中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

同步原语：死锁问题

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

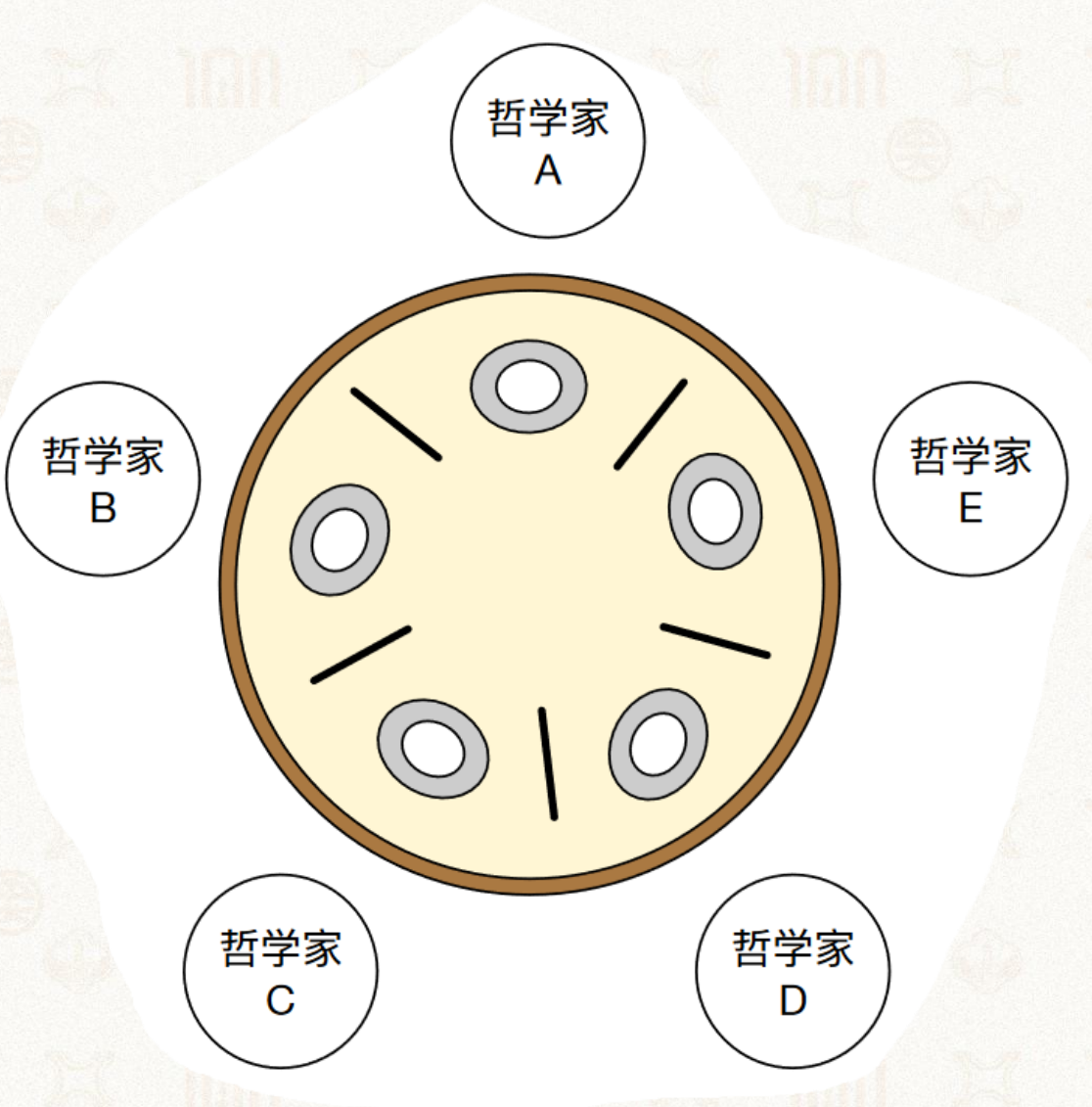
- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498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

大纲

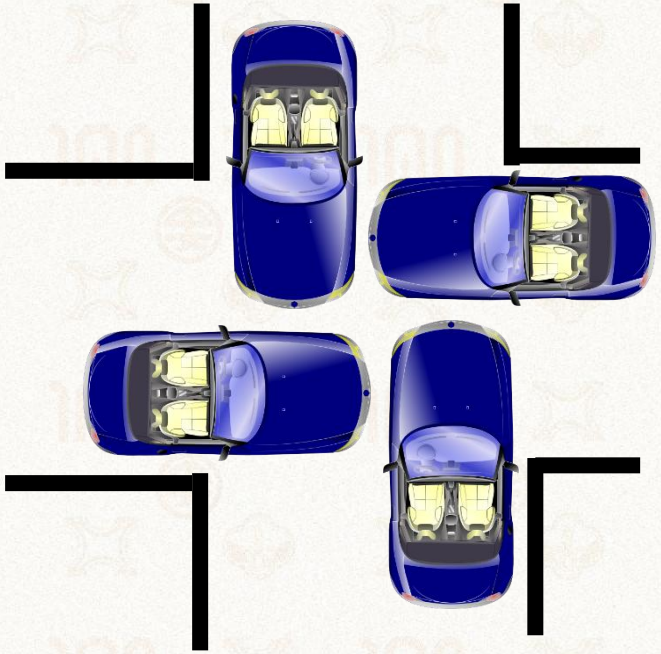
- 同步问题的背景
  - 多核场景
  - 生产者消费者模型
  - 临界区问题
- 互斥锁
  - 皮特森算法
  - 原子操作
  - 互斥锁抽象
    - 自旋锁
    - 排号自旋锁
- 条件变量
- 信号量
  - PV原语
- 读写锁
- 同步原语产生的问题
  - 死锁
    - 银行家算法
  - 活锁
  - 优先级反转

死锁：哲学家问题

- 每一位哲学家面前放着一份食物，左手与右手边分别放着一支筷子
  - 必须同时拿着这两支筷子才能开始进食
- 当哲学家感到饥饿
  - 尝试拿起这两支筷子
  - 如果成功获取，则开始进食，并在吃饱后放下筷子
  - 否则其将一直拿着已经获取到的筷子并等待邻座的哲学家放下筷子
- 当所有的哲学家都尝试进食且都拿起了他们左手的筷子时，每一位哲学家都在等待其右手的哲学家放下筷子



死锁



十字路口的“困境”

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

T1时刻的死锁

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);
    }
}
```

empty_cnt_lock被锁着，
怎么更新empty_slot?

条件变量的实现

```
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));
    }
}
```

有一个放锁再重新加锁的过程，为了什么？

为了给其它等待线程一个机会

一次性唤醒所有等待线程

死锁产生的原因

- 互斥访问
- 持有并等待
- 资源非抢占
- 循环等待

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

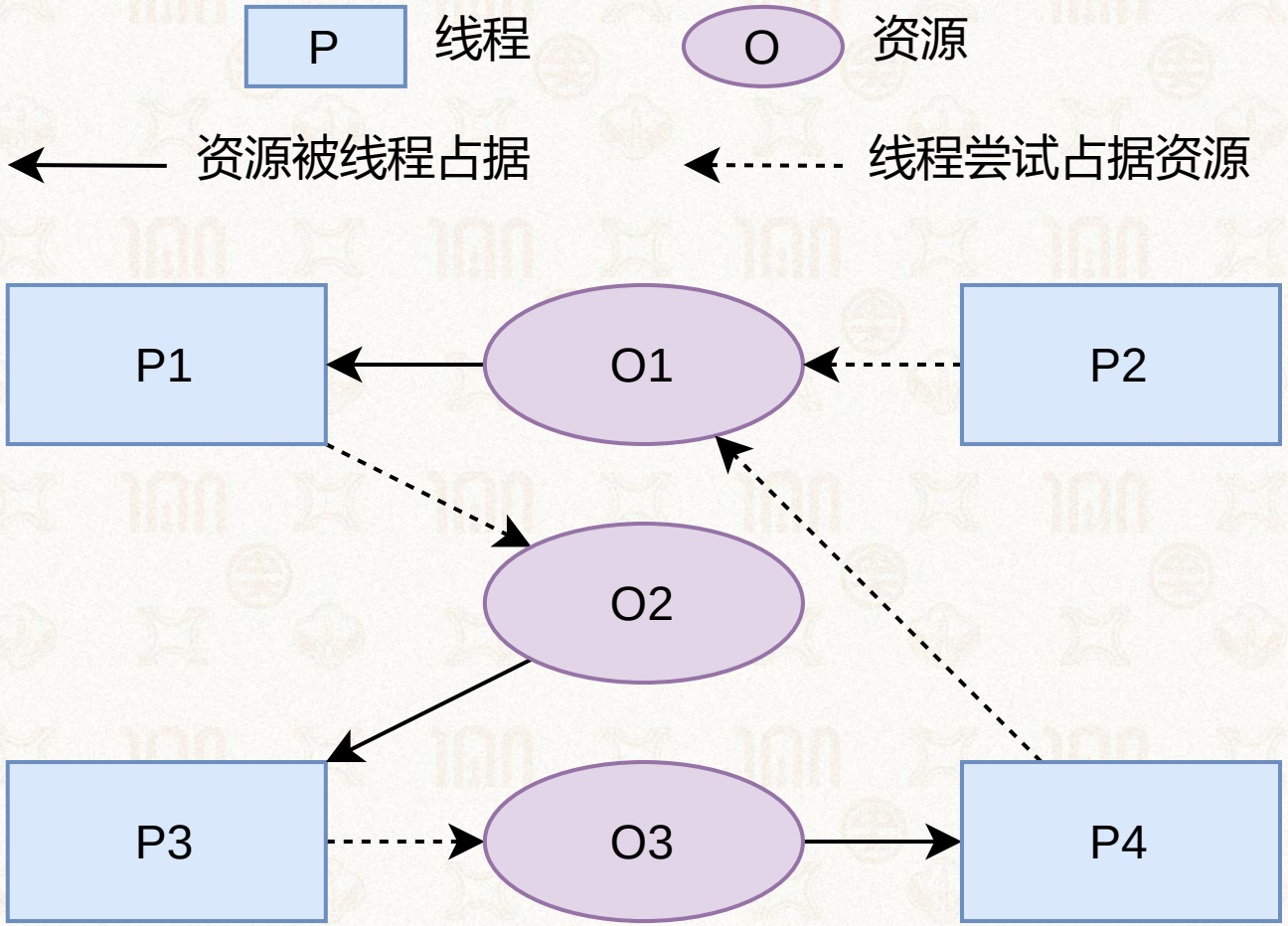
```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

T1时刻的死锁

如何解决死锁?

- 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

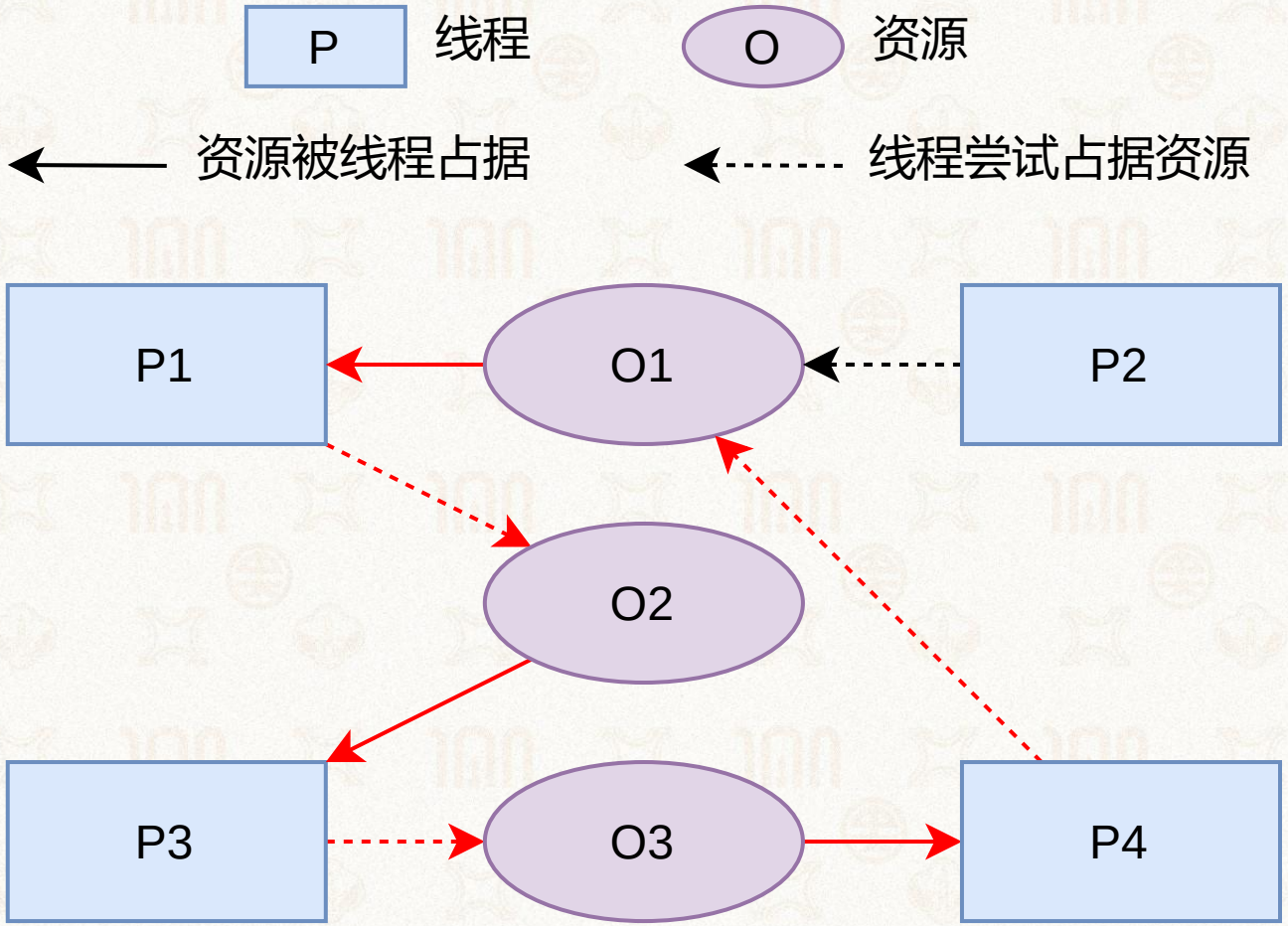


资源分配图

资源分配表	
线程号	资源号
P1	O1
P3	O2
P4	O3

线程等待表	
线程号	资源号
P1	O2
P2	O1
P3	O3
P4	O1

检测死锁与恢复



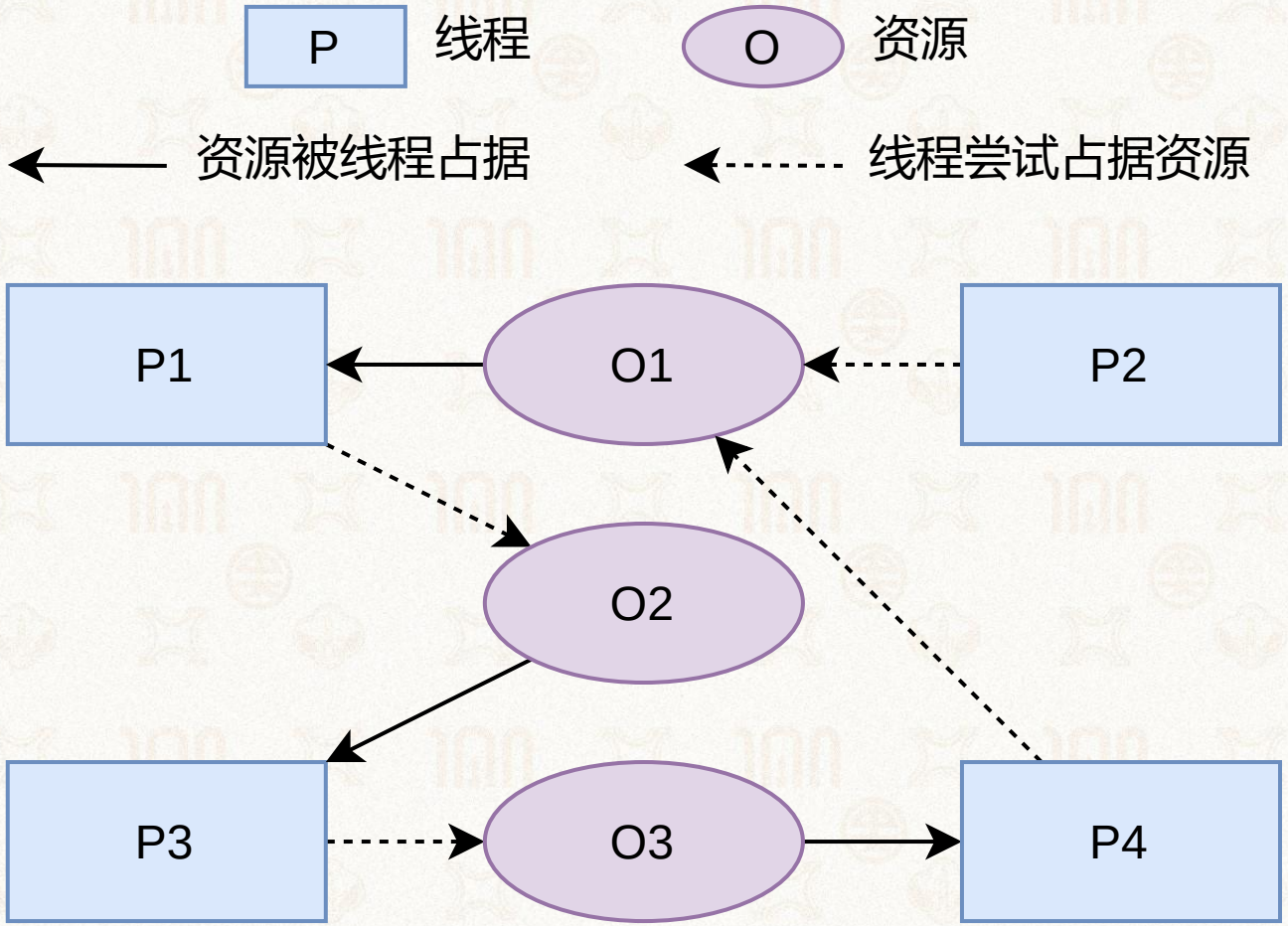
资源分配图

资源分配表	
线程号	资源号
P1	O1
P3	O2
P4	O3

线程等待表	
线程号	资源号
P1	O2
P2	O1
P3	O3
P4	O1

存在一个环，形成死锁

检测死锁与恢复



资源分配图

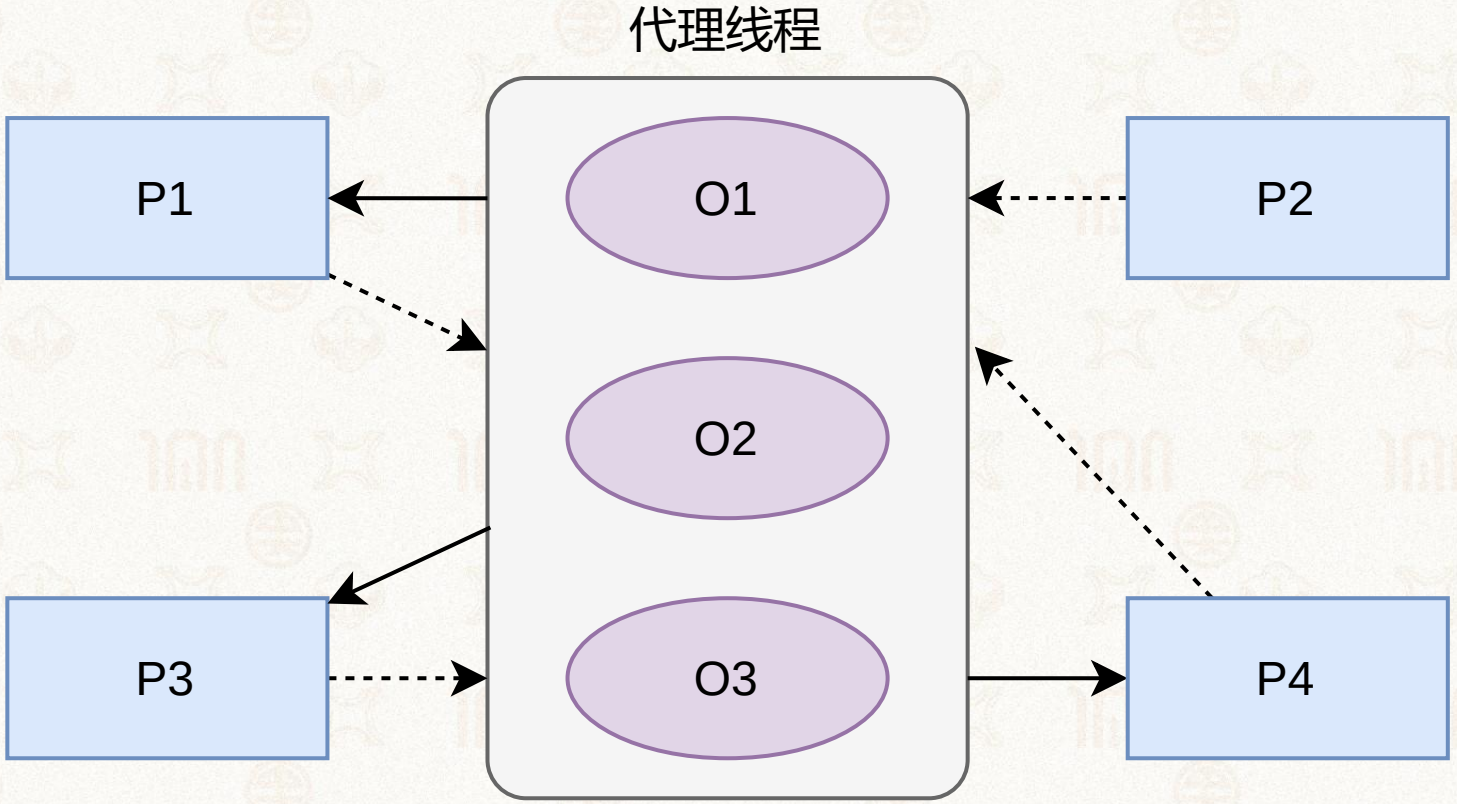
- 如何恢复？打破循环等待！
  - 直接kill所有循环中的线程
  - Kill一个，看有没有环，有的话继续kill
  - 全部回滚到之前的某一状态

如何解决死锁?

- 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）




```
lock()
count++
unlock()
```

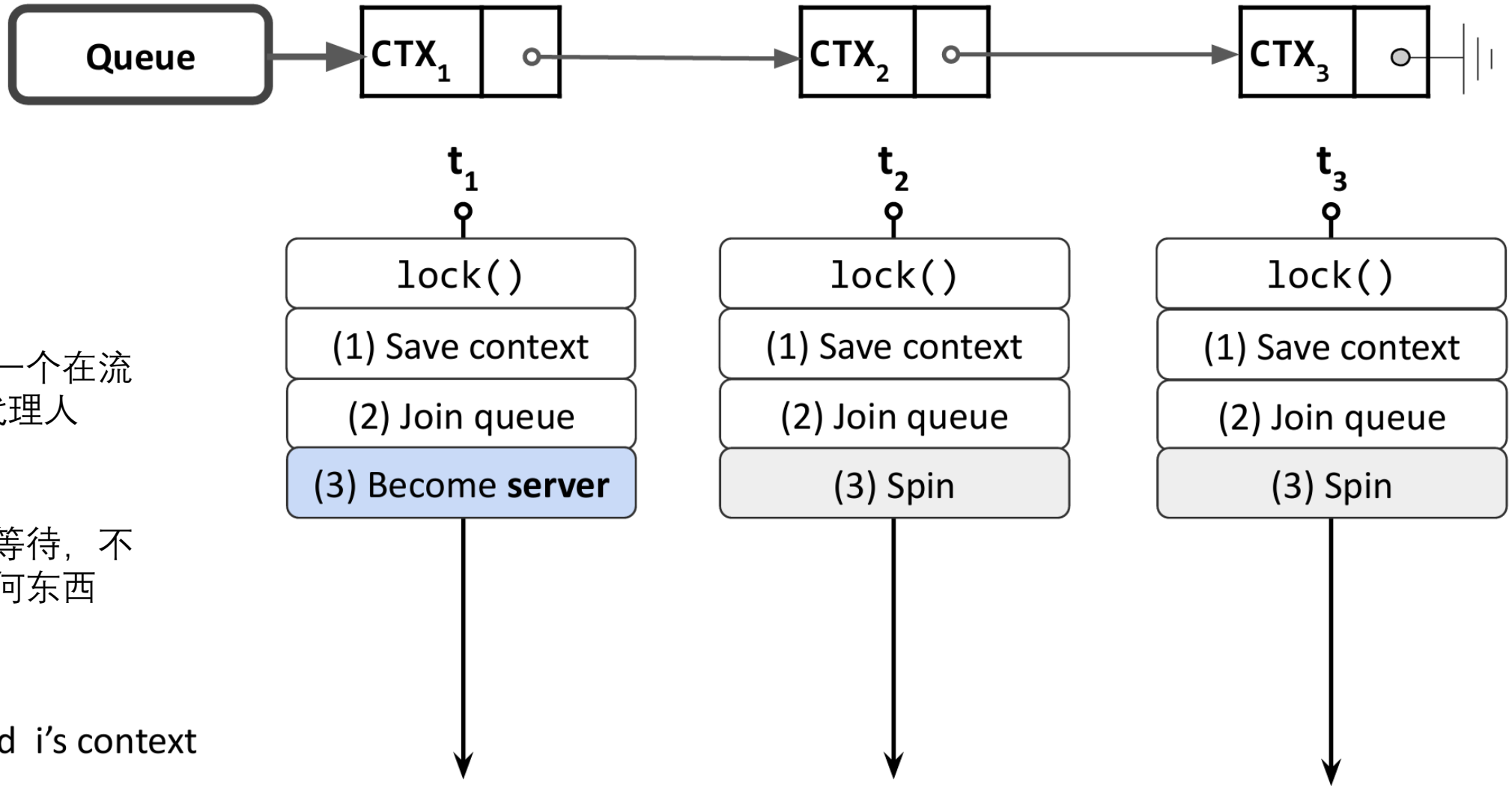
```
void incr_func() =
     count++

send_req_to_server(&incr_func)
```

接收执行结果

发送执行请求

- 缺点是大部分程序都不太容易修改为这种模式
  - 但能看到点希望：[OSDI23] Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLOCKS

TCLocks：透明代理锁



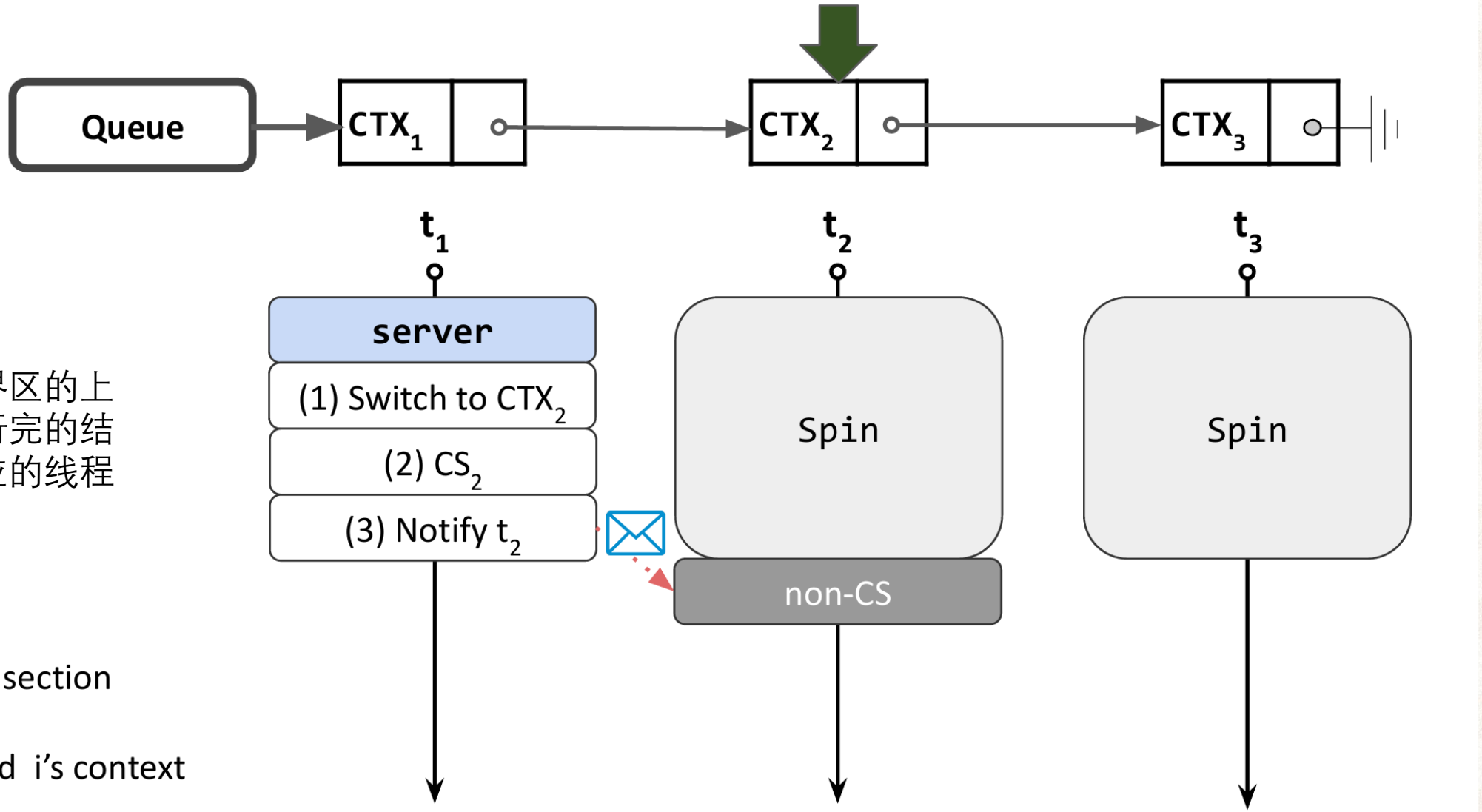
逻辑上有一个在流转的代理人

其它人只等待，不改变任何东西

t_i: thread i

CTX_i: thread i's context

TCLocks：透明代理锁



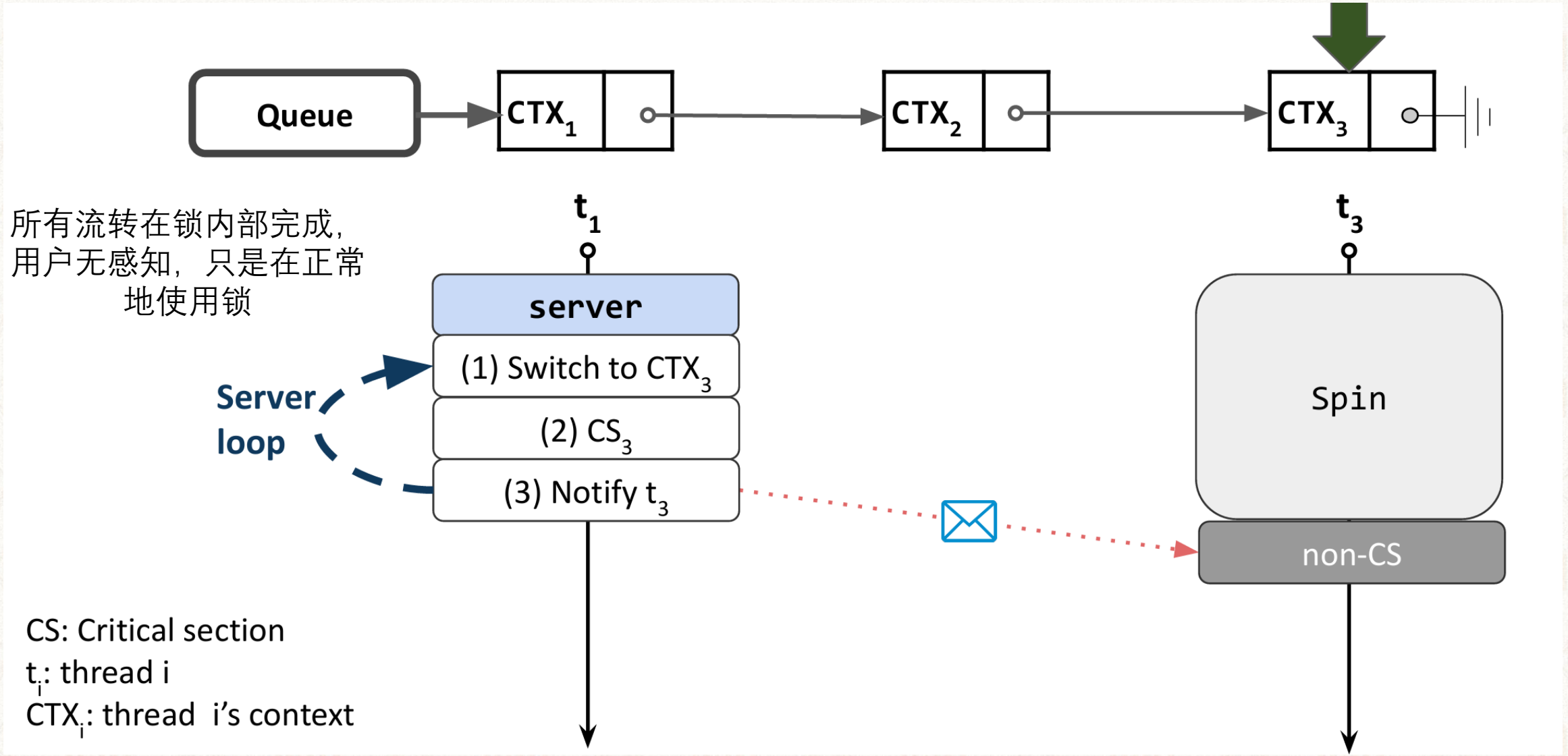
只切换临界区的上下文，执行完的结果发给对应的线程

CS: Critical section

t_i: thread i

CTX_i: thread i's context

TCLocks：透明代理锁



死锁预防：四个方向

- 2. 不允许持有并等待：一次性申请所有资源

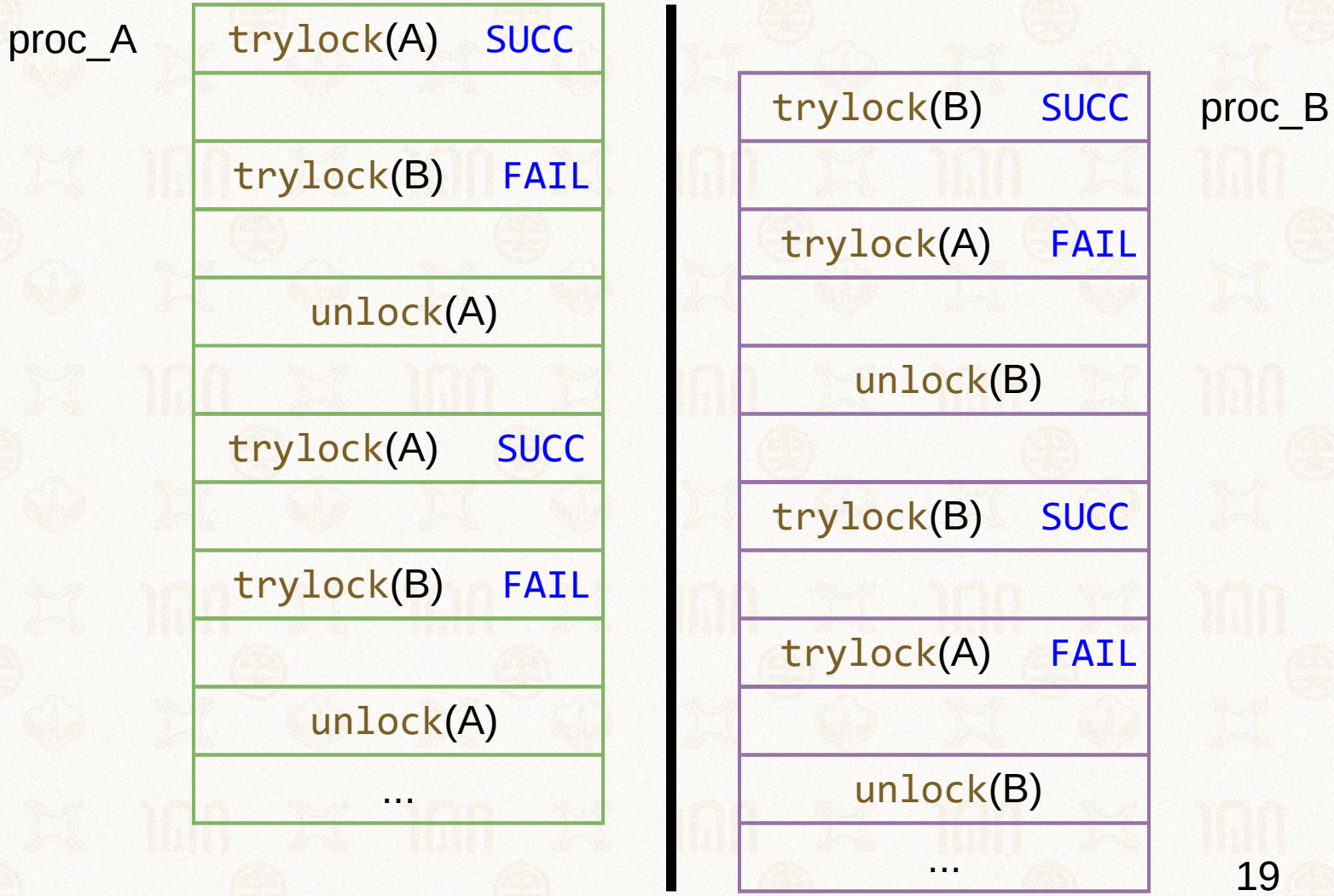
```
while(true) {
    if(trylock(A) == SUCC) {    // trylock非阻塞，立即返回成功或失败
        if(trylock(B) == SUCC) {
            // 临界区代码
            // ...
            unlock(B);
            unlock(A);
            break;
        } else {
            unlock(A); // 无法获取B，那么释放A
        }
    }
}
```

死锁预防：四个方向

- 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) {
            // 临界区代码
            // ...
            unlock(B);
            unlock(A);
            break;
        } else {
            unlock(A);
        }
    }
}
```

运气很差时可能出现如此往复，但运气不会一直这么差



死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源
  - 让proc_B回滚

```
void proc_A(void) {
    // …
    lock(A);
    // …
→   lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
→   // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源
  - 让proc_B回滚

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
→   unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
→   // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源
  - 让proc_B回滚
  - proc_A结束后再恢复proc_B的执行

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 3. 资源允许抢占：需要考虑如何恢复
  - proc_A挤占proc_B的资源
  - 让proc_B回滚
  - proc_A结束后再恢复proc_B的执行

- 回滚和恢复只适用于易于保存和恢复的场景

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
→   unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
→   // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

死锁预防：四个方向

- 4. 打破循环等待：按照特定顺序获取资源
  - 所有资源进行编号
  - 所有线程递增获取：(A、B、C、D...)

  - 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

```
void proc_A(void) {
    // …
    lock(A);
    // …
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    // …
    lock(B);
    // …
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

如何解决死锁?

- 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

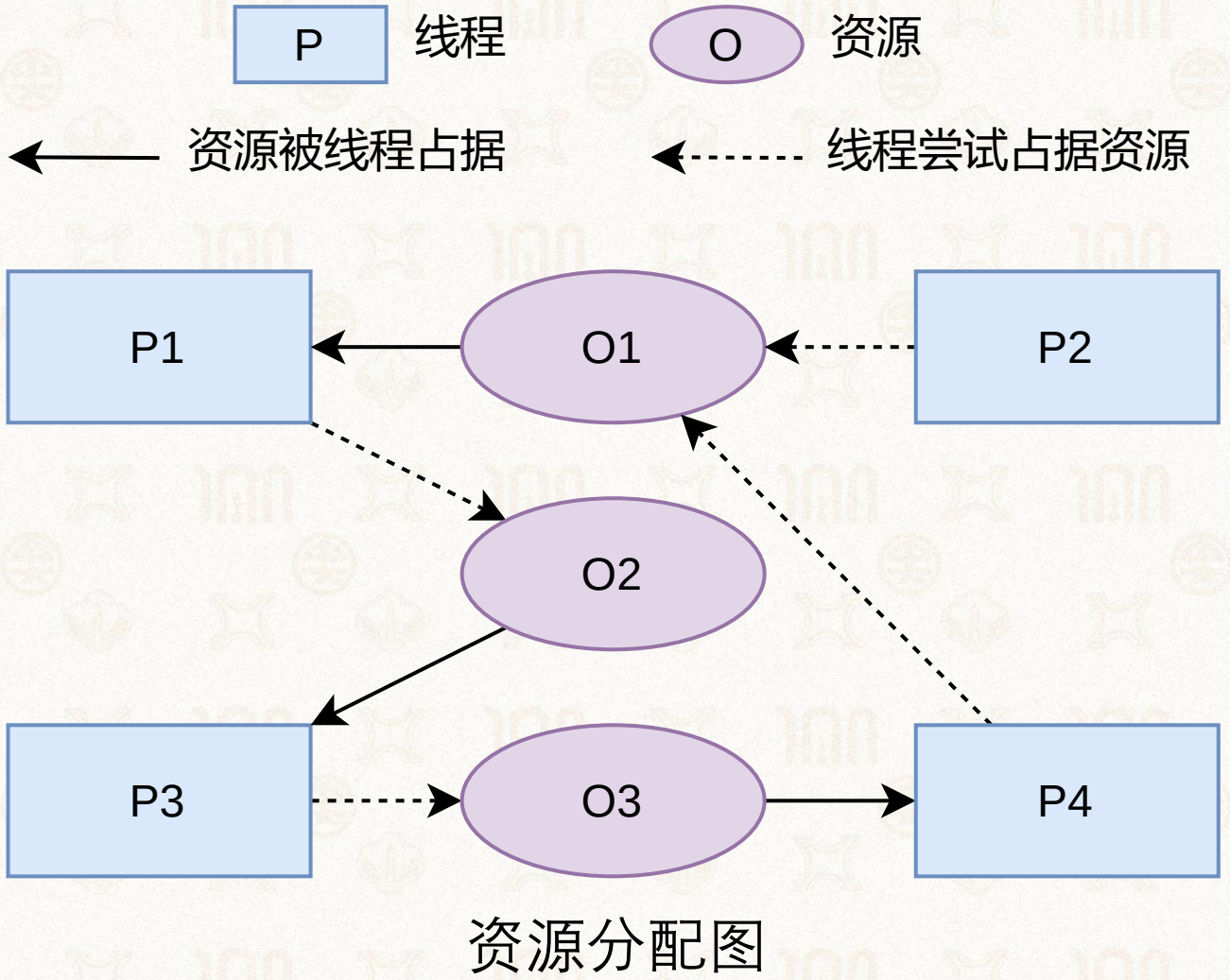
大纲

- 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

死锁避免：运行时检查是否会出现死锁

- 特定顺序获取资源就可能不死锁
  - 不死锁的调用顺序就是安全序列

- 银行家算法
  - 所有线程获取资源需要管理者同意
  - 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



资源分配图

银行家算法

- 初始状态

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

银行家算法：安全性检查

- 1. 创建临时数组，模拟执行过程中可用的资源数量

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

模拟全局可利用资源数量	
A类资源	B类资源
3	1

银行家算法：安全性检查

- 2. 找到可以被执行的线程(模拟资源都满足)

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

B类资源不够用，不能执行P1

模拟全局可利用资源数量	
A类资源	B类资源
3	1

银行家算法：安全性检查

- 2. 找到可以被执行的线程(模拟资源都满足)

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

AB类资源都不够用，不能执行P3

模拟全局可利用资源数量	
A类资源	B类资源
3	1

银行家算法：安全性检查

- 2. 找到可以被执行的线程(模拟资源都满足)

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

资源都够用，可以执行P2

模拟全局可利用资源数量	
A类资源	B类资源
3	1

银行家算法：安全性检查

➢ 3. 模拟执行，结束之后要释放所有资源

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
→ P2	3	1	3	1	0	0		
P3	10	11	5	1	5	10		

A类资源被消耗

模拟全局可利用资源数量	
A类资源	B类资源
0	1

银行家算法：安全性检查

➢ 3. 模拟执行，结束之后要释放所有资源

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
→ P2	3	1	0	0	0	0		
P3	10	11	5	1	5	10		

P2执行结束，释放所有资源

模拟全局可利用资源数量	
A类资源	B类资源
3	2

银行家算法：安全性检查

➢ 4. 遍历所有线程，检查是否都可以被执行

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
P2	3	1	0	0	0	0		
P3	10	11	5	1	5	10		

资源够用，可执行P1

模拟全局可利用资源数量	
A类资源	B类资源
3	2

银行家算法：安全性检查

➢ 4. 遍历所有线程，检查是否都可以被执行

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	5	10	0	0	3	1
P2	3	1	0	0	0	0		
P3	10	11	5	1	5	10		

执行P1，资源都分配

模拟全局可利用资源数量	
A类资源	B类资源
0	0

银行家算法：安全性检查

➢ 4. 遍历所有线程，检查是否都可以被执行

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
→ P1	5	10	0	0	0	0	3	1
P2	3	1	0	0	0	0		
P3	10	11	5	1	5	10		

P1执行结束，退回所有资源

模拟全局可利用资源数量	
A类资源	B类资源
5	10

银行家算法：安全性检查

➢ 4. 遍历所有线程，检查是否都可以被执行

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	0	0	0	0	3	1
P2	3	1	0	0	0	0		
→ P3	10	11	5	1	5	10		

模拟全局可利用资源数量	
A类资源	B类资源
5	10

继续检查，发现资源够执行P3

银行家算法：安全性检查

➢ 4. 遍历所有线程，检查是否都可以被执行

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	0	0	0	0	3	1
P2	3	1	0	0	0	0		
→ P3	10	11	5	1	5	10		

模拟全局可利用资源数量	
A类资源	B类资源
5	10

继续检查，发现资源够执行P3

银行家算法：安全性检查

- 所有线程都已遍历，都可以被执行

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	0	0	0	0	3	1
P2	3	1	0	0	0	0		
→ P3	10	11	0	0	0	0		

所以P2, P1, P3就是一个安全的序列

按照安全序列的顺序调度，就不会死锁

模拟全局可利用资源数量	
A类资源	B类资源
10	11

银行家算法：安全性检查

- 算法有效的前提条件：

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

这些数据都应该是静态的

银行家算法：安全性检查

- 来试试强行分配部分资源

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	2	8	3	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

模拟全局可利用资源数量	
A类资源	B类资源
3	1

银行家算法：安全性检查

- 来试试强行分配部分资源

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
P1	5	10	5	8	0	2	3	1
P2	3	1	0	1	3	0		
P3	10	11	5	1	5	10		

把A分配给P1之后，再无可以运行的线程

模拟全局可利用资源数量	
A类资源	B类资源
0	1

银行家算法：实战

- 资源有两类：A、B，每类都只有1份

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	0	0	1	1	1	1
proc_B	1	1	0	0	1	1		

银行家算法：实战

- 假设先调度proc_A, 消耗A类资源1份

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	1	0	0	1	0	1
proc_B	1	1	0	0	1	1		

银行家算法：实战

- 可否调度执行proc_B?

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

模拟全局可利用资源数量	
A类资源	B类资源
0	1

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	1	0	0	1	0	1
proc_B	1	1	0	0	1	1		

银行家算法：实战

- 可否调度执行proc_B?

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

模拟全局可利用资源数量	
A类资源	B类资源
0	0

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	1	0	0	1	0	1
proc_B	1	1	0	1	1	0		

银行家算法：实战

- proc_B无法顺利执行

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

模拟全局可利用资源数量	
A类资源	B类资源
0	0

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	1	0	0	1	0	1
proc_B	1	1	0	1	1	0		

银行家算法：实战

- 安全序列是？

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

模拟全局可利用资源数量	
A类资源	B类资源
0	1

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	1	0	0	1	0	1
proc_B	1	1	0	0	1	1		

银行家算法：实战

- proc_A可完整执行

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

模拟全局可利用资源数量	
A类资源	B类资源
0	0

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	1	1	0	0	0	1
proc_B	1	1	0	0	1	1		

银行家算法：实战

- proc_A执行完成

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

模拟全局可利用资源数量	
A类资源	B类资源
1	1

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	0	0	0	0	0	1
proc_B	1	1	0	0	1	1		

银行家算法：实战

- proc_B可执行

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

模拟全局可利用资源数量	
A类资源	B类资源
1	1

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	0	0	0	0	0	1
proc_B	1	1	0	0	1	1		

银行家算法：实战

- 安全序列为：proc_A, proc_B 执行完A后再执行B可避免死锁

```
void proc_A(void) {
    lock(A);
    // T1 时刻
    lock(B);
    // 临界区
    unlock(B);
    unlock(A);
}
```

```
void proc_B(void) {
    lock(B);
    // T1 时刻
    lock(A);
    // 临界区
    unlock(A);
    unlock(B);
}
```

模拟全局可利用资源数量	
A类资源	B类资源
1	1

线程	每个线程的最大需求量		已分配资源数量		还需要的资源数量		全局可利用资源数量	
	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源	A类资源	B类资源
proc_A	1	1	0	0	0	0	0	1
proc_B	1	1	0	0	1	1		

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

活锁

- “不允许持有并等待”会产生活锁

```
while(true) {
    if(trylock(A) == SUCC) {
        if(trylock(B) == SUCC) {
            // 临界区代码
            // ...
            unlock(B);
            unlock(A);
            break;
        } else {
            unlock(A);
        }
    }
}
```

proc_A

proc_A	proc_B
trylock(A) SUCC	trylock(B) SUCC
trylock(B) FAIL	trylock(A) FAIL
unlock(A)	unlock(B)
trylock(A) SUCC	trylock(B) SUCC
trylock(B) FAIL	trylock(A) FAIL
unlock(A)	unlock(B)
...	...

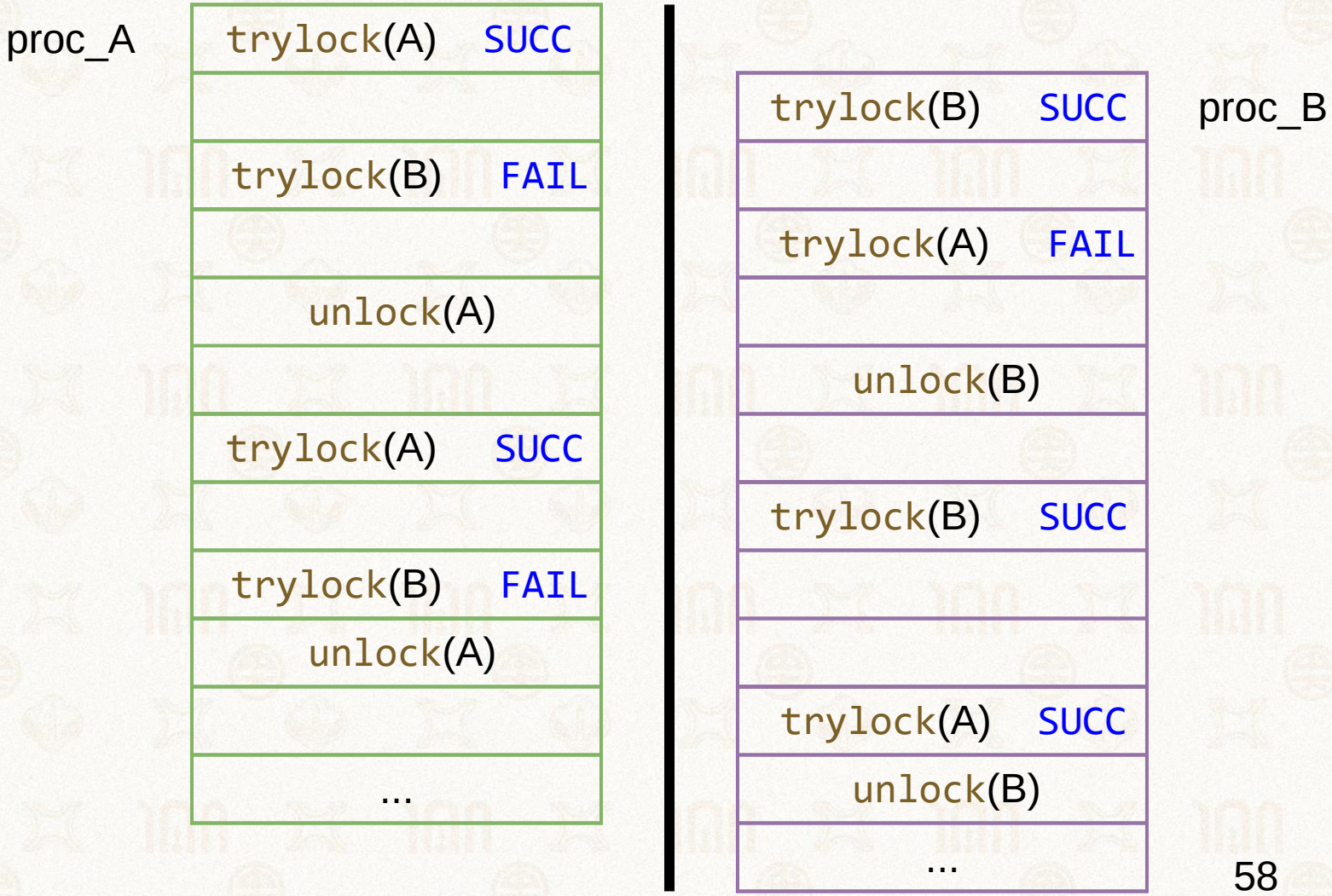
运气很差时可能出现如此往复，但运气不会一直这么差

活锁

- “不允许持有并等待”会产生活锁

```
while(true) {
    if(trylock(A) == SUCC) {
        if(trylock(B) == SUCC) {
            // 临界区代码
            // ...
            unlock(B);
            unlock(A);
            break;
        } else {
            unlock(A);
        }
    }
}
```

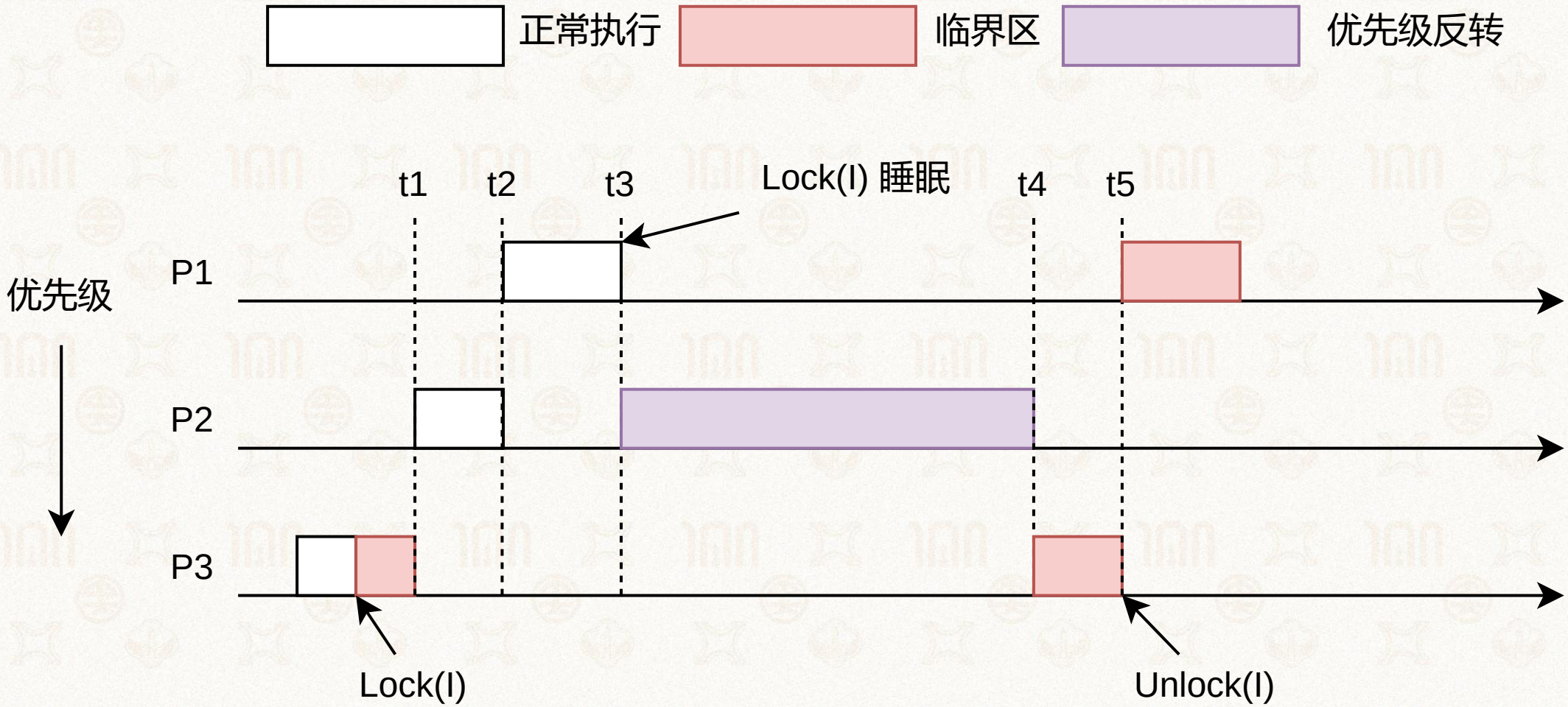
活锁大概率可自行消除



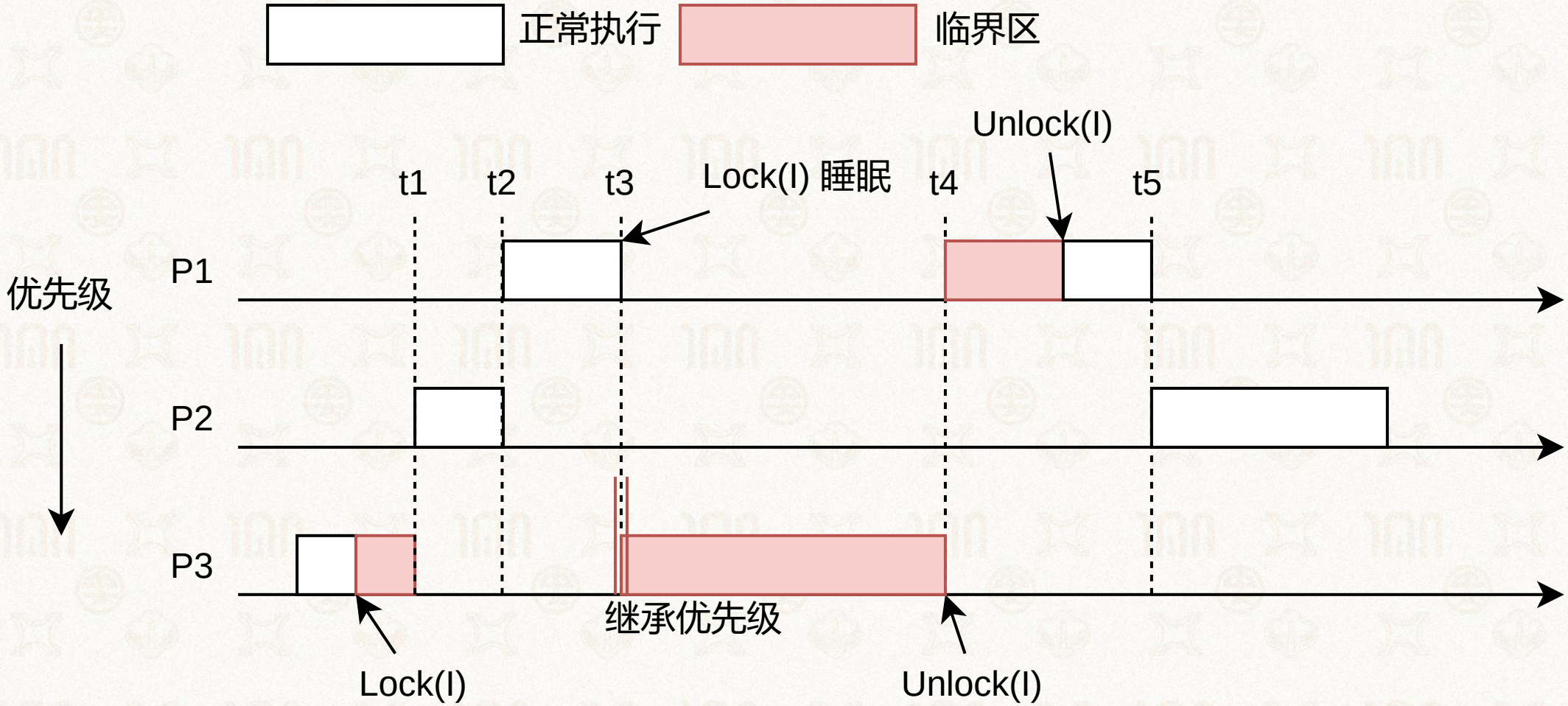
大纲

- 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

优先级反转


正常执行
临界区
优先级反转
t1
t2
t3
Lock(I) 睡眠
t4
t5
优先级
P1
P2
P3
Lock(I)
Unlock(I)

优先级继承协议


正常执行
临界区
Unlock(I)
t1
t2
t3
Lock(I) 睡眠
t4
t5
优先级
P1
P2
P3
继承优先级
Lock(I)
Unlock(I)

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

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

- 很快会见到

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