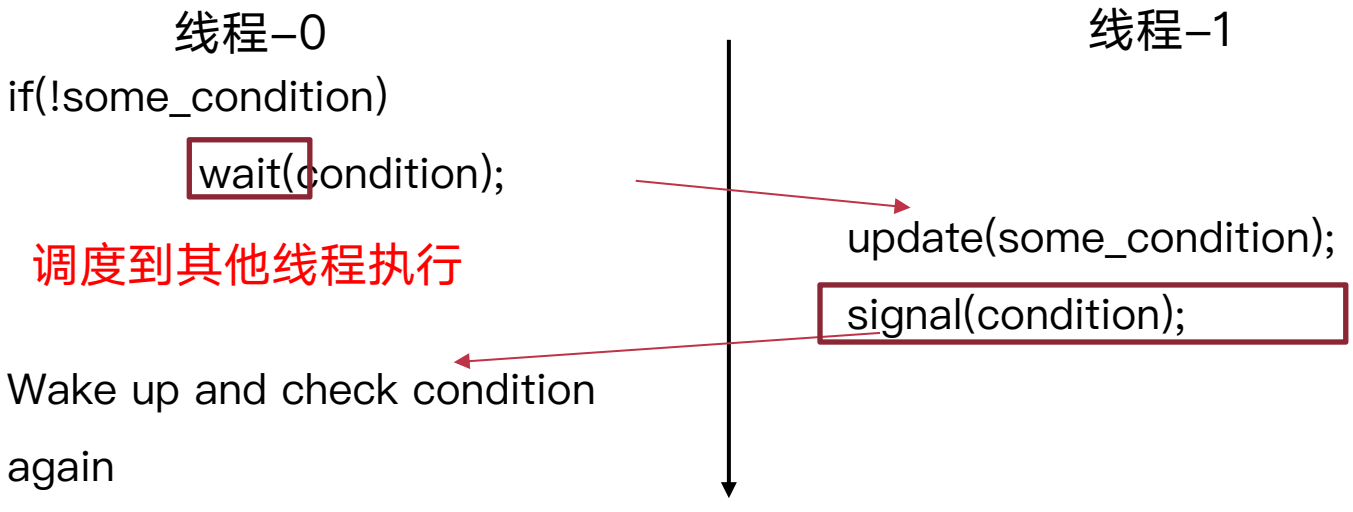
条件变量：利用**睡眠/唤醒**机制，避免无意义的等待

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**

唤醒是否需要在临界区中？也需要的。
为什么？下节课讲实现的时候再揭晓

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);
```

线程 2

```
empty_slot = 1
```

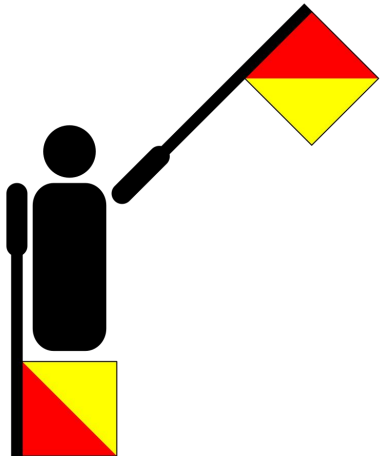
有新的空位，唤醒

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

```
empty_slot = 0
```

重新拿到锁

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



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);
        // ...
}
```

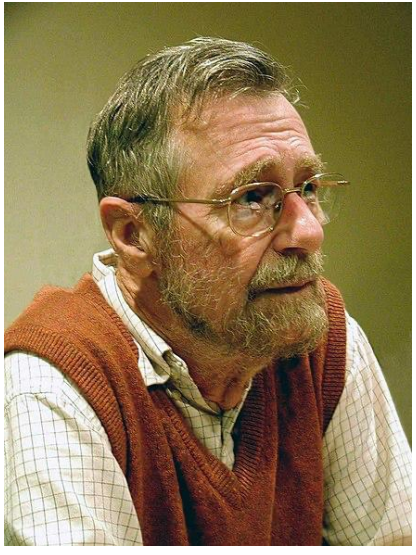
当前实现：需要单独创建互斥锁与条件变量，并手动通过计数器来管理资源数量

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



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

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 */;
        S--;
}
```

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

信号量的使用

```
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);
        // ...
}
```

使用信号量可以将
其压缩到**一行代码**

```
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);
}

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

消耗empty_slot

增加filled_slot

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

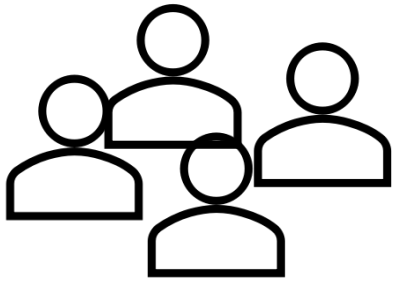
读者

思考：多个读者如果希望读公告栏，他们互斥吗？

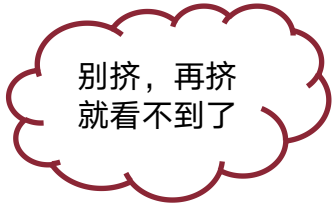
思考：如何避免读者看到一半就被写者撤走了，我们怎么办？

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

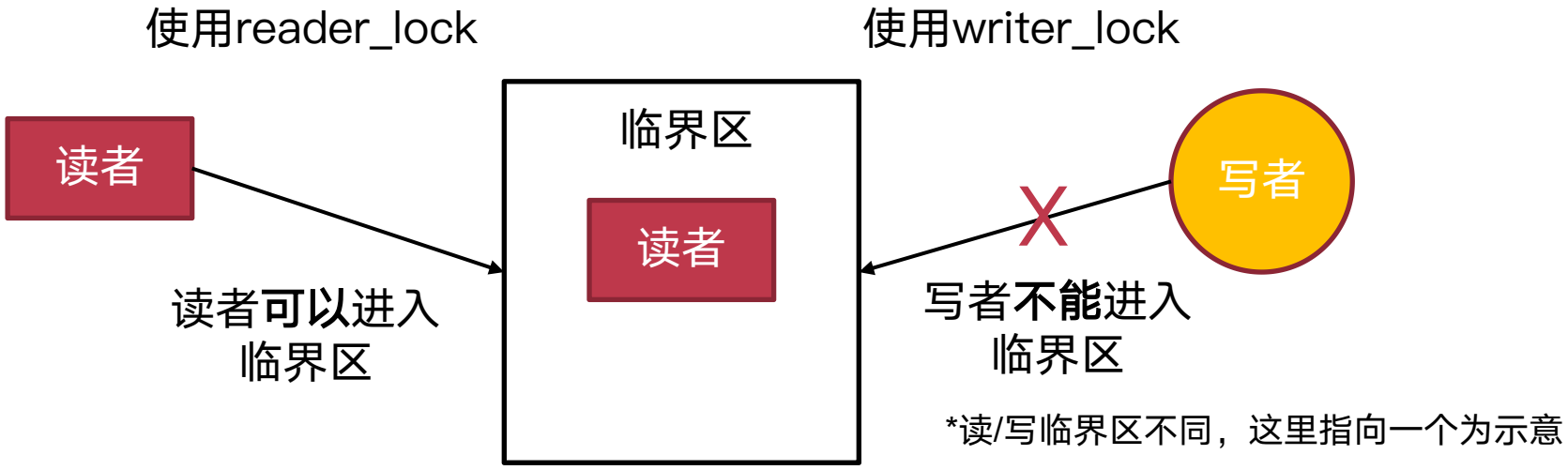
```
struct rwlock *lock;
char data[SIZE];

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



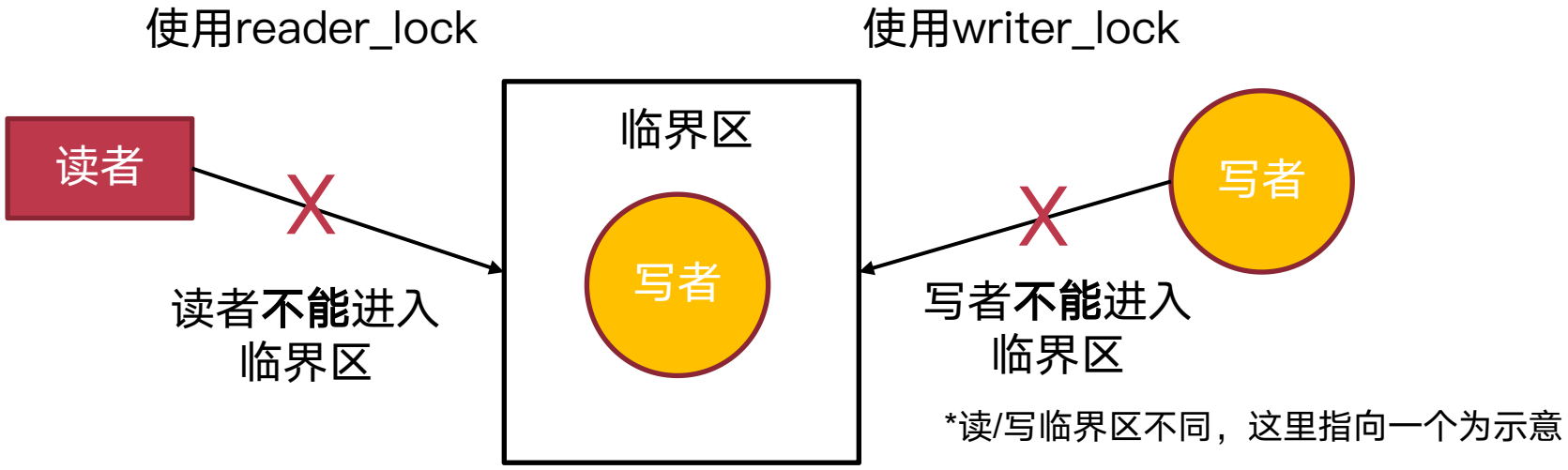
*读/写临界区不同，这里指向一个为示意

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/写临界区不同，这里指向一个为示意

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);

同一线程

unlock(&lock0);

Thread 0

sem_wait(&s0);

Thread 1

另一个线程

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

```
lock(&glock);

// Reader CS

unlock(&glock);
```

Reader 1

```
lock(&glock);
```

被阻塞

```
lock(&glock);

// Reader CS

unlock(&glock);
```

Reader 0

```
reader_lock(
  &glock);

// Reader CS

reader_unlock(
  &glock);
```

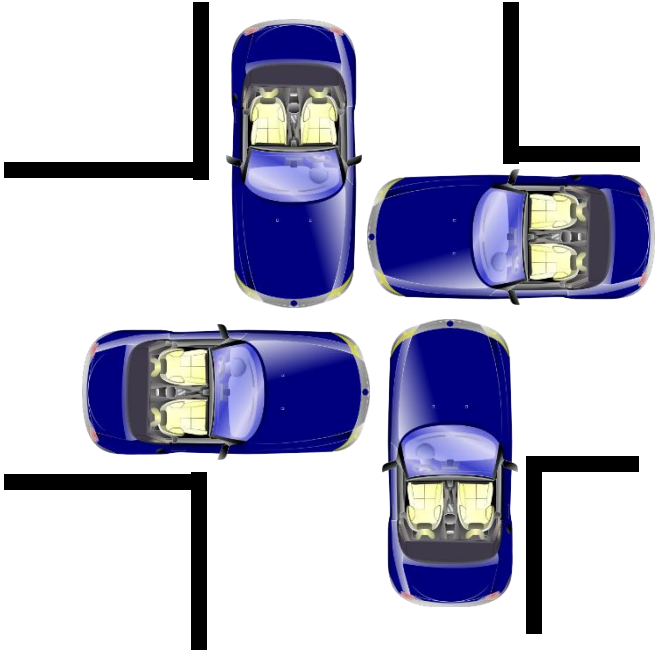
Reader 1

```
reader_lock(
  &glock);

// Reader CS

reader_unlock(
  &glock);
```

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

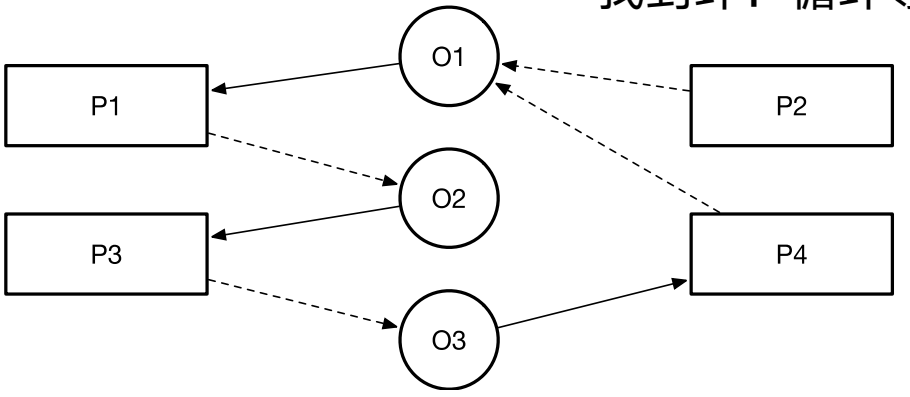
如何解决死锁？

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

找到环：循环等待



资源分配图

资源分配表

进程号	资源号
P1	O1
P3	O2
P4	O3

进程等待表

进程号	资源号
P1	O2
P2	O1
P3	O3

如何恢复？打破循环等待！

- 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

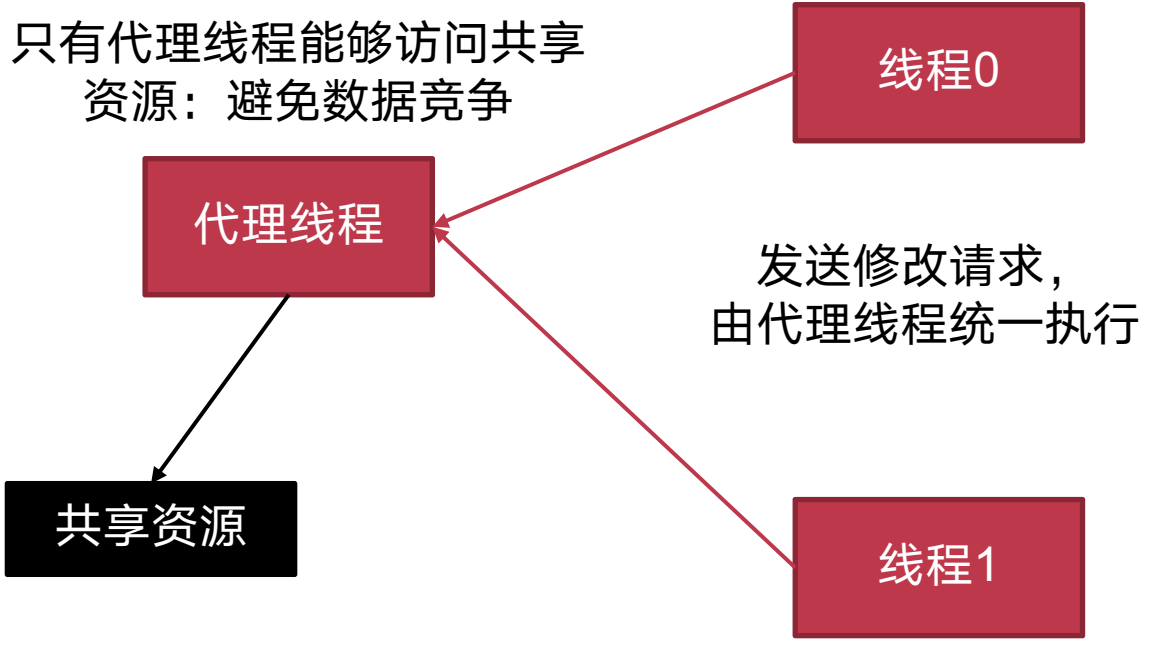
如何解决死锁？

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）



*代理锁 (Delegation Lock) 实现了该功能

死锁预防：四个方向

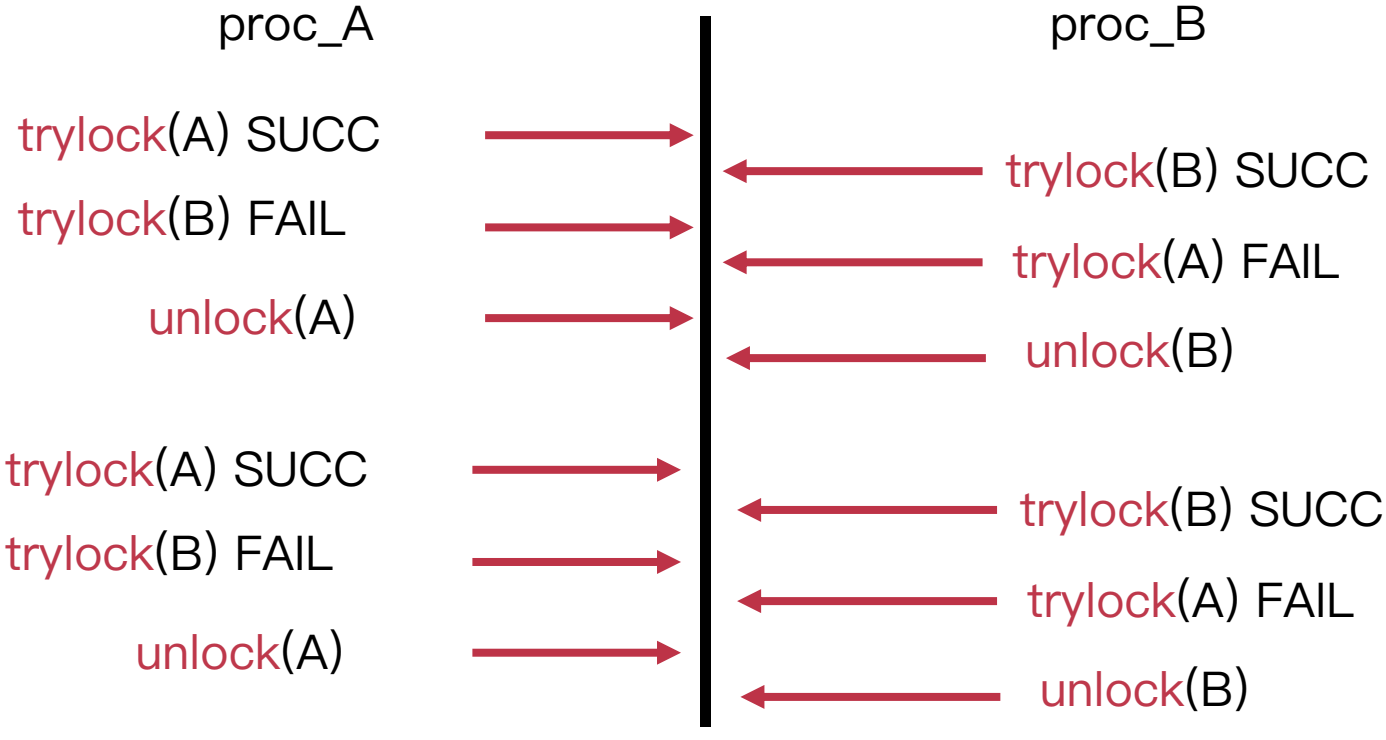
- 1、避免互斥访问：通过其他手段（如代理执行）
- 2、不允许持有并等待：一次性申请所有资源

```
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, ... , Pn}：

- 安全状态

  能找出至少一个执行序列，如P2->P1->P5...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		
P2							3	2
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		
P2	3	1	0	1	3	0	3→1	1→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