



中山大學

SUN YAT-SEN UNIVERSITY

软件工程学院

SCHOOL OF SOFTWARE ENGINEERING



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

同步原语: 互斥锁

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



- 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



进程间通信 (Inter-process Communication, IPC)

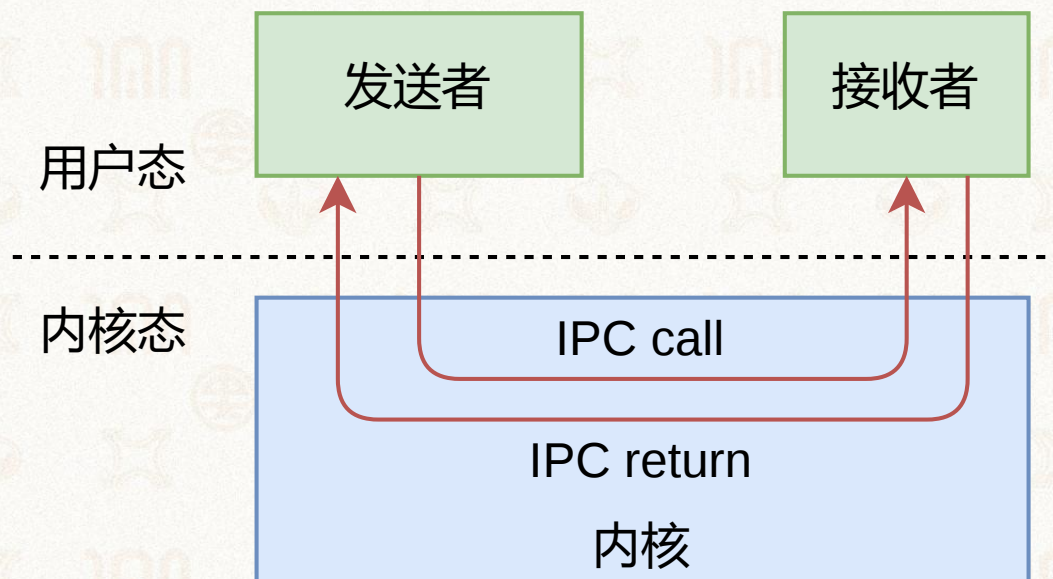


1924-2024
中山大學 世紀華誕
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

➤ 进程协作的达成依赖于进程间通信

➤ 进程间通信:

- 两个(或多个)不同的进程, 通过内核或其他共享资源进行通信, 来传递控制信息或数据
- 交互的双方: 发送者/接收者、客户端/服务端、调用者/被调用者
- 通信的内容一般叫做“消息”



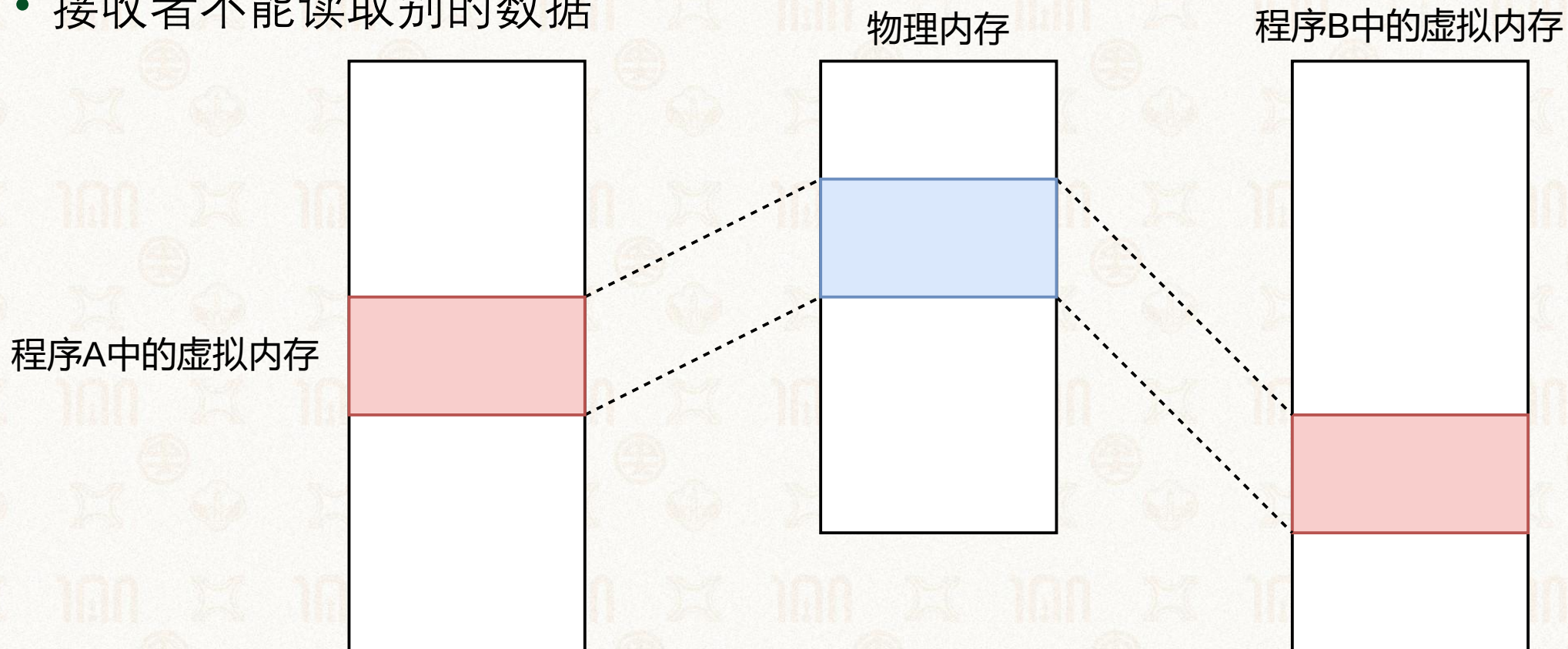


共享内存



1924-2024
中山大學 世紀华诞
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

- 系统内核为两个进程映射共同的内存区域
- 挑战: 做好同步
 - 发送者不能覆盖掉未读取的数据
 - 接收者不能读取别的数据





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

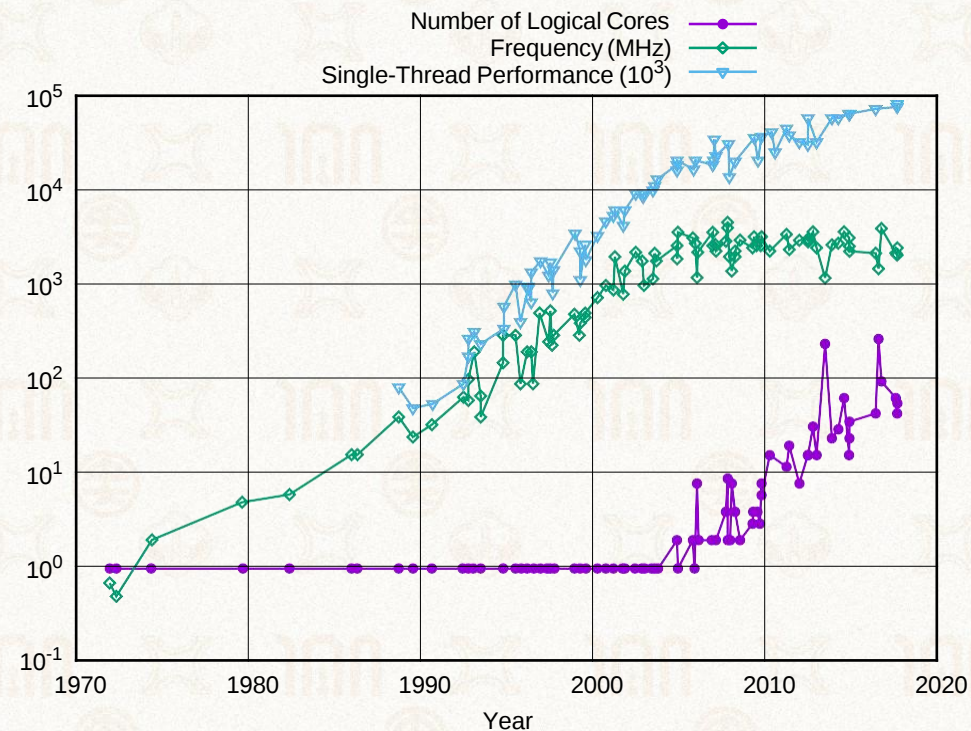


多处理器与多核



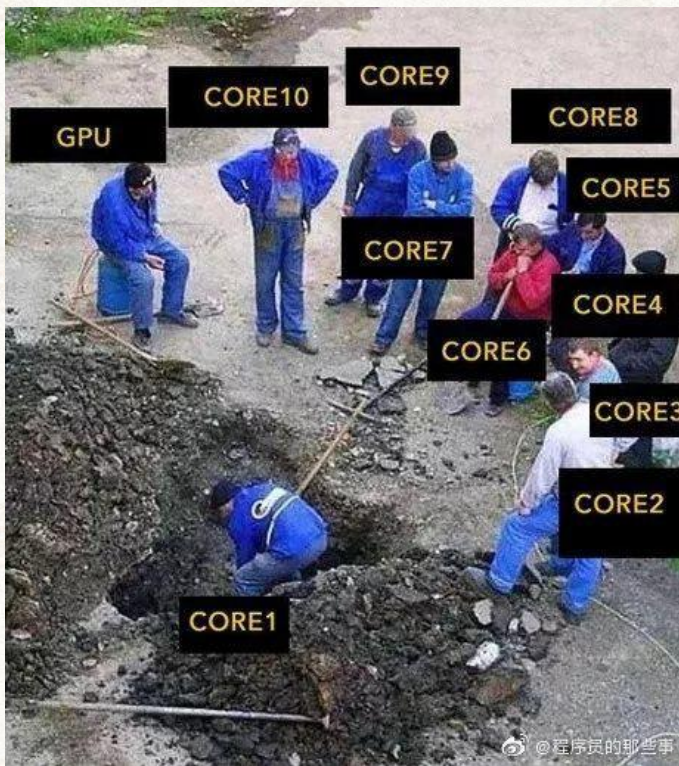
1924-2024
中山大學 世纪华诞
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

- 单核性能提升遇到瓶颈
- 不能通过一味提升频率来获得更好的性能
- 通过增加CPU核数来提升软件的性能
- 桌面/移动平台均向多核迈进





多核不是免费的午餐



网图：多核的真相

➤ 假设现在需要建房子：

- 工作量 = 1000人/年
- 工头找了10万人，需要多久？

➤ 面临的两个问题：

- 工人人多手杂，不听指挥，导致施工事故（**正确性**问题）
- 工具有限，大部分工人无事可干（**性能可扩展性**问题）



正确性问题：多线程计数器示例



```
volatile int balance = 0;
```

```
void *mythread(void *arg) {  
    int i;  
    for (i = 0; i < 20000; i++) {  
        balance++;  
    }  
    printf("Balance is % d\n", balance);  
    return NULL;  
}
```

```
int main(int argc, char *argv[]) {  
    pthread_t p1, p2, p3;  
    pthread_create(&p1, NULL, mythread, (void *)"A");  
    pthread_create(&p2, NULL, mythread, (void *)"B");  
    pthread_create(&p3, NULL, mythread, (void *)"C");  
    pthread_join(p1, NULL);  
    pthread_join(p2, NULL);  
    pthread_join(p3, NULL);  
    printf("Final Balance is % d\n", balance);  
}
```

➤ 为什么会出错？

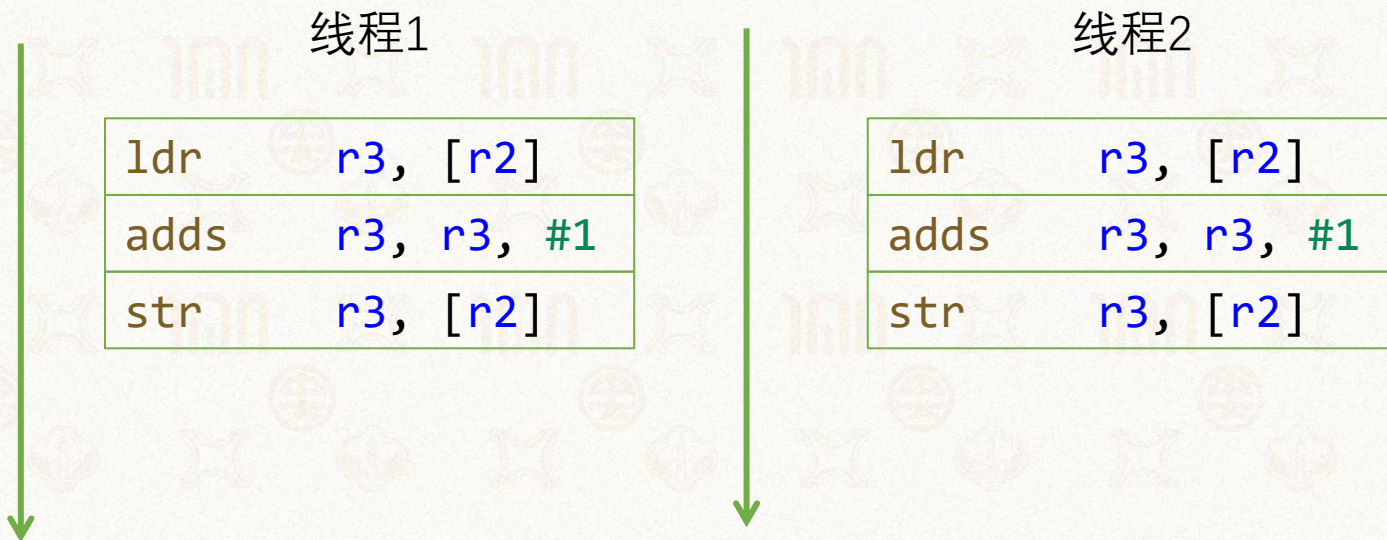


正确性问题：多线程计数器示例

```
volatile int balance = 0;
```

```
void *mythread(void *arg) {  
    int i;  
    for (i = 0; i < 20000; i++) {  
        balance++;  
    }  
    printf("Balance is % d\n", balance);  
    return NULL;  
}
```

- 分别位于两个CPU中运行：
- 错误：应该加2，但实际上只加了1





正确性问题：多线程计数器示例



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

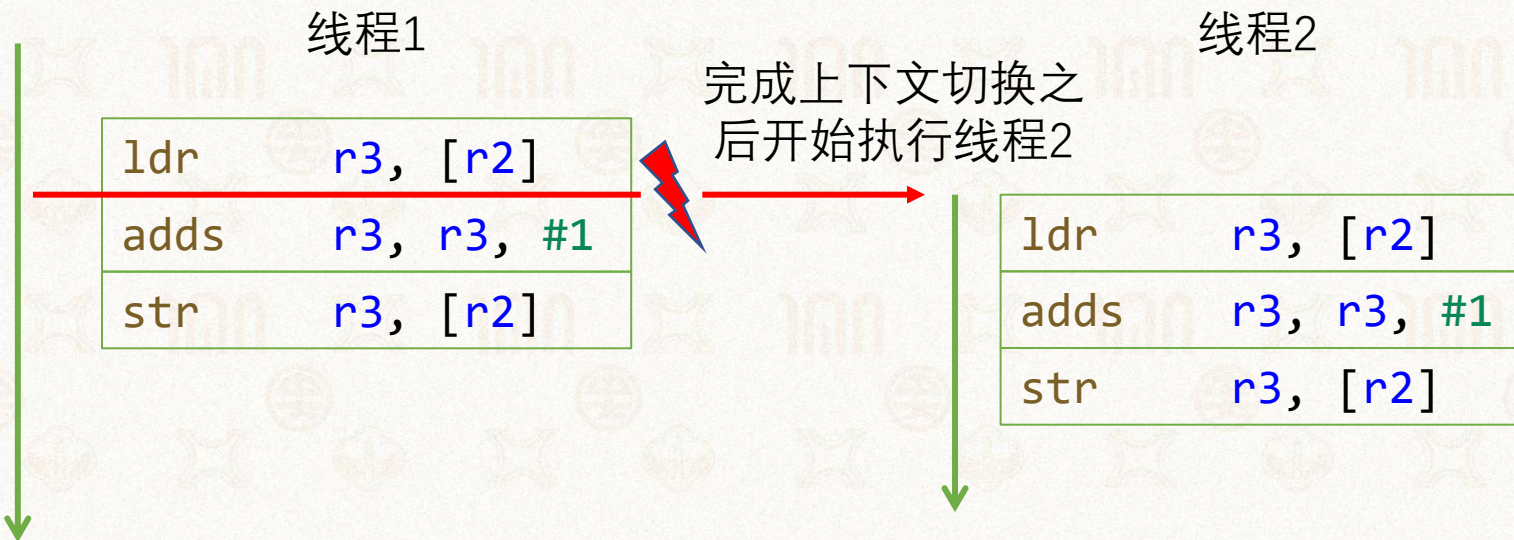
```
volatile int balance = 0;
```

```
void *mythread(void *arg) {  
    int i;  
    for (i = 0; i < 20000; i++) {  
        balance++;  
    }  
    printf("Balance is %d\n", balance);  
    return NULL;  
}
```

➤ 在同一个CPU中运行：

- 错误：应该加2，但实际上只加了1

在此时发生
抢占式调度





正确性问题：多线程计数器示例

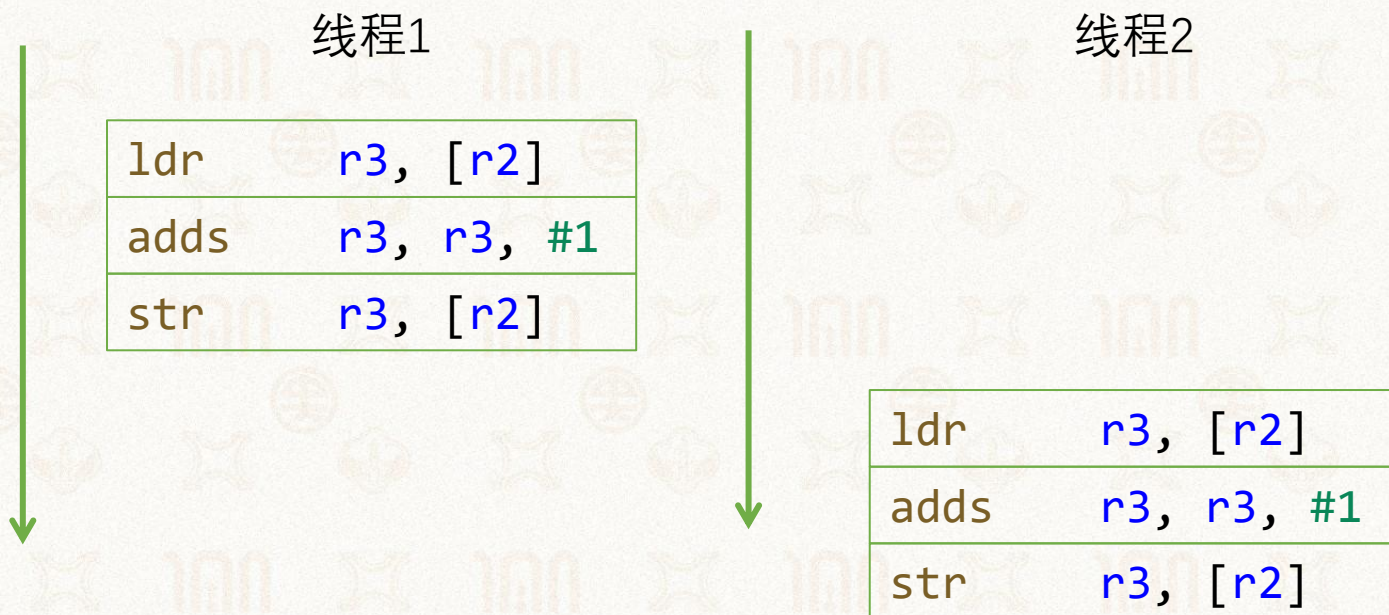


1924-2024
中山大學 世紀華誕
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

```
volatile int balance = 0;
```

```
void *mythread(void *arg) {  
    int i;  
    for (i = 0; i < 20000; i++) {  
        balance++;  
    }  
    printf("Balance is % d\n", balance);  
    return NULL;  
}
```

- 分别位于两个CPU中运行：
- 正确：[r2]地址处的变量加2





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
- 之后再讨论



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

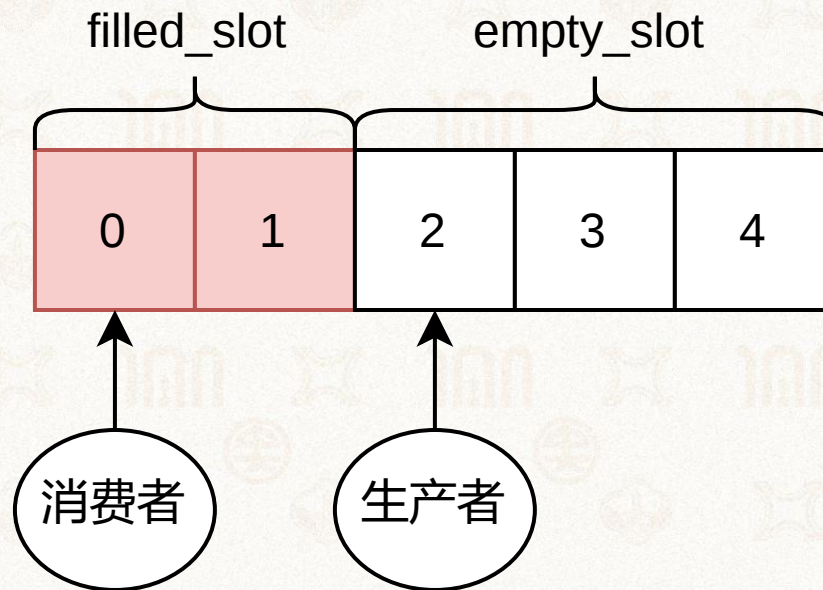


生产者消费者问题：单生产者、单消费者



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

```
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);
    }
}
```





生产者消费者问题：缓冲区操作

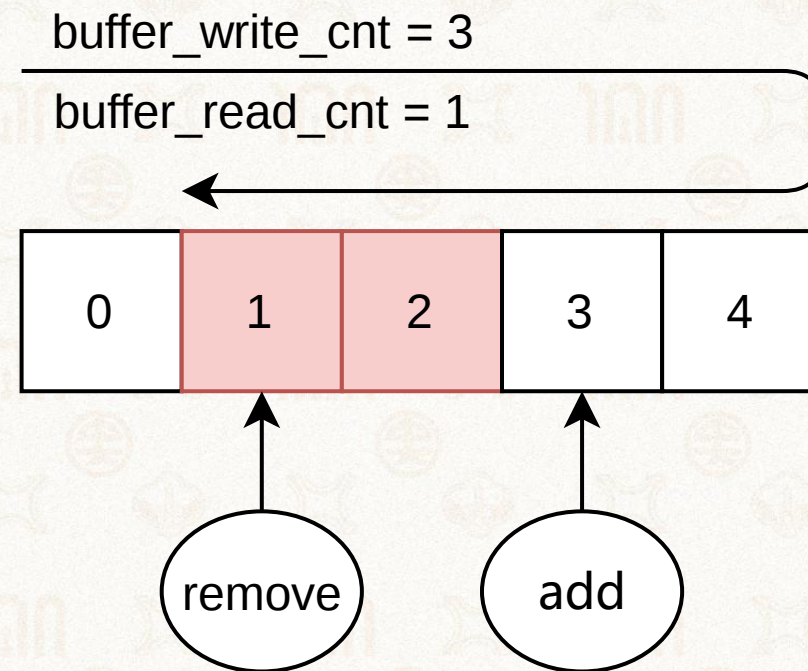


```
volatile int buffer_write_cnt = 0; // 共享的
volatile int buffer_read_cnt = 0; // 共享的

int buffer[5]; // 共享的

void buffer_add(int msg) {
    buffer[buffer_write_cnt] = msg;
    buffer_write_cnt = (buffer_write_cnt + 1) % 5;
}

int buffer_remove(void) {
    int ret = buffer[buffer_read_cnt];
    buffer_read_cnt = (buffer_read_cnt + 1) % 5;
    return ret;
}
```

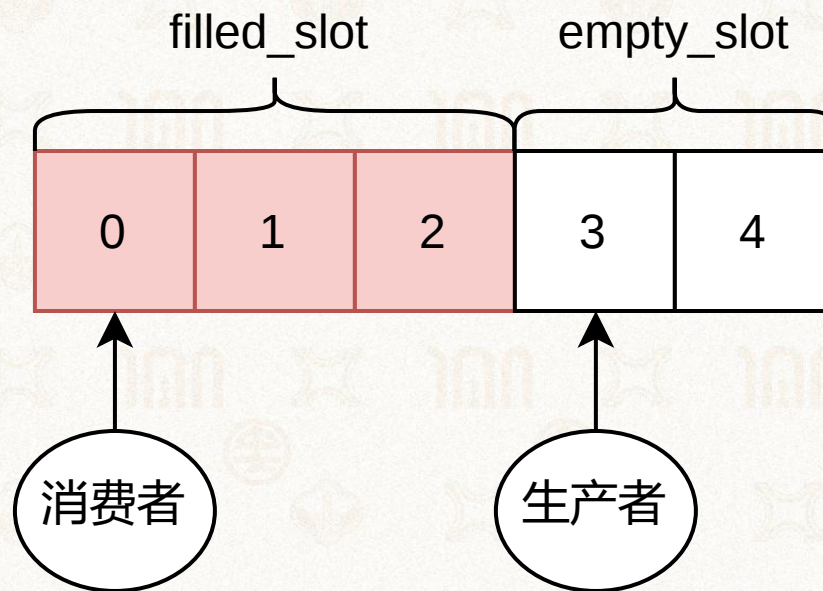




生产者消费者问题：单生产者、单消费者



```
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);
    }
}
```



生产者新产生一个消息

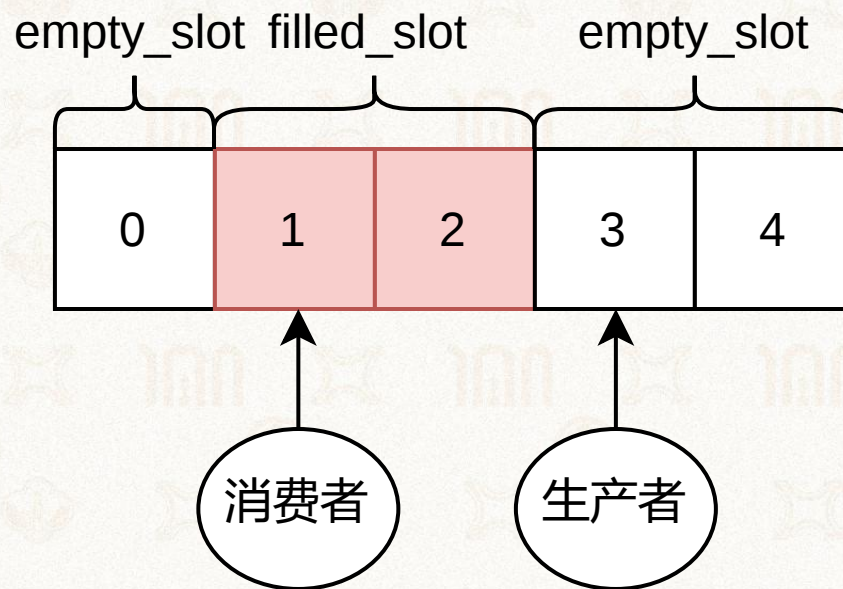


生产者消费者问题：单生产者、单消费者



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

```
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);
    }
}
```



消费者消耗一个消息



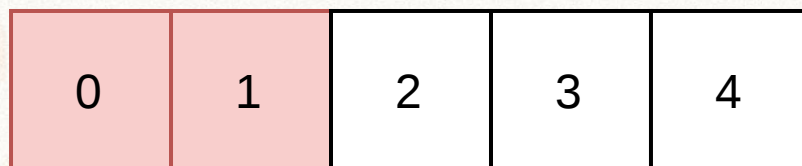
生产者消费者问题：多生产者、单消费者



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

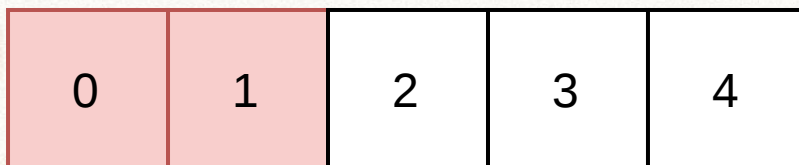
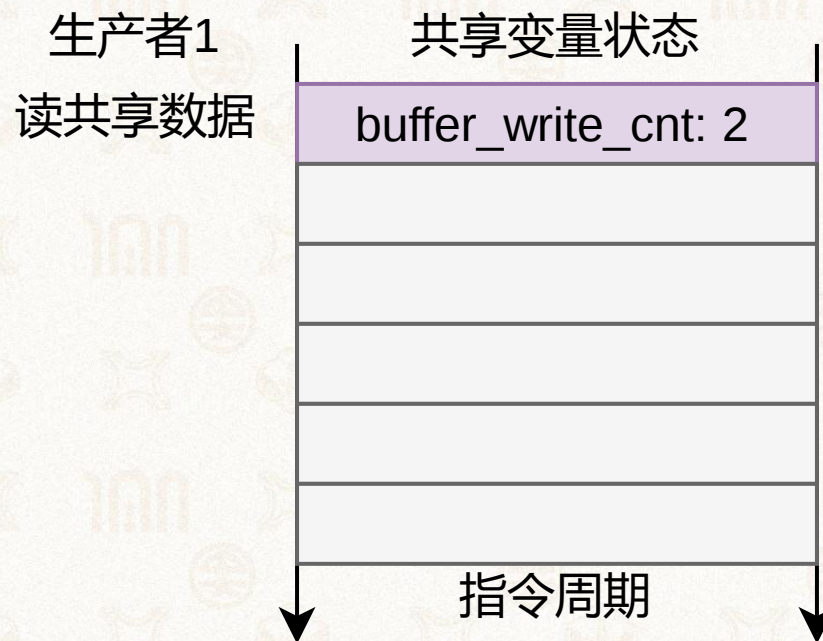
生产者1

共享变量状态





生产者消费者问题：多生产者、单消费者

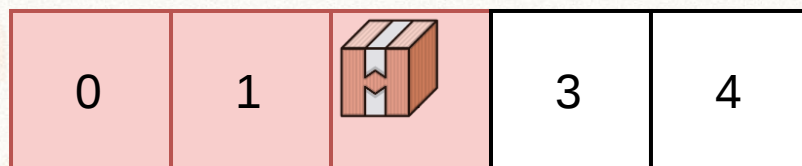
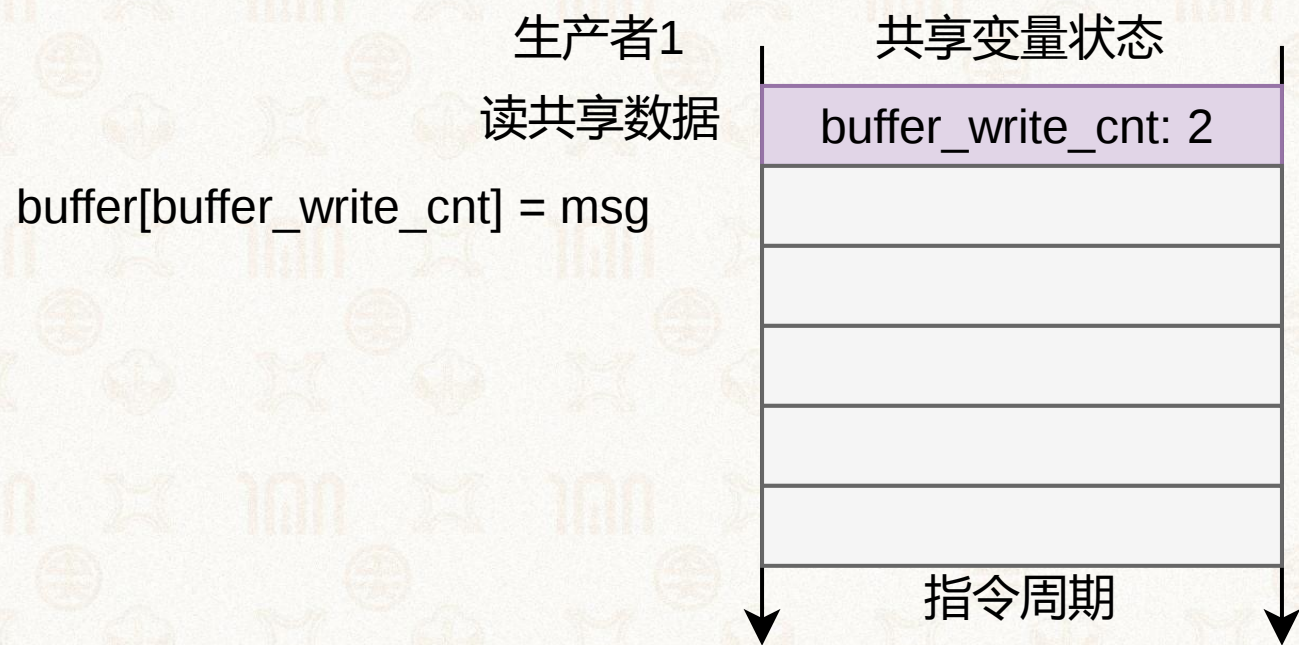




生产者消费者问题：多生产者、单消费者



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

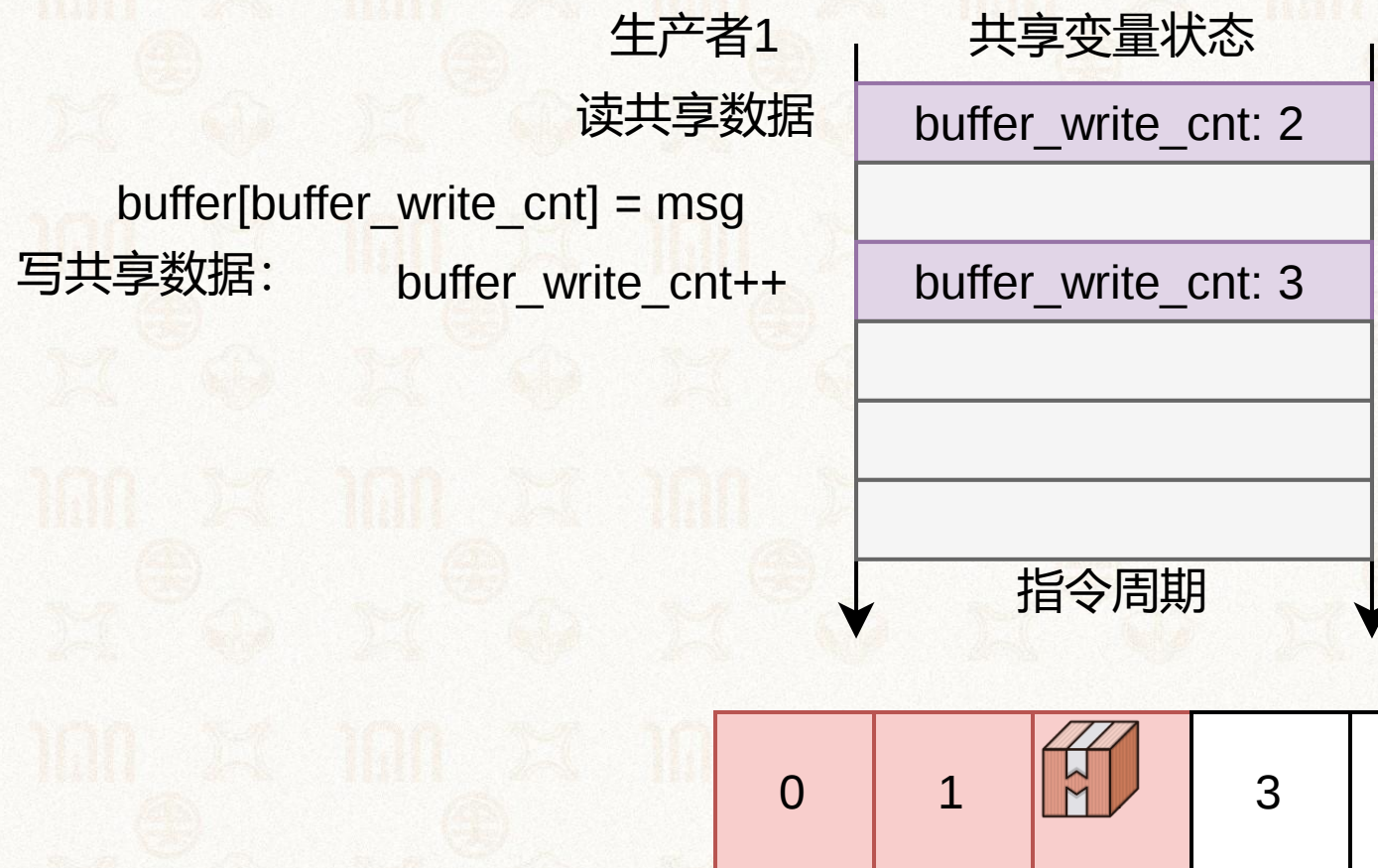




生产者消费者问题：多生产者、单消费者

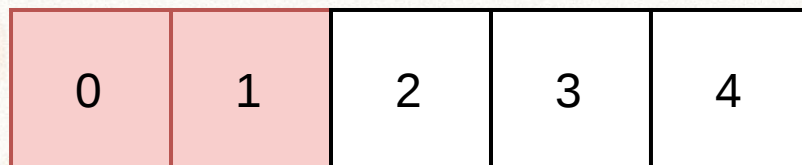
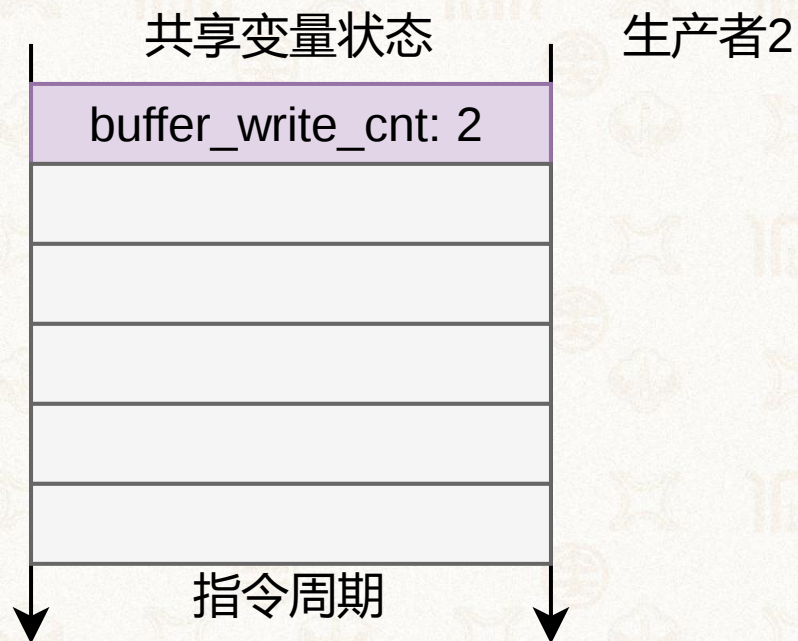


1924-2024
中山大学 世纪华诞
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY



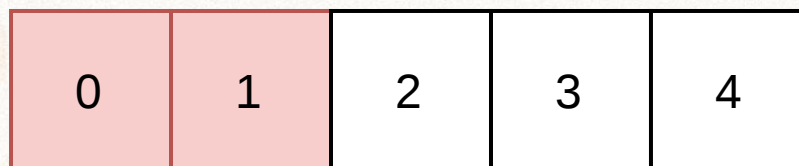
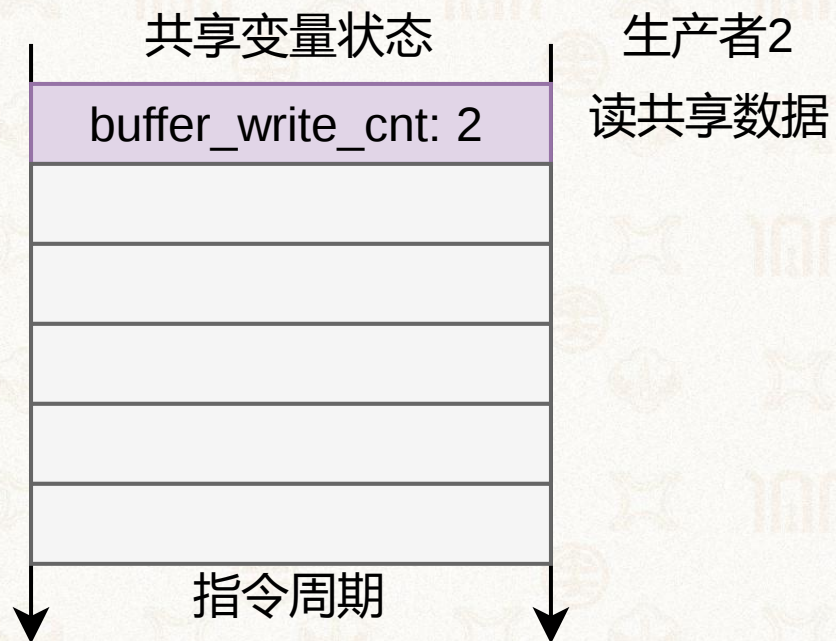


生产者消费者问题：多生产者、单消费者





生产者消费者问题：多生产者、单消费者

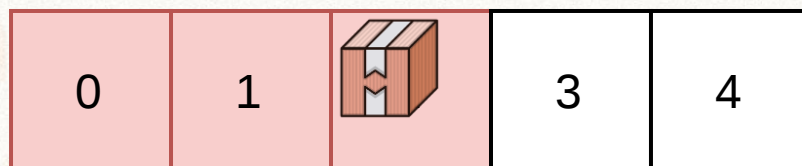
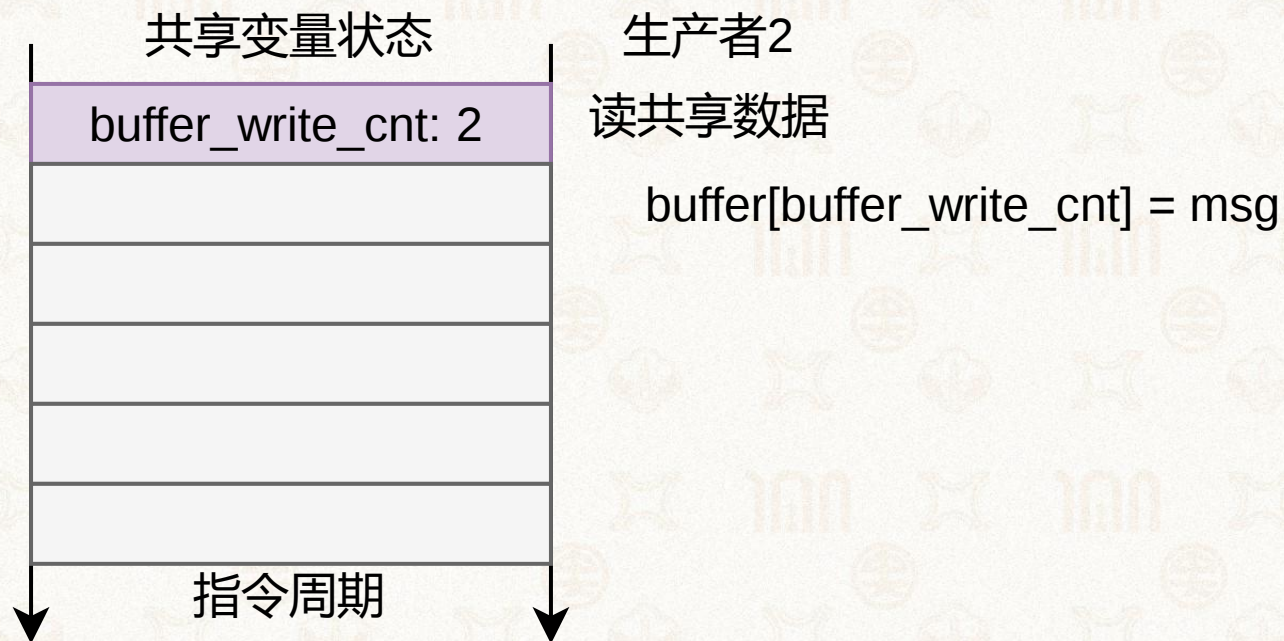




生产者消费者问题：多生产者、单消费者

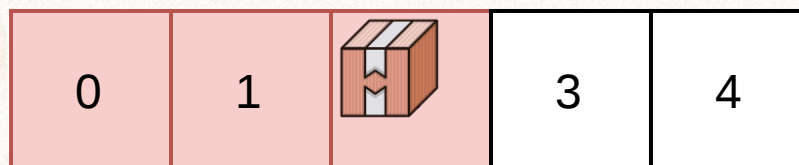
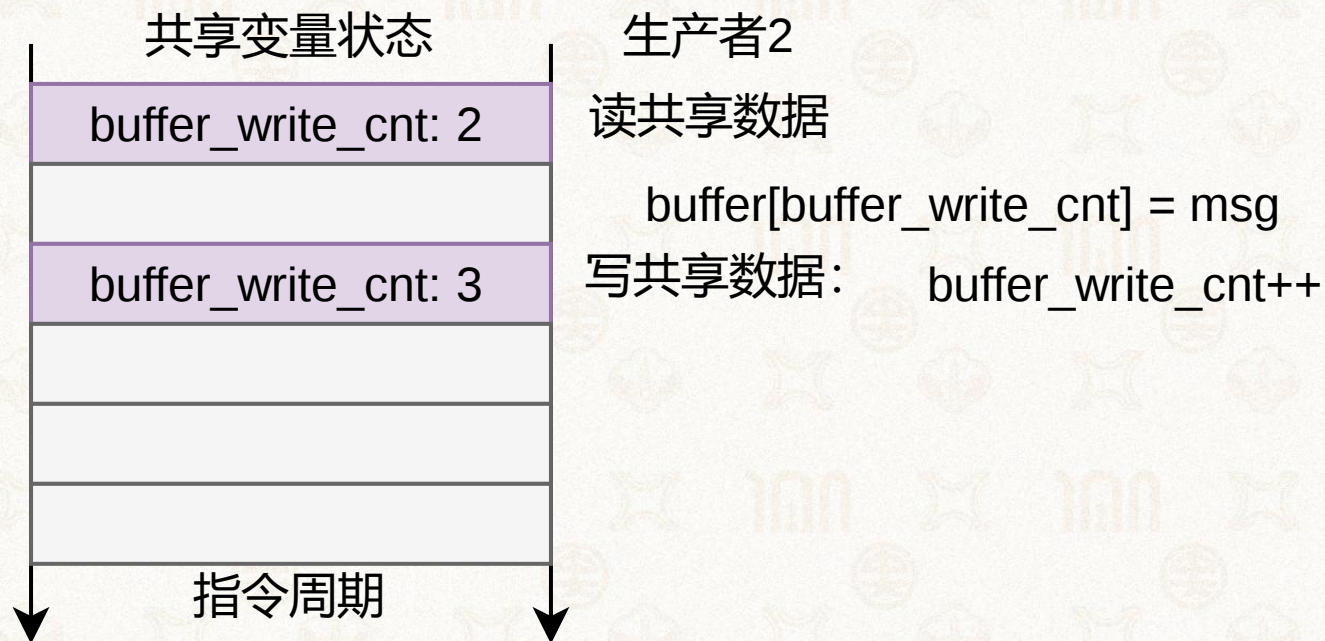


1924-2024
中山大學 世紀华诞
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY



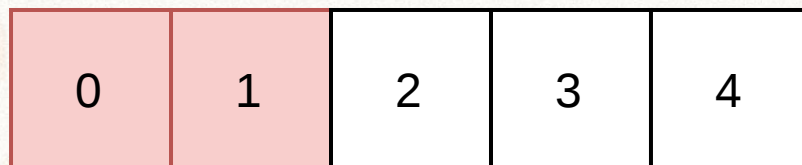


生产者消费者问题：多生产者、单消费者



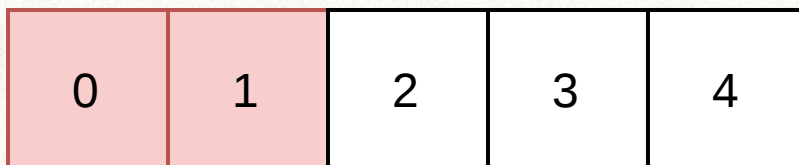
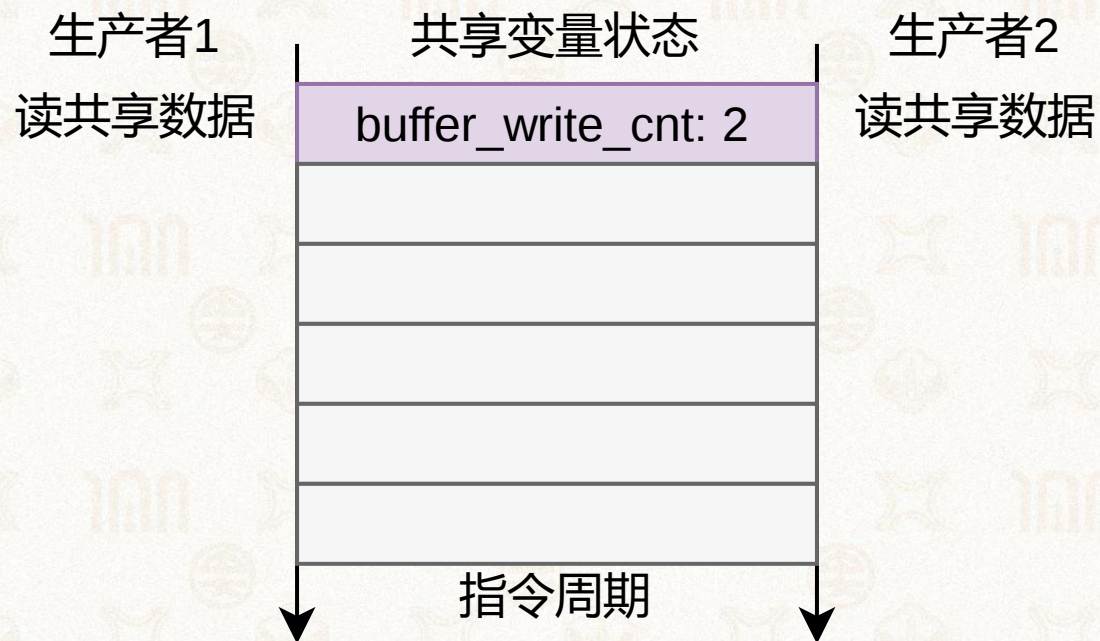


生产者消费者问题：多生产者、单消费者





生产者消费者问题：多生产者、单消费者



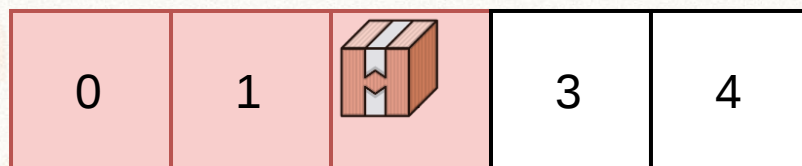
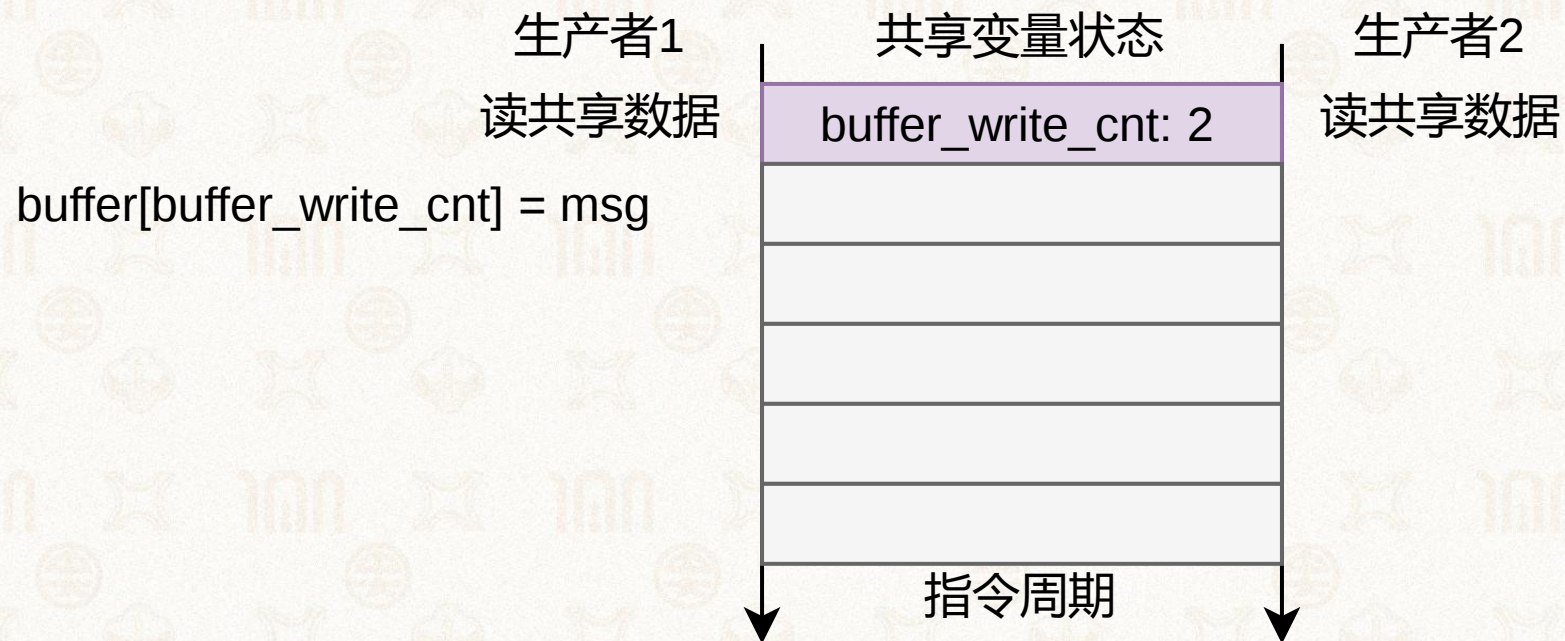
同时将数据读入寄存器



生产者消费者问题：多生产者、单消费者



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



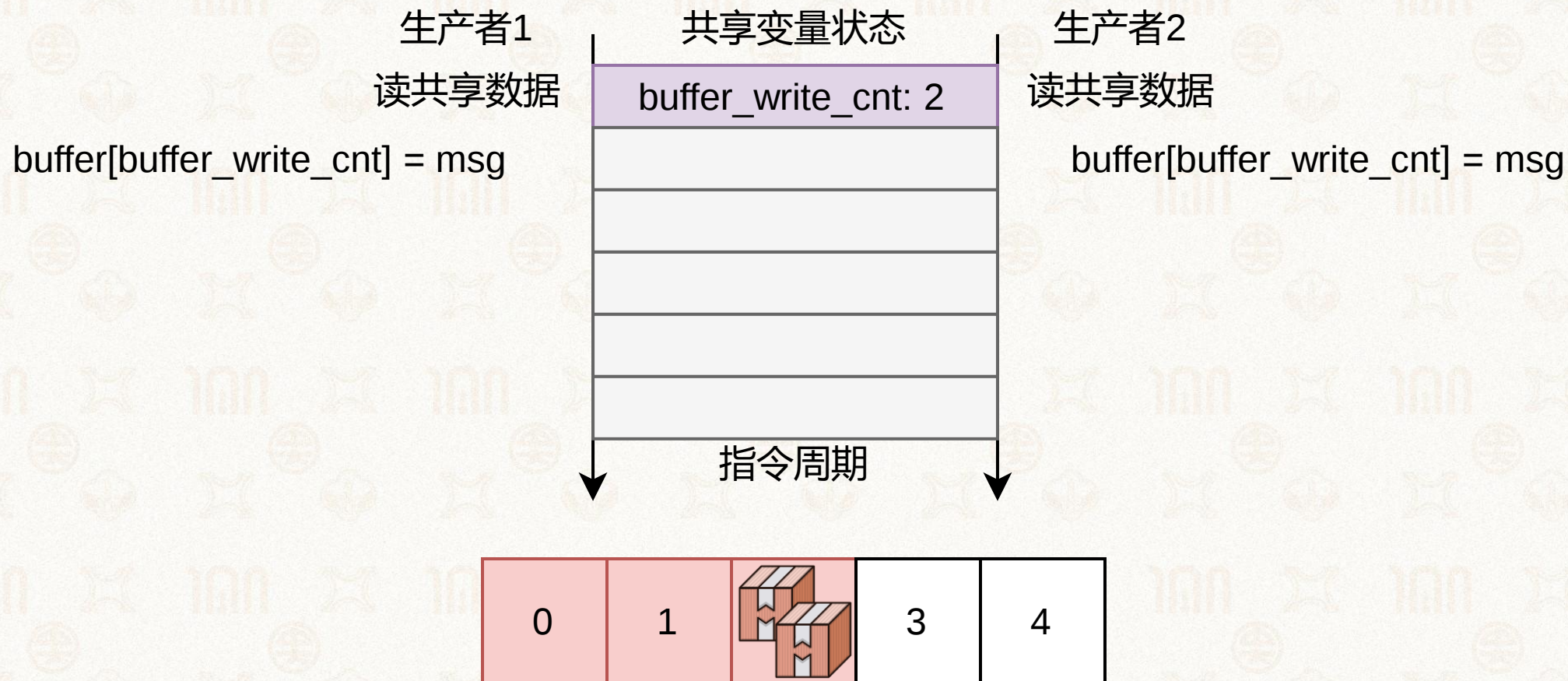
生产者1先写入数据



生产者消费者问题：多生产者、单消费者



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



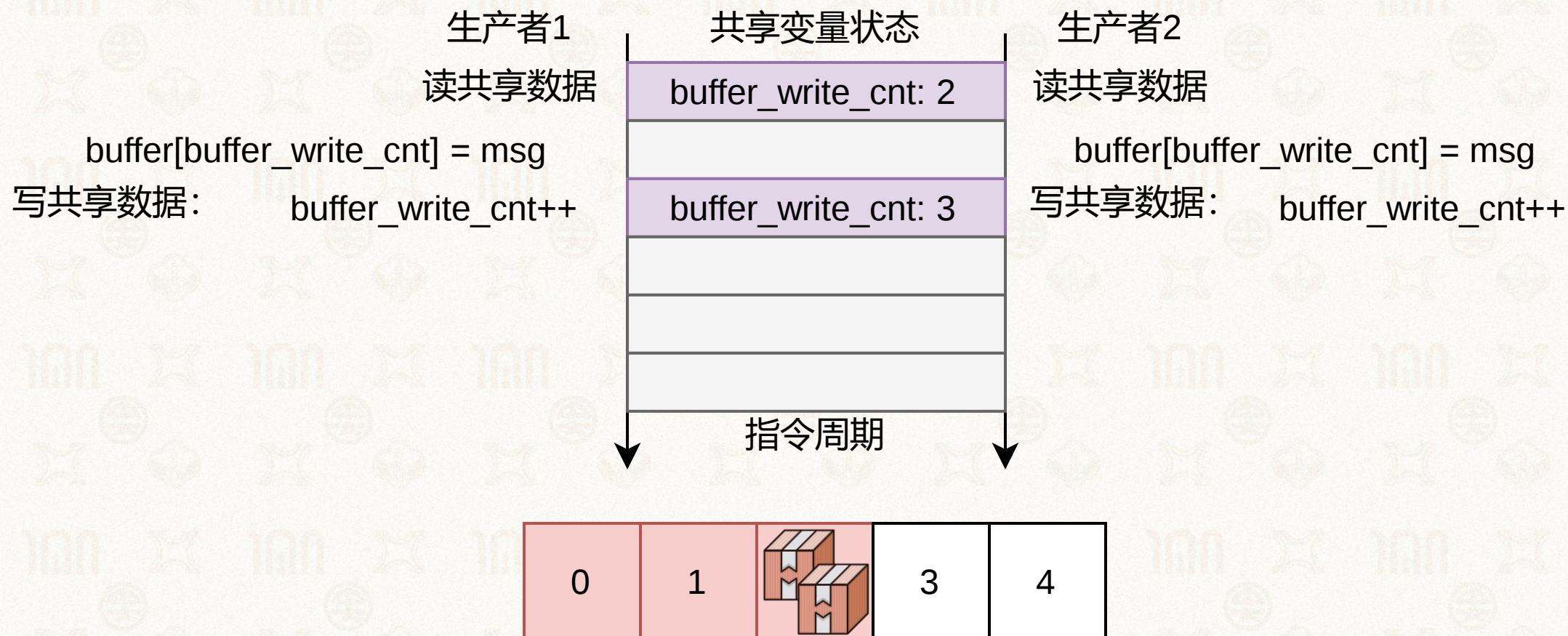
生产者2也在同样位置写数据，造成数据覆盖



生产者消费者问题：多生产者、单消费者



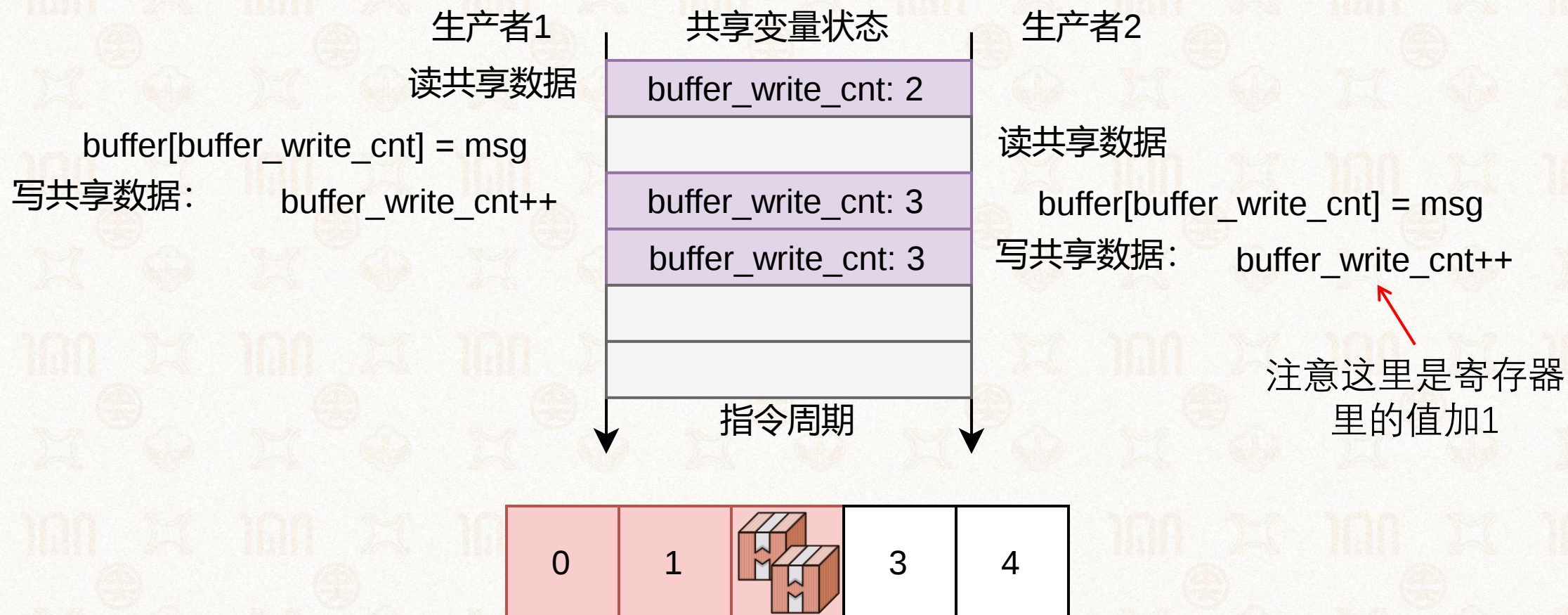
1924-2024
中山大学 世纪华诞
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY



不仅是数据覆盖，连计数都不正确了



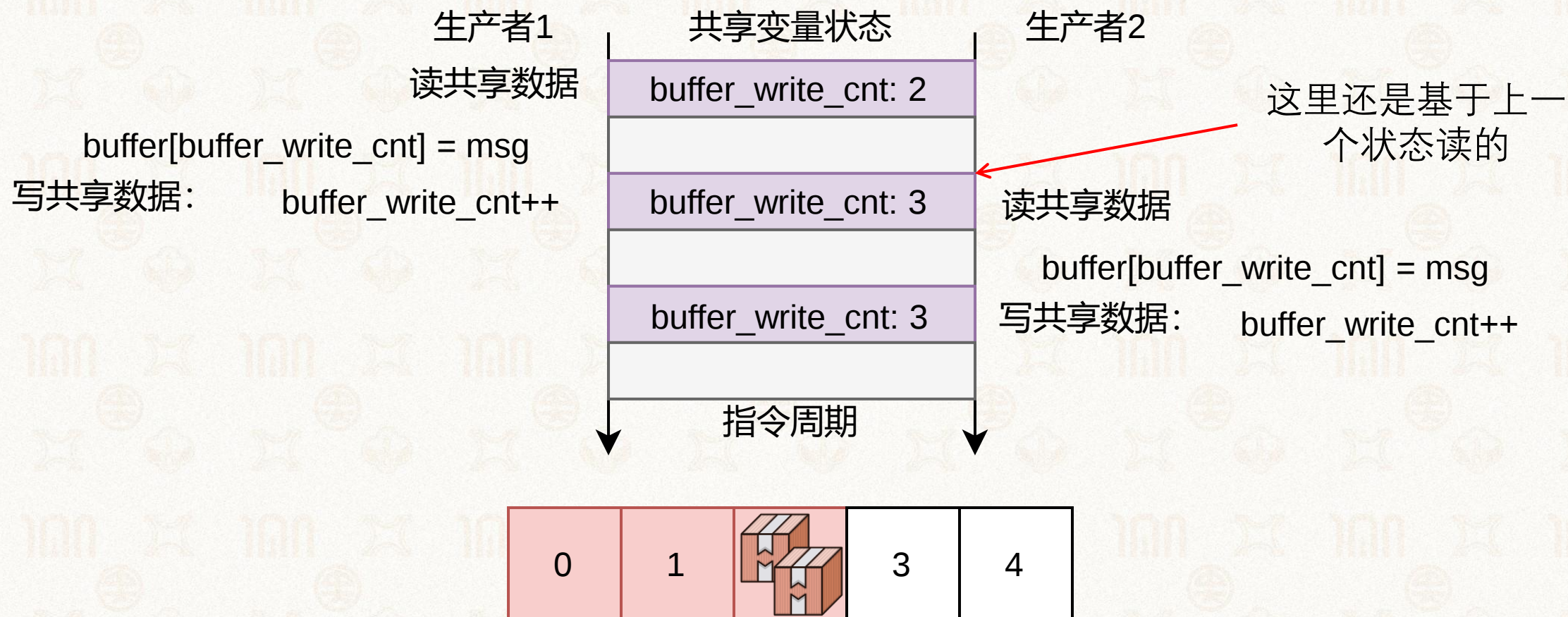
生产者消费者问题：多生产者、单消费者



即便生产者2慢一些，程序还是不正确



生产者消费者问题：多生产者、单消费者



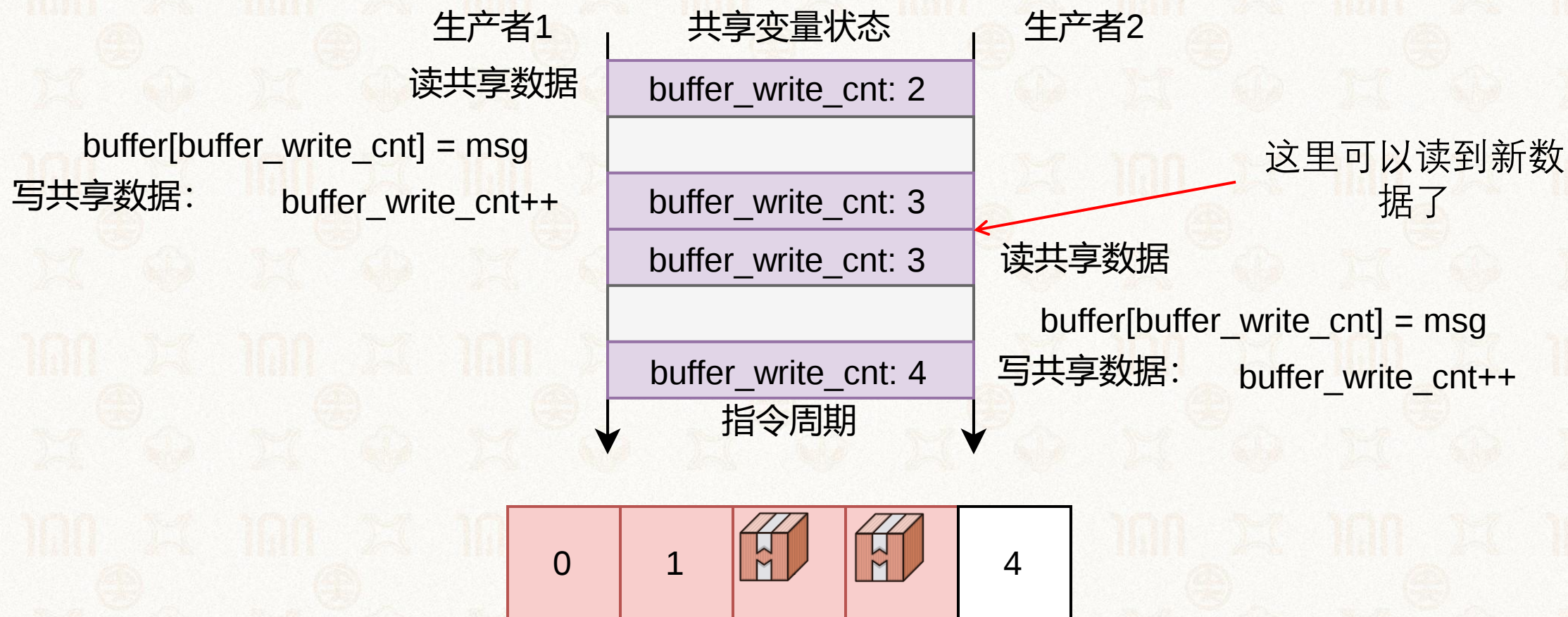
即便生产者2慢一些，程序还是不正确



生产者消费者问题：多生产者、单消费者



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

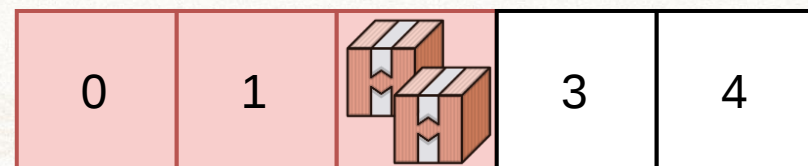


生产者2慢到顺序执行的地步，程序终于正确了

➤ 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（竞争冒险、竞态条件）：

- 当多个线程同时对共享的数据进行操作
- 该共享数据最后的结果**依赖于**这些进程**特定的执行顺序**





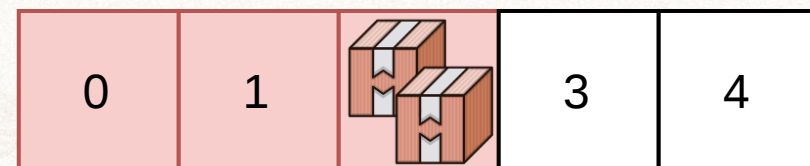
竞争条件 Race Condition



➤ 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（竞争冒险、竞态条件）：

- 当多个线程同时对共享的数据进行操作
- 该共享数据最后的结果**依赖**于这些进程**特定的执行顺序**





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



临界区问题



➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

同学A举手：我想去卫生间

申请进入临界区

老师：可以，去吧

临界区部分

标示退出临界区

其它代码

```
}
```



临界区问题



➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

申请进入临界区

同学A在卫生间

临界区部分

标示退出临界区

其它代码

```
}
```



临界区问题



➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

同学B举手：我想去卫生间

申请进入临界区

老师：不行，有人在卫生间还没回来

同学A在卫生间

临界区部分

标示退出临界区

其它代码

```
}
```



临界区问题



➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

同学C举手：我想去卫生间

申请进入临界区

老师：不行，有人在卫生间还没回来

同学A在卫生间

临界区部分

标示退出临界区

其它代码

```
}
```



临界区问题



➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

同学B举手：我急！

申请进入临界区

老师：不行，等着！

同学A在卫生间

临界区部分

标示退出临界区

其它代码

```
}
```



临界区问题

➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

同学C举手：我也急！

申请进入临界区

老师：不行，等着！

同学A在卫生间

临界区部分

标示退出临界区

其它代码

```
}
```



临界区问题

➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

同学C：好吧…

同学B：好的！

申请进入临界区

老师： A回来了， B你去吧， C再等等

临界区部分

同学A回来了

标示退出临界区

其它代码

```
}
```



临界区问题



➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

申请进入临界区

同学B在卫生间

临界区部分

标示退出临界区

其它代码

```
}
```



临界区问题



➤ 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？

以考试时多人想去卫生间为例：

```
while (TRUE) {
```

申请进入临界区

临界区部分

同学B回来了

标示退出临界区

其它代码

```
}
```



解决临界区问题的三个要求

➤ **互斥**访问(mutual exclusion): 同一时刻, 最多只有一个线程进入临界区

```
while (TRUE) {
```

同学B举手: 我急!

申请进入临界区

老师: **不行**, 等着!

同学A在卫生间

临界区部分

标示退出临界区

其它代码

```
}
```



解决临界区问题的三个要求

- 空闲让进：没有线程在临界区时，必须**选一个**线程允许其进入临界区

```
while (TRUE) {
```

同学C：好吧…

同学B：好的！

申请进入临界区

老师：A回来了，**B你去吧**，C再等等

临界区部分

同学A回来了

标示退出临界区

其它代码

```
}
```



解决临界区问题的三个要求



- 有限等待：当线程申请进入临界区时，必须在有限时间内获得许可

```
while (TRUE) {
```

同学B举手：我急！

申请进入临界区

老师：等一会！如果A五分钟后不回来，就让你去

同学A在卫生间

临界区部分

标示退出临界区

其它代码

```
}
```



解决临界区问题的算法：关闭硬件中断

```
while (TRUE) {
```

申请进入临界区

关闭中断

临界区部分

标示退出临界区

开启中断

其它代码

```
}
```

- 单核下，线程是按时间片调度的，只要关了中断，就能保证临界区只有一个程序
- 但在多核下，不能阻塞其它核的线程执行
 - 不能阻止多个CPU核同时进入临界区



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



解决临界区问题的算法：皮特森算法



- flag[] 对应位为TRUE, 表示该线程申请进入临界区
- turn 裁决谁可以进入临界区, 数字1表示线程1进, 0表示线程0进

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    turn = 1;  
    while (flag[1] == TRUE && turn == 1);
```

临界区部分

```
flag[0] = FALSE;
```

其它代码

```
}
```

线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] == TRUE && turn == 0);
```

临界区部分

```
flag[1] = FALSE;
```

其它代码

```
}
```



解决临界区问题的算法：皮特森算法



- turn: 为何要如此谦让?
- 竞争式的写法

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0; ←  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```





解决临界区问题的算法：皮特森算法



- turn: 为何要如此谦让?
- 竞争式的写法

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

先执行

共享变量状态

turn: 1

turn: 0

后执行

线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0; ←  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```

临界区部分

其它代码

指令周期



解决临界区问题的算法：皮特森算法



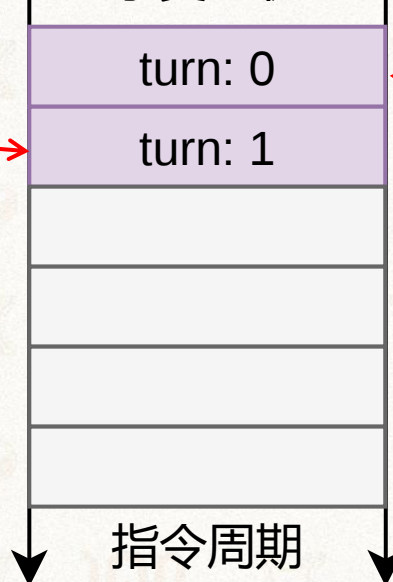
- turn: 为何要如此谦让?
- 无论如何, turn总是在0、1二选一

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

后执行

共享变量状态



线程1

```
while (TRUE) {  
    先执行 flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```



解决临界区问题的算法：皮特森算法



➤ 毕竟有寄存器读取局部变量的情况，turn二选一就安全了？

线程0

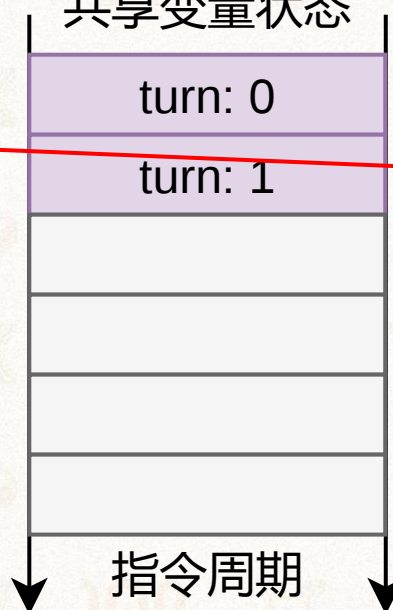
```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

线程0被线程1读取

线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0; ←  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```

共享变量状态



指令周期

可能出现红色交叉

不可能出现红色交叉

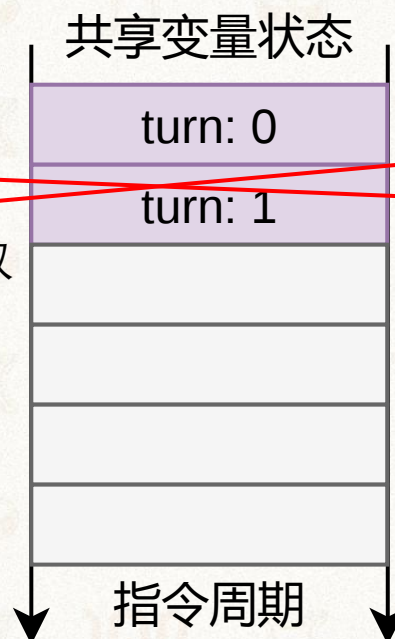
线程0

线程1

```
while (TRUE) {
    flag[0] = TRUE;
    → turn = 1;
    while (flag[1] == TRUE && turn == 1);
    临界区部分
    flag[0] = FALSE;
    其它代码
}
```

线程0写线程1读取

线程1写线程0读取



```
while (TRUE) {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] == TRUE && turn == 0);
    临界区部分
    flag[1] = FALSE;
    其它代码
}
```

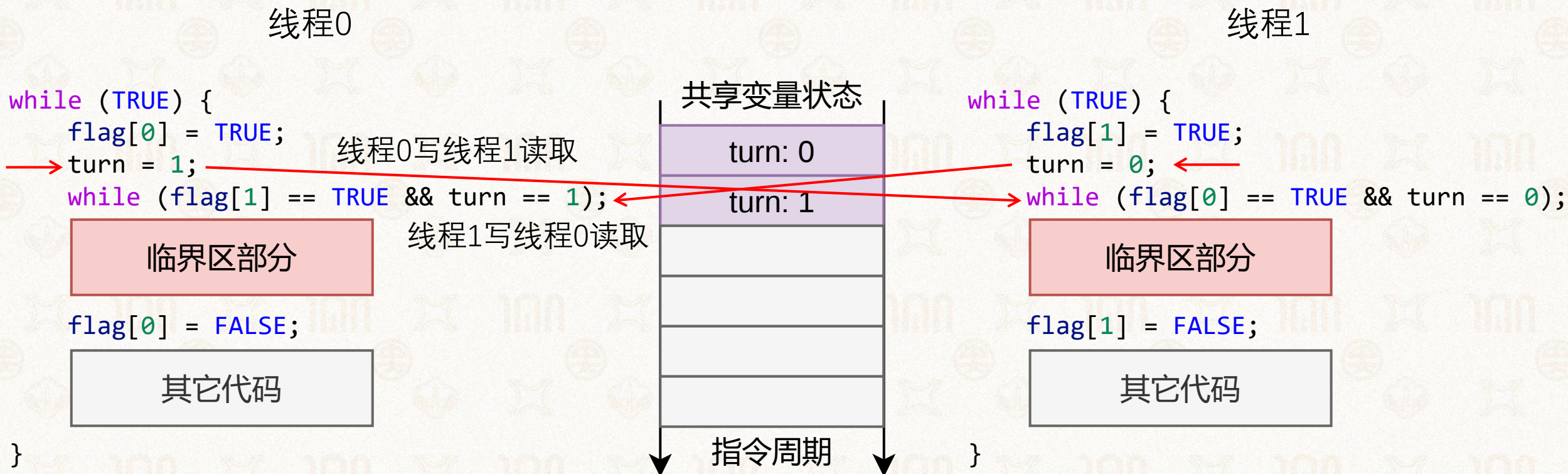
提交



解决临界区问题的算法：皮特森算法



- 毕竟有寄存器读取局部变量的情况，turn二选一就安全了？
- 一旦两条红线出现交叉，那每个线程都认为可以自己进入临界区了





解决临界区问题的算法：皮特森算法



- 为什么不会出现红线交叉？
- 线程0写线程1读取只出现在线程0后写的情况下

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

线程0写线程1读取

共享变量状态

turn: 0

turn: 1

指令周期

线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```

临界区部分

其它代码



解决临界区问题的算法：皮特森算法

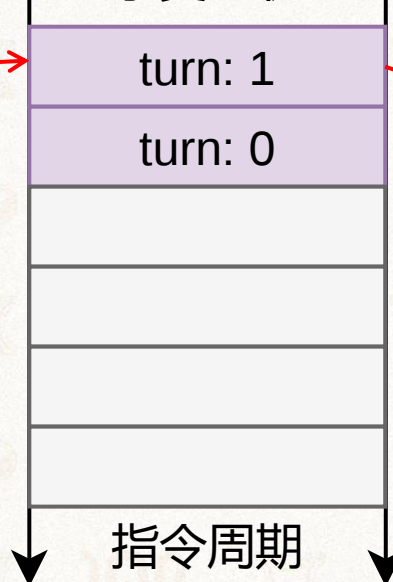


- 为什么不会出现红线交叉？
- 线程0写线程1读取只出现在线程0后写的情况下
- 为什么不能是线程0先写？

线程0

```
while (TRUE) {  
    flag[0] = TRUE;    线程0写线程1读取  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

共享变量状态



线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```

前一句写还没执行，后一句读先被执行了？



解决临界区问题的算法：皮特森算法

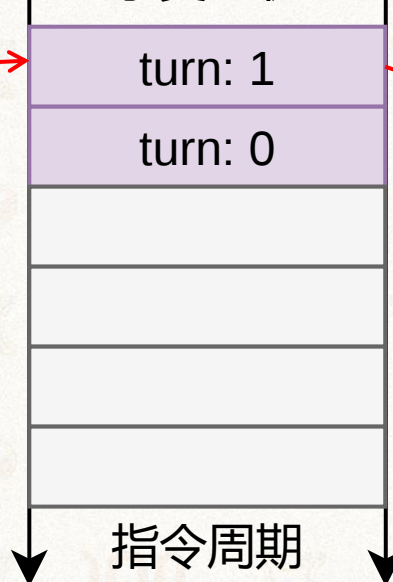


- 为什么不会出现红线交叉？
- 线程0写线程1读取只出现在线程0后写的情况下
- 为什么不能是线程0先写？（因为线程1内部顺序不能乱！）

线程0

```
while (TRUE) {  
    flag[0] = TRUE;    线程0写线程1读取  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

共享变量状态



线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```



解决临界区问题的算法：皮特森算法



- 为什么不会出现红线交叉？
- 线程0写线程1读取只出现在线程0后写的情况下
 - 此时线程1先写的数据已经被覆盖，传不到线程0 (线程0内部的读写顺序不能乱!)

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

共享变量状态

turn: 0

turn: 1

指令周期

线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```



解决临界区问题的算法：皮特森算法



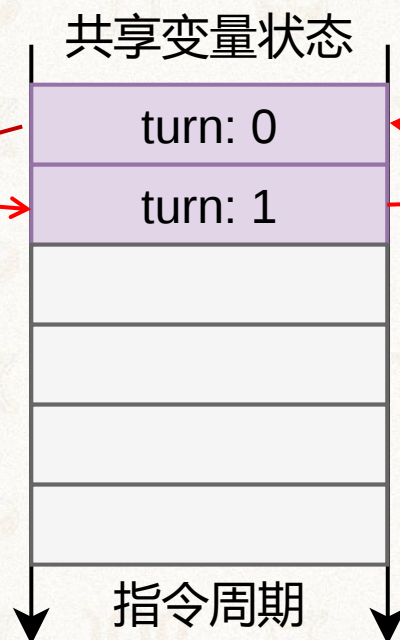
- 所以不会出现红线交叉
- 所以turn二选一是安全的
- 所以turn实现了互斥访问

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```





解决临界区问题的算法：皮特森算法



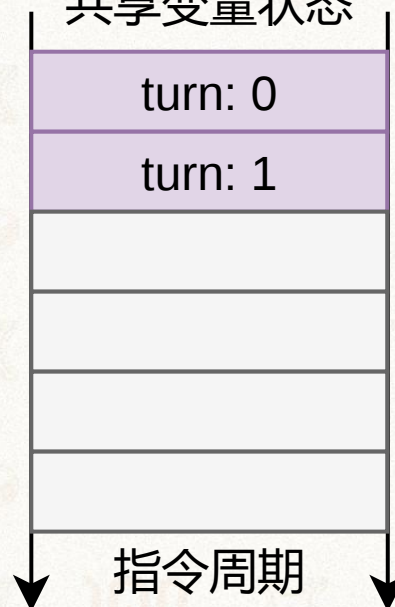
➤ flag与turn结合，同样实现了有限等待和空闲让进

- 由于互相谦让，所以没有“卡死”的地方

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

共享变量状态



线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0; ←  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```



皮特森算法的特点

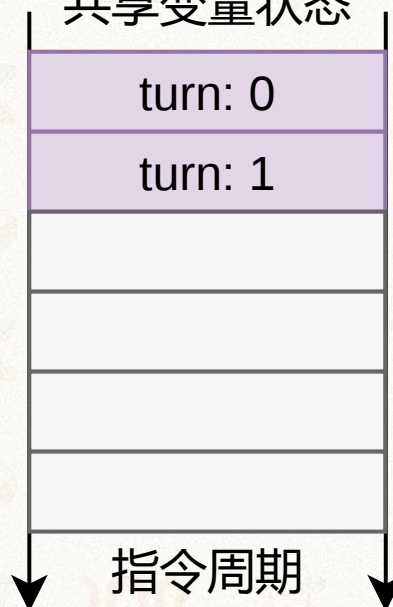


- 原始算法只适用于两个线程，但后来有人扩展了一下，可以适用任意多线程
- 要求读取操作严格按程序顺序执行，
 - 但现代CPU有乱序执行机制，皮特森算法会失效

线程0

```
while (TRUE) {  
    flag[0] = TRUE;  
    → turn = 1;  
    while (flag[1] == TRUE && turn == 1);  
    临界区部分  
    flag[0] = FALSE;  
    其它代码  
}
```

共享变量状态



线程1

```
while (TRUE) {  
    flag[1] = TRUE;  
    turn = 0; ←  
    while (flag[0] == TRUE && turn == 0);  
    临界区部分  
    flag[1] = FALSE;  
    其它代码  
}
```



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

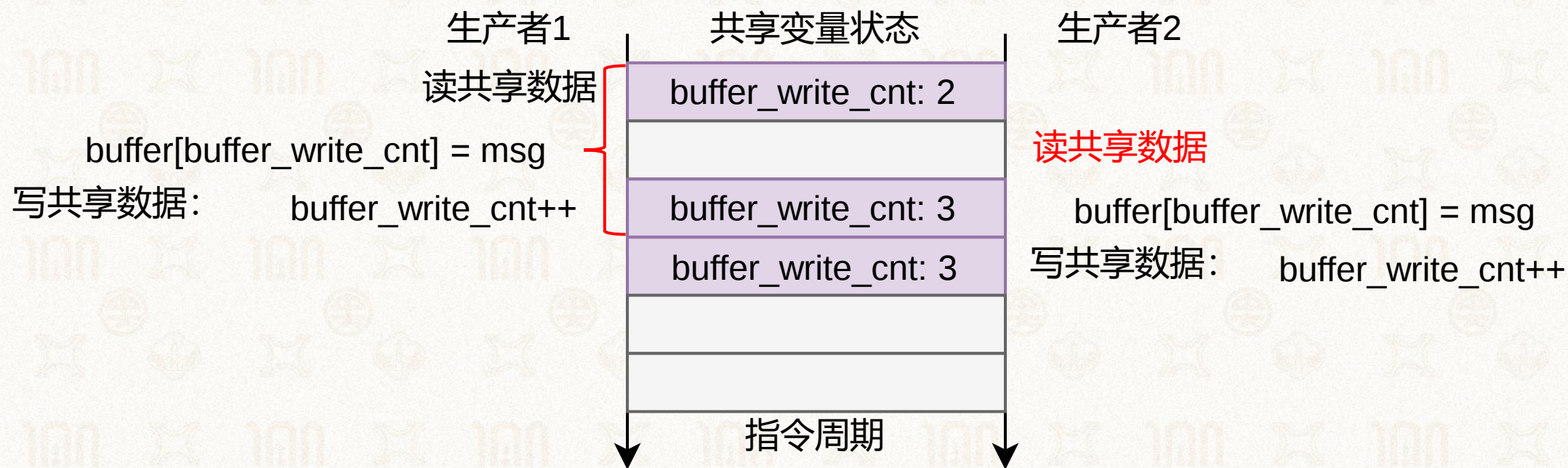


解决临界区问题的算法：原子操作



- 软硬结合的方法
- 原子操作：不可被打断的一个或一系列操作

错误的根本原因是生产者1的读写过程被打断了！





常见的原子操作：比较与置换

➤ Compare-And-Swap, CAS

- 由一条汇编语言完成

这条汇编语言等价的含义：

```
int CAS(int *addr, int expected, int new_value) {  
    int tmp = *addr;  
    if(*addr == expected)  
        *addr = new_value;  
    return tmp;  
}
```

```
while (TRUE) {
```

申请进入临界区

```
while(CAS(lock, 0, 1) != 0)  
; // 死循环等待
```

加锁操作: lock

临界区部分

标示退出临界区

```
*lock = 0;
```

解锁操作: unlock

其它代码

```
}
```



常见的原子操作：拿取并累加

➤ 拿取并累加(Fetch-And-Add, FAA)

- 由一条汇编语言完成

这条汇编语言等价的含义：

```
int FAA(int *addr, int add_value) {  
    int tmp = *addr;  
    *addr = tmp + add_value;  
    return tmp;  
}
```

假如不是一条汇编语言，
而是普通C语言函数：

程序发生错误，这些C语言代码只表示逻辑，本身并不是原子操作

`int FAA(&var, 2) {` 生产者1

<code>int tmp = *addr // tmp = 10</code>
<code>*addr = tmp + add_value;</code>
<code>return tmp;</code>
<code>}</code>

共享变量状态

var: 10
var: 11
var: 12

指令周期

`int FAA(&var, 1) {` 生产者2

<code>int tmp = *addr // tmp = 10</code>
<code>*addr = tmp + add_value;</code>
<code>return tmp;</code>
<code>}</code>



硬件原子操作

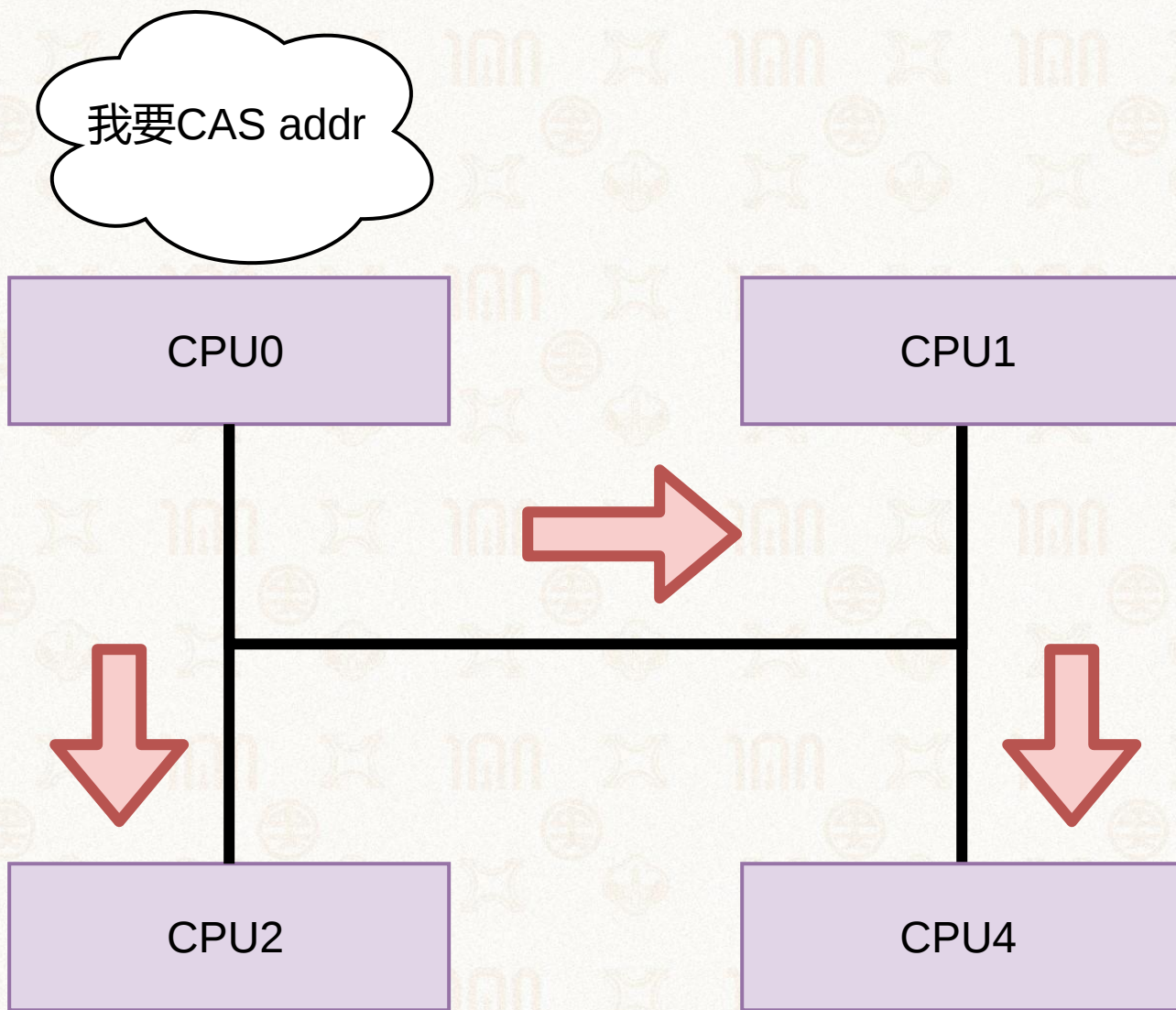


1924-2024
中山大學 世纪华诞
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

- 无论是CAS还是FAA，需要硬件支持才是真的原子操作
- 真的原子操作应该：
 - 不可被打断的操作集合
 - 如同执行一条指令
 - 其他CPU核心不会看到中间状态 all-or-nothing



硬件原子操作：Intel锁总线实现



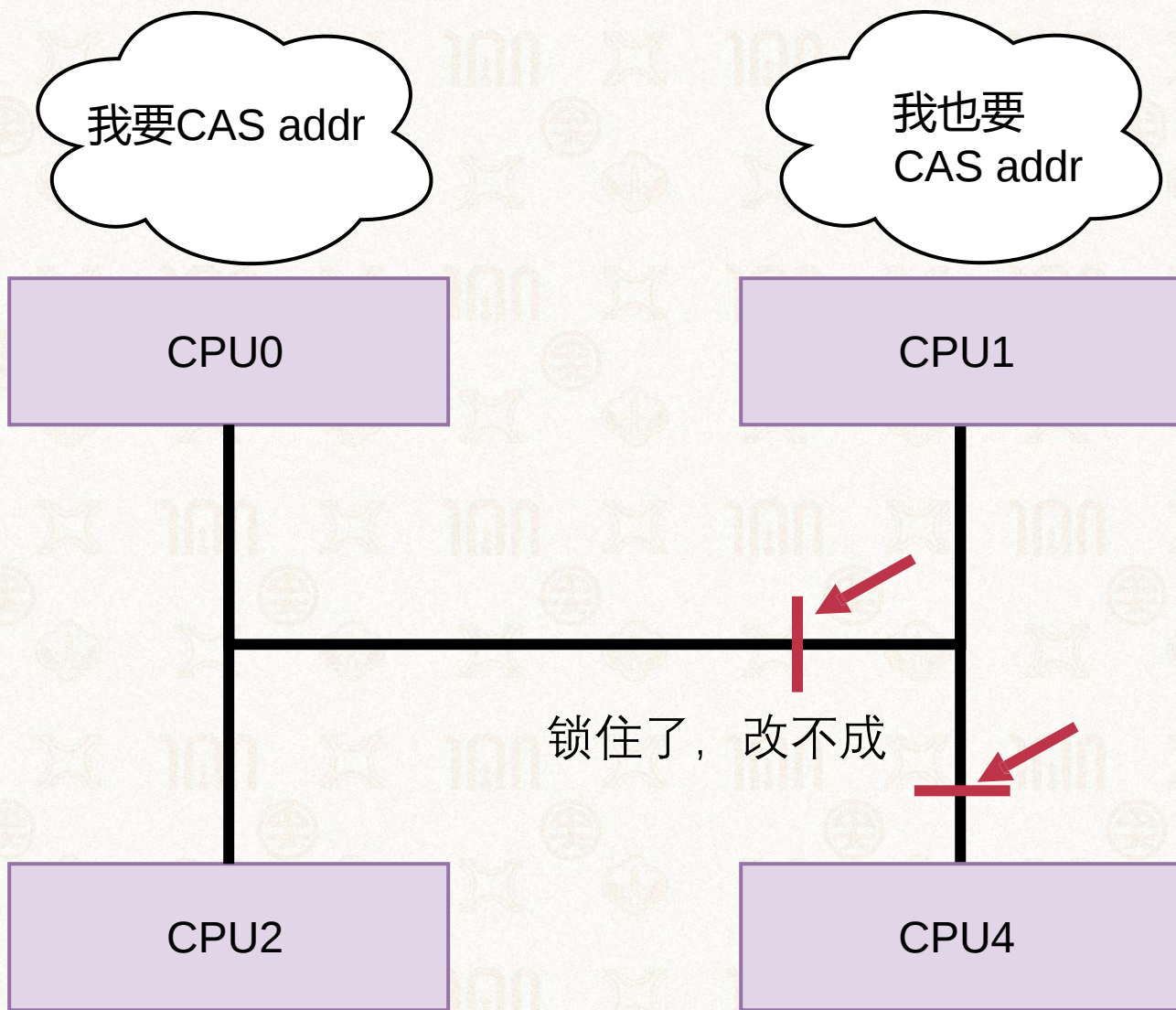
```
int CAS(int *addr, int expected, int new_value) {  
    int tmp = *addr;  
    if(*addr == expected)  
        *addr = new_value;  
    return tmp;  
}
```

CPU 0 独占

- 对任意地址的修改都要经过总线
- 通过**锁总线**来实现原子操作



硬件原子操作：Intel锁总线实现



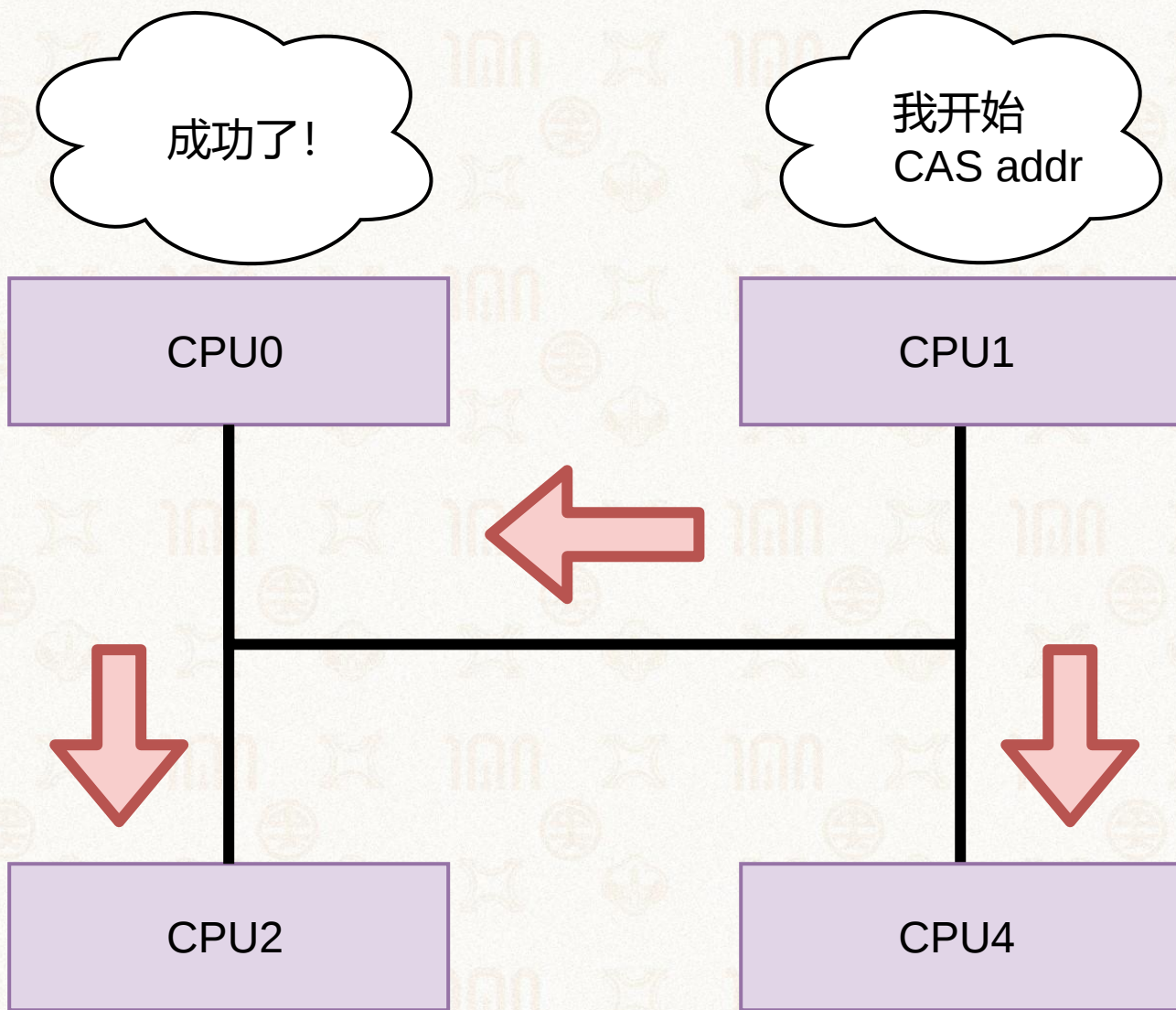
```
int CAS(int *addr, int expected, int new_value) {  
    int tmp = *addr;  
    if(*addr == expected)  
        *addr = new_value;  
    return tmp;  
}
```

CPU 0 独占

- 对任意地址的修改都要经过总线
- 通过锁总线来实现原子操作



硬件原子操作：Intel锁总线实现



```
int CAS(int *addr, int expected, int new_value) {  
    int tmp = *addr;  
    if(*addr == expected)  
        *addr = new_value;  
    return tmp;  
}
```

CPU 1 独占

CPU 0

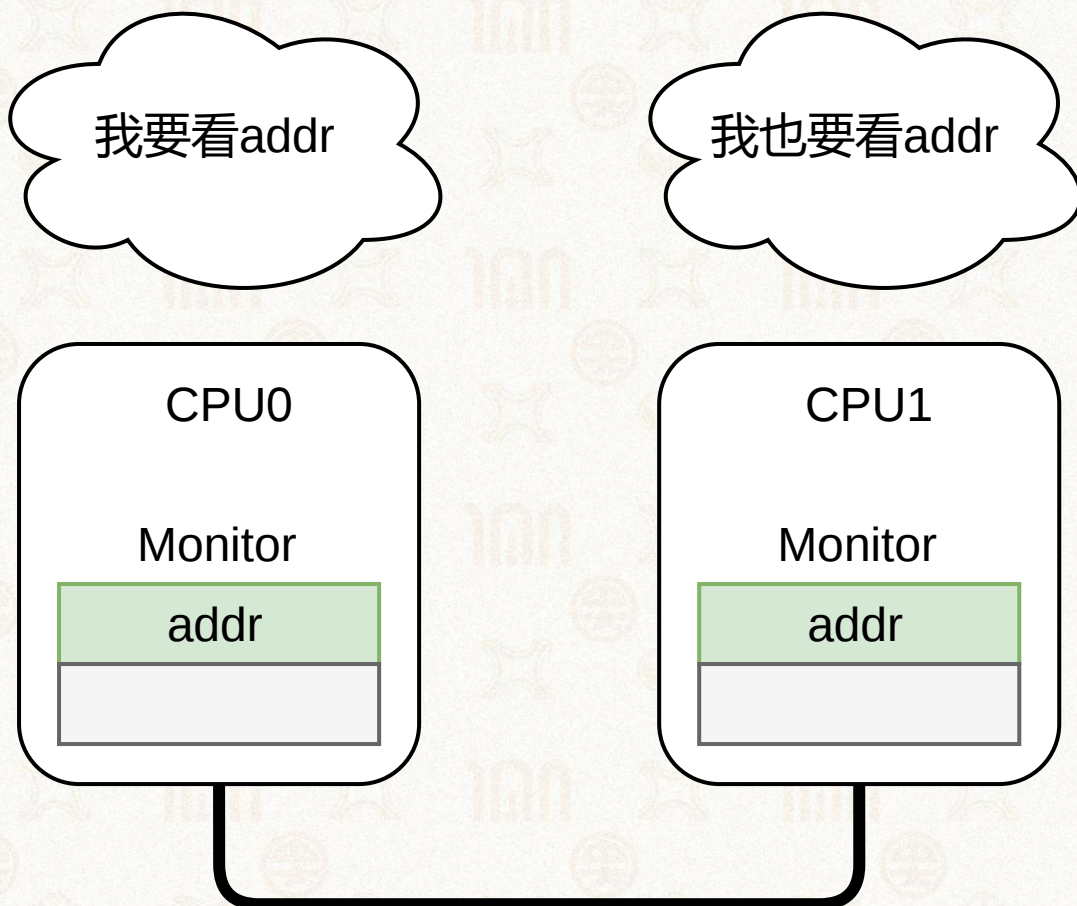
- 对任意地址的修改都要经过总线
- 通过**锁总线**来实现原子操作



硬件原子操作：ARM使用LL/SC实现



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



CPU 0/1

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

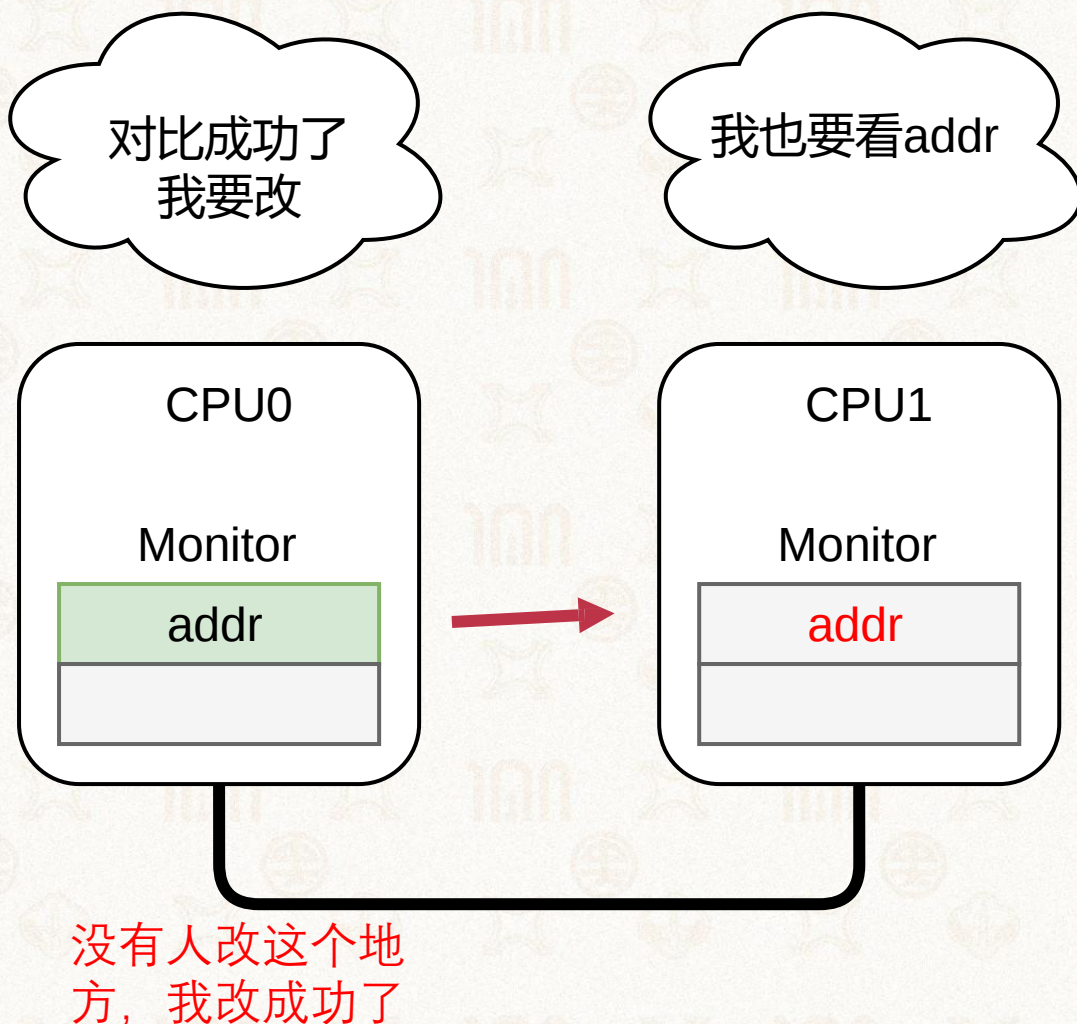
out:

绿色没人修改
红色被修改

- Load-linked & Store-conditional
- 第二行读的时候监视addr
- 第四行修改的时候看addr是否被其他人修改
- 没人修改就写成功，否则回到第二行



硬件原子操作：ARM使用LL/SC实现



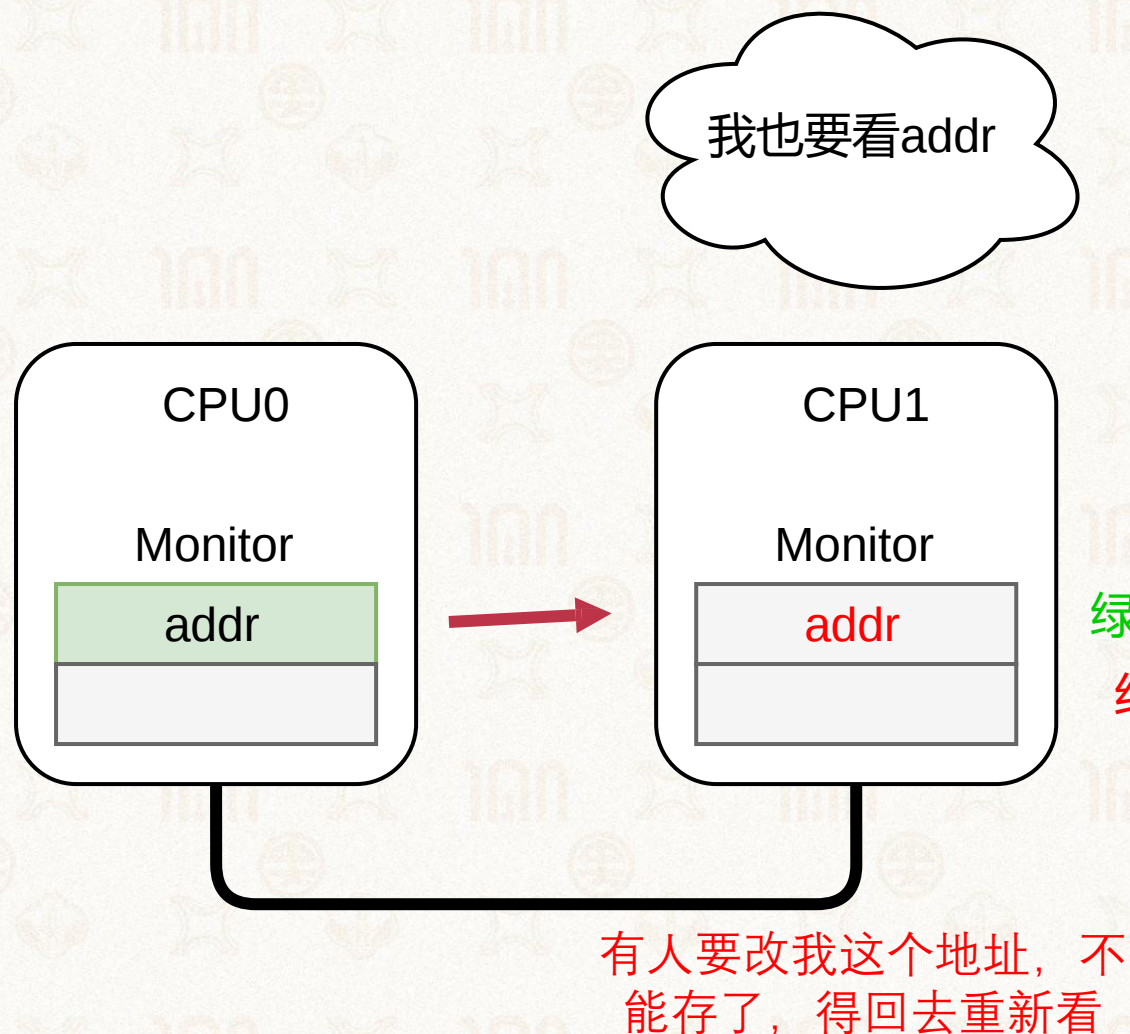
```
retry:  ldxr    x0, addr      LL
        cmp    x0, expected
        bne    out
CPU 0   stxr    x1, new_value, addr  SC
CPU 1   cbnz   x1, retry
out:
```

- Load-linked & Store-conditional
- 第二行读的时候监视addr
- 第四行修改的时候看addr是否被其他人修改
- 没人修改就写成功，否则回到第二行

绿色没人修改
红色被修改



硬件原子操作：ARM使用LL/SC实现



失败重试

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

- Load-linked & Store-conditional
- 第二行读的时候监视addr
- 第四行修改的时候看addr是否被其他人修改
- 没人修改就写成功，否则回到第二行

绿色没人修改
红色被修改



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

有两个线程P1, P2, 它们分别执行下面的代码。其中, total是两个线程都能访问的共享变量, 初值为0; count是每个线程的私有变量。假设这两个线程并发执行, 并可自由交叉, 则这两个线程都执行完后, 变量total可能得到的最小取值为

```
int total = 0;
```

```
P1: {
    int count;
    for(count = 1; count <= 50; count++) {
        total = total + 0;
    }
}
```

```
P2: {
    int count;
    for(count = 1; count <= 50; count++) {
        total = total + 2;
    }
}
```

100

0

2

3

提交

```
typedef struct Node {
    struct Node *next;
    int value;
} Node;

void push(Node **top_ptr, Node *n) {
    n->next = *top_ptr;
    *top_ptr = n;
}
```

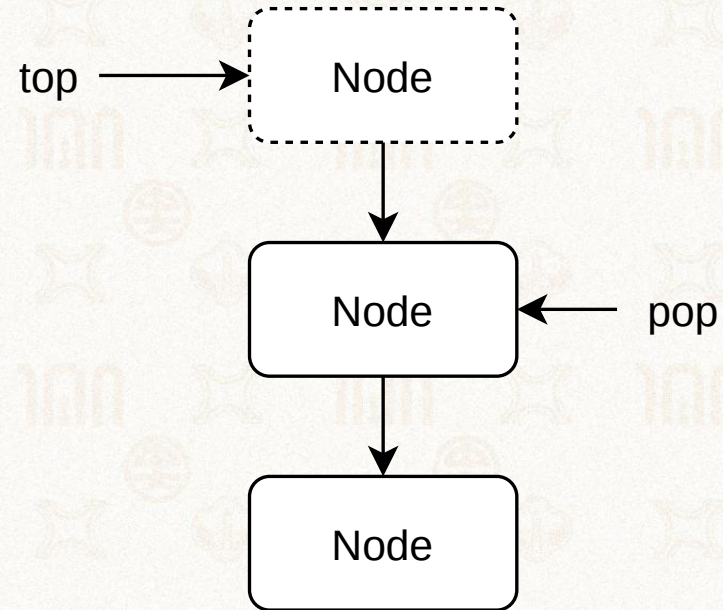
```
Node *pop(Node **top_ptr) {
    if (*top_ptr == NULL) {
        return NULL;
    }
    Node *p = *top_ptr;
    *top_ptr = (*top_ptr)->next;
    return p;
}

Node *top = NULL; // 栈顶初始化为NULL
```

```
Node *new_node = (Node*)malloc(sizeof(Node));
push(&top, new_node);
```

教材P327的思考题：

左边是用链表实现的一个栈，如果多个线程同时创建Node并执行push操作，请问是否有不对的地方？



作答



互斥锁抽象



1924-2024
中山大學 世紀華誕
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

```
volatile int buffer_write_cnt = 0;  
volatile int buffer_read_cnt = 0;  
lock_t buffer_lock;  
int buffer[5];
```

```
void buffer_init(void) {  
    lock_init(&buffer_lock);  
}
```

```
void buffer_add_safe(int msg) {  
    lock(&buffer_lock);  
    buffer[buffer_write_cnt] = msg;  
    buffer_write_cnt = (buffer_write_cnt + 1) % 5;  
    unlock(&buffer_lock);  
}
```

```
int buffer_remove_safe(void) {  
    lock(&buffer_lock);  
    int ret = buffer[buffer_read_cnt];  
    buffer_read_cnt = (buffer_read_cnt + 1) % 5;  
    unlock(&buffer_lock);  
    return ret;  
}
```

➤ buffer_lock: 全局互斥锁

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

}



互斥锁抽象



```
volatile int buffer_write_cnt = 0;  
volatile int buffer_read_cnt = 0;  
lock_t buffer_lock;  
int buffer[5];
```

```
void buffer_init(void) {  
    lock_init(&buffer_lock);  
}
```

```
void buffer_add_safe(int msg) {  
    lock(&buffer_lock);  
    buffer[buffer_write_cnt] = msg;  
    buffer_write_cnt = (buffer_write_cnt + 1) % 5;  
    unlock(&buffer_lock);  
}
```

```
int buffer_remove_safe(void) {  
    lock(&buffer_lock);  
    int ret = buffer[buffer_read_cnt];  
    buffer_read_cnt = (buffer_read_cnt + 1) % 5;  
    unlock(&buffer_lock);  
    return ret;  
}
```

➤ buffer_lock: 全局互斥锁

```
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

```
}
```



互斥锁抽象



1924-2024
中山大學 世紀華誕
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

```
volatile int buffer_write_cnt = 0;  
volatile int buffer_read_cnt = 0;  
lock_t buffer_lock;  
int buffer[5];
```

```
void buffer_init(void) {  
    lock_init(&buffer_lock);  
}
```

```
void buffer_add_safe(int msg) {  
    lock(&buffer_lock);  
    buffer[buffer_write_cnt] = msg;  
    buffer_write_cnt = (buffer_write_cnt + 1) % 5;  
    unlock(&buffer_lock);  
}
```

```
int buffer_remove_safe(void) {  
    lock(&buffer_lock);  
    int ret = buffer[buffer_read_cnt];  
    buffer_read_cnt = (buffer_read_cnt + 1) % 5;  
    unlock(&buffer_lock);  
    return ret;  
}
```

➤ buffer_lock: 全局互斥锁

```
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

```
}
```



互斥锁抽象



1924-2024
中山大學 世紀華誕
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

```
volatile int buffer_write_cnt = 0;  
volatile int buffer_read_cnt = 0;  
lock_t buffer_lock;  
int buffer[5];
```

```
void buffer_init(void) {  
    lock_init(&buffer_lock);  
}
```

```
void buffer_add_safe(int msg) {  
    lock(&buffer_lock);  
    buffer[buffer_write_cnt] = msg;  
    buffer_write_cnt = (buffer_write_cnt + 1) % 5;  
    unlock(&buffer_lock);  
}
```

```
int buffer_remove_safe(void) {  
    lock(&buffer_lock);  
    int ret = buffer[buffer_read_cnt];  
    buffer_read_cnt = (buffer_read_cnt + 1) % 5;  
    unlock(&buffer_lock);  
    return ret;  
}
```

➤ buffer_lock: 全局互斥锁

```
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

```
}
```



互斥锁抽象



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

```
volatile int buffer_write_cnt = 0;  
volatile int buffer_read_cnt = 0;  
lock_t buffer_lock;  
int buffer[5];
```

```
void buffer_init(void) {  
    lock_init(&buffer_lock);  
}
```

```
void buffer_add_safe(int msg) {  
    lock(&buffer_lock);  
    buffer[buffer_write_cnt] = msg;  
    buffer_write_cnt = (buffer_write_cnt + 1) % 5;  
    unlock(&buffer_lock);  
}
```

```
int buffer_remove_safe(void) {  
    lock(&buffer_lock);  
    int ret = buffer[buffer_read_cnt];  
    buffer_read_cnt = (buffer_read_cnt + 1) % 5;  
    unlock(&buffer_lock);  
    return ret;  
}
```

➤ buffer_lock: 全局互斥锁

```
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

```
}
```



互斥锁抽象



1924-2024
中山大學 世紀華誕
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

```
volatile int buffer_write_cnt = 0;  
volatile int buffer_read_cnt = 0;  
lock_t buffer_lock;  
int buffer[5];
```

```
void buffer_init(void) {  
    lock_init(&buffer_lock);  
}
```

```
void buffer_add_safe(int msg) {  
    lock(&buffer_lock);  
    buffer[buffer_write_cnt] = msg;  
    buffer_write_cnt = (buffer_write_cnt + 1) % 5;  
    unlock(&buffer_lock);  
}
```

```
int buffer_remove_safe(void) {  
    lock(&buffer_lock);  
    int ret = buffer[buffer_read_cnt];  
    buffer_read_cnt = (buffer_read_cnt + 1) % 5;  
    unlock(&buffer_lock);  
    return ret;  
}
```

➤ buffer_lock: 全局互斥锁

```
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

```
}
```



用原子操作实现互斥锁：自旋锁(spin lock)



```
void lock_init(int *lock) {  
    // 初始化自旋锁  
    *lock = 0; // 0 表示空闲, 1表示锁  
}
```

```
void lock(int *lock) {  
    while(atomic_CAS(lock, 0, 1) != 0)  
        ; //说明锁值为1, 需要死循环忙等  
}
```

```
void unlock(int *lock) {  
    *lock = 0;  
}
```

```
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

```
}
```



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
 - 运气差的进程可能永远也不能成功，出现饥饿





用原子操作实现互斥锁：排号自旋锁(ticket lock)



```
struct lock {
    volatile int owner;
    volatile int next;
};

void lock_init(struct lock *lock) {
    // 初始化排号锁
    lock->owner = 0;
    lock->next = 0;
}

void lock(struct lock *lock) {
    // 拿取自己的序号
    volatile int my_ticket = atomic_FAA(&lock->next, 1);
    while (lock->owner != my_ticket)
        ; // 循环忙等
}

void unlock(struct lock *lock) {
    // 传递给下一位竞争者
    lock->owner++;
}
```

- 保证竞争者的公平性：
 - 先到先得，和吃饭排队一样的逻辑
- owner: 当前正在吃的食客
- next: 目前放号的最新值



用原子操作实现互斥锁：排号自旋锁(ticket lock)

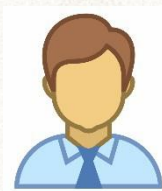


1924-2024
中山大学 世纪华诞
100th ANNIVERSARY
SUN YAT-SEN UNIVERSITY

- owner: 当前正在吃的食客
- next: 目前放号的最新值



owner = 3
next = 6



新顾客

第一步：拿号，拿到6号

```
my_ticket = atomic_FAA(&lock->next, 1);
```

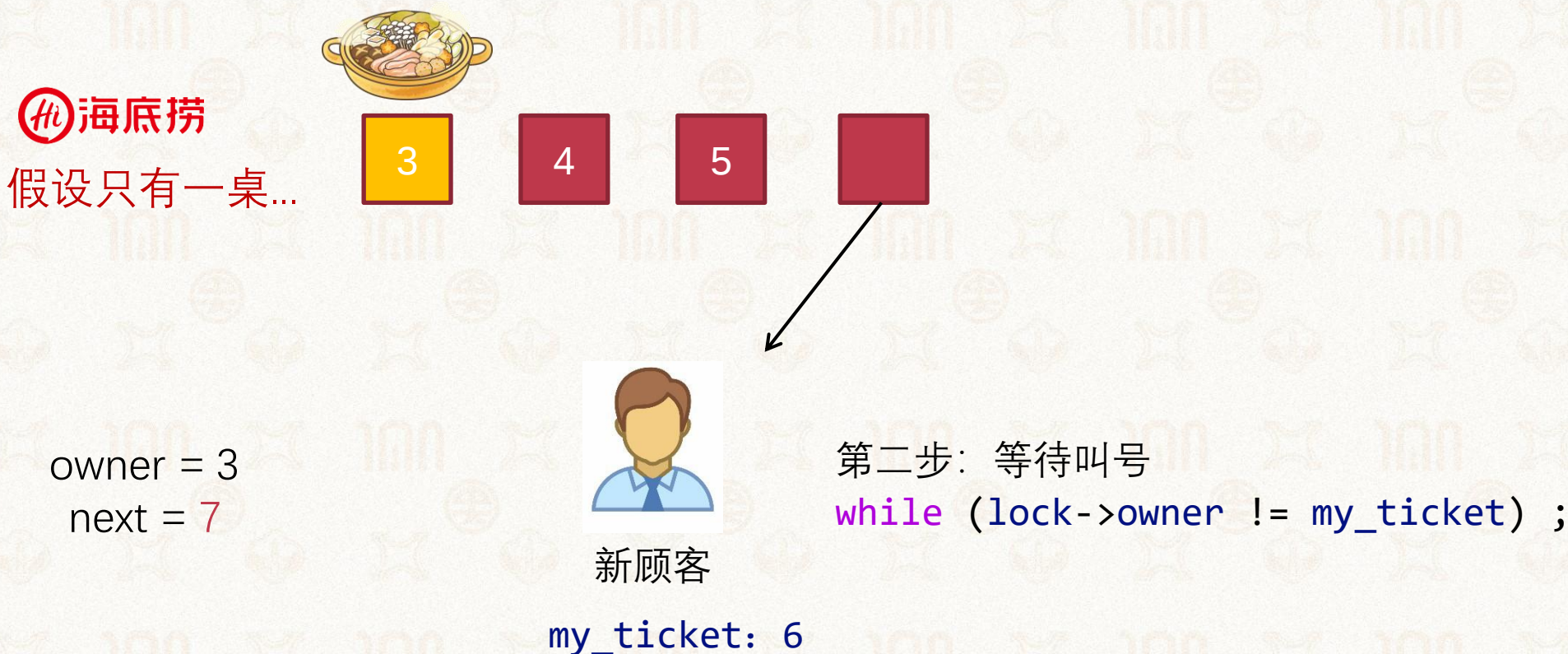


用原子操作实现互斥锁：排号自旋锁(ticket lock)



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

- owner: 当前正在吃的食客
- next: 目前放号的最新值





用原子操作实现互斥锁：排号自旋锁(ticket lock)



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

- owner: 当前正在吃的食客
- next: 目前放号的最新值



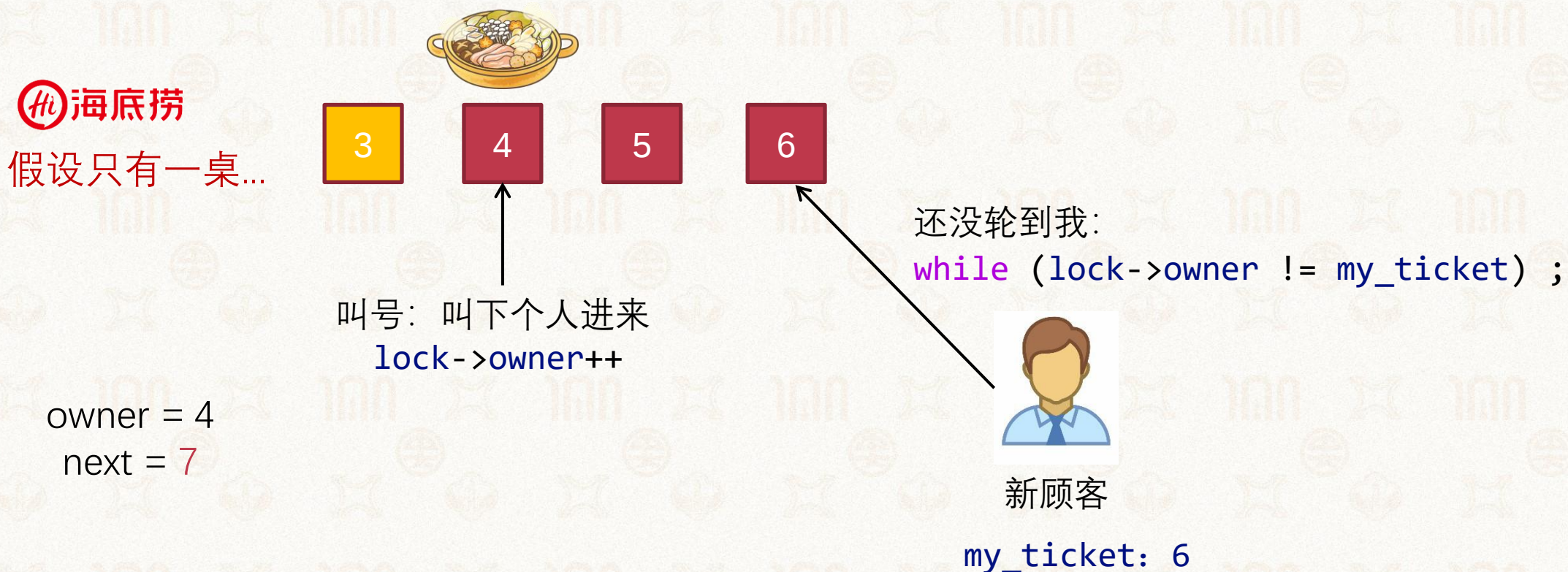


用原子操作实现互斥锁：排号自旋锁(ticket lock)



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

- owner: 当前正在吃的食客
- next: 目前放号的最新值



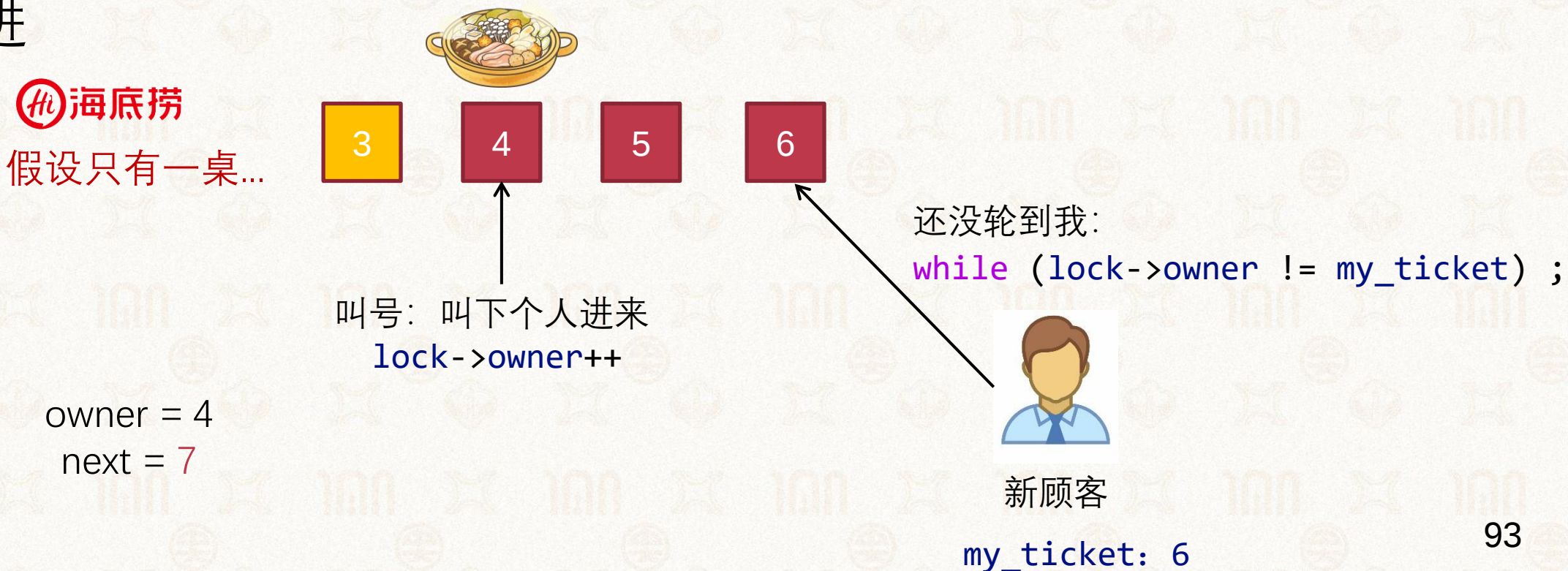


用原子操作实现互斥锁：排号自旋锁(ticket lock)



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

- 是否满足解决临界区问题的三个必要条件？
- 互斥访问
- 有限等待
 - 按照顺序，在前序竞争者保证有限 时间释放时，可以达到有限等待
- 空闲让进





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



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

1924-2024

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>

世 纪 中 大

山 高 水 长