

操作系统

Operating Systems

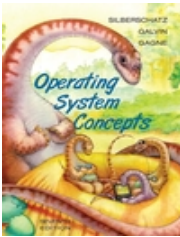
L17 信号量临界区保护

Critical Section

授课教师：李治军

lizhijun_os@hit.edu.cn
综合楼404室

温故而知新：什么是信号量？ 通过对这个量的访问和修改，让大家有序推进。**哪里还有问题吗？**



共同修改信号量引出的问题

初始情况

empty = -1;

这是什么含义?

```
Producer(item) {  
    P(empty);  
    ... }
```

一个可能的执行(调度)

```
P1.register = empty;  
P1.register = P1.register - 1;  
P2.register = empty;  
P2.register = P2.register - 1;  
empty = P1.register;  
empty = P2.register;    -2
```

最终的**empty**等于多少?对吗?

正确的应该是-3

生产者**P₁**

```
register = empty;  
register = register - 1;  
empty = register;
```

生产者**P₂**

```
register = empty;  
register = register - 1;  
Empty = register;
```



竞争条件(Race Condition)

■ 竞争条件: 和调度有关的共享数据语义错误

第i次执行

```
P1.register = empty;  
P1.register = P1.register - 1;  
P2.register = empty;  
P2.register = P2.register - 1;  
empty = P1.register;  
empty = P2.register;
```

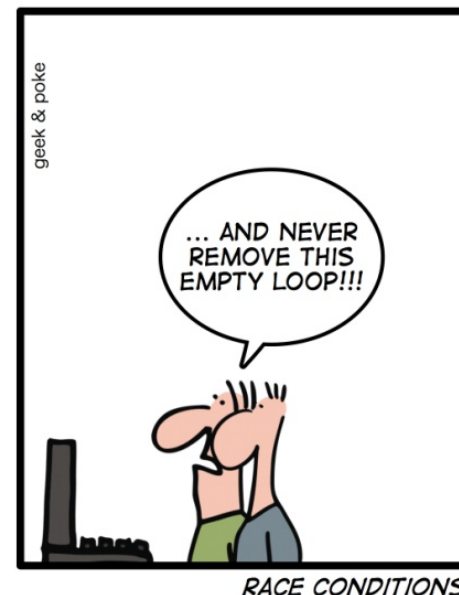
第j次执行

```
P1.register = empty;  
P1.register = P1.register - 1;  
empty = P1.register;  
P2.register = empty;  
P2.register = P2.register - 1;  
empty = P2.register;
```

- 错误由多个进程并发操作共享数据引起
- 错误和调度顺序有关，难于发现和调试

问题：右面的图这两个人的方法
有效果吗？

SIMPLY EXPLAINED



解决竞争条件的直观想法

- 在写共享变量**empty**时阻止其他进程也访问**empty**

仍是那个执行序列

```
P1.register = empty;  
P1.register = P1.register - 1;  
P2.register = empty;  
P2.register = P2.register - 1;  
empty = P1.register;  
empty = P2.register;
```

生产者P₁

检查并给**empty**上锁

```
P1.register = empty;  
P1.register = P1.register - 1;
```

生产者P₂

检查**empty**的锁

?

生产者P₁

```
empty = P1.register;
```

给**empty**开锁

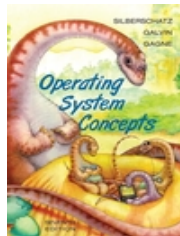
生产者P₂

检查并给**empty**上锁

```
P2.register = empty;  
P2.register = P2.register - 1;  
empty = C.register;
```

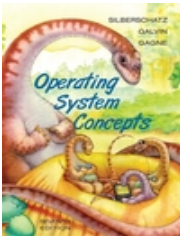
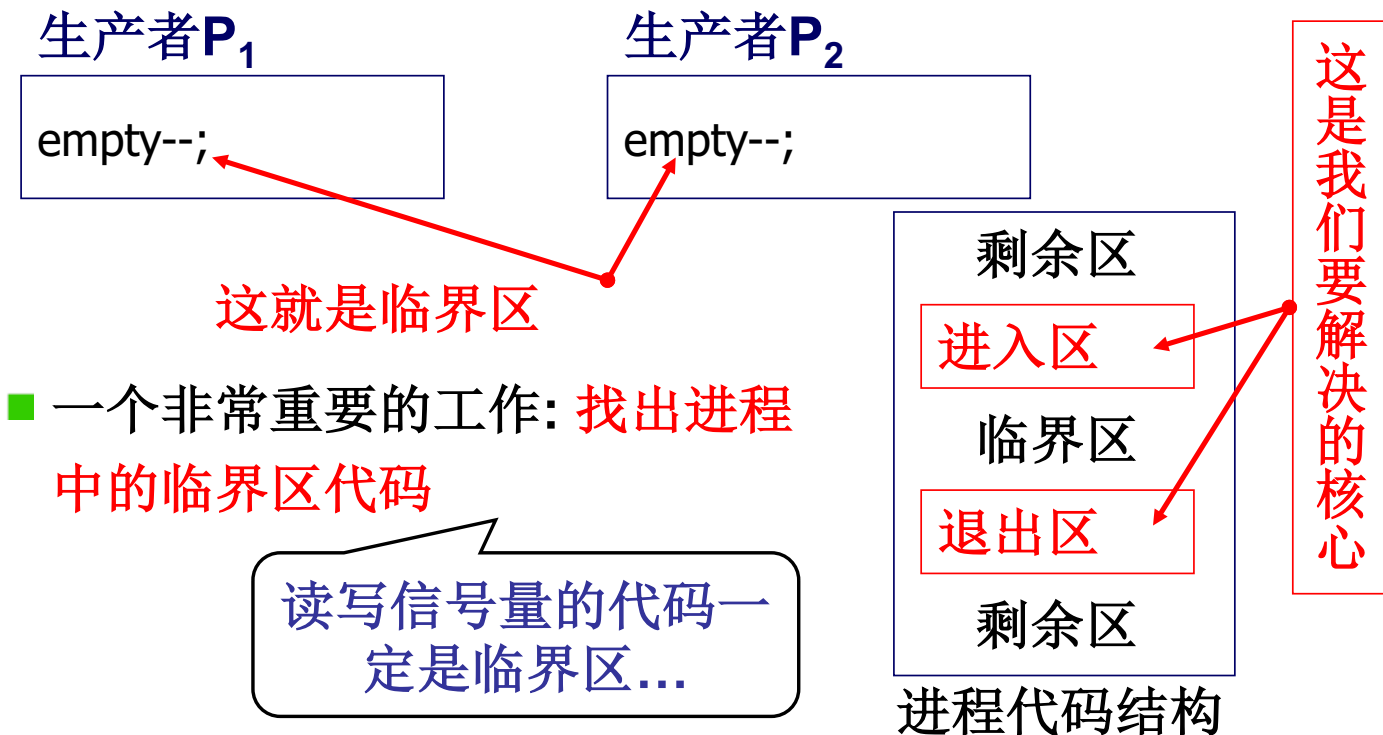
给**empty**开锁

一段代码一次只允许一个进程进入



临界区(Critical Section)

- **临界区**: 一次只允许一个进程进入的该进程的那一段代码



临界区代码的保护原则

- **基本原则：互斥进入：**如果一个进程在临界区中执行，则其他进程不允许进入
 - 这些进程间的约束关系称为**互斥(mutual exclusion)**
 - **这保证了是临界区**
- **好的临界区保护原则**
 - **2. 有空让进：**若干进程要求进入空闲临界区时，应尽快使一进程进入临界区
 - **3. 有限等待：**从进程发出进入请求到允许进入，不能无限等待



进入临界区的一个尝试 – 轮换法

```
while (turn != 0) ;
```

临界区

```
turn = 1;
```

剩余区

进程 P_0

```
while (turn != 1) ;
```

临界区

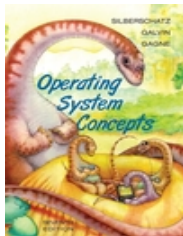
```
turn = 0;
```

剩余区

进程 P_1

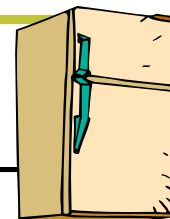
- 问题: P_0 完成后不能接着再次进入, 尽管进程 P_1 不在临界区...(不满足有空让进)

- 满足互斥进入要求



进入临界区的又一个尝试

- 似乎没有任何头绪... 可借鉴生活中的道理



时间	丈夫	妻子
3:00	打开冰箱，没有牛奶了	
3:05	离开家去商店	
3:10	到达商店	打开冰箱，没有牛奶了
3:15	买牛奶	离开家去商店
3:20	回到家里，牛奶放进冰箱	到达商店
3:25		买牛奶
3:30		回到家里，牛奶放进冰箱

- 上面的轮换法类似于什么？值日
- 更好的方法应该是立即去买，留一个便条

许多复杂的道理往往就埋藏在日常生活中！



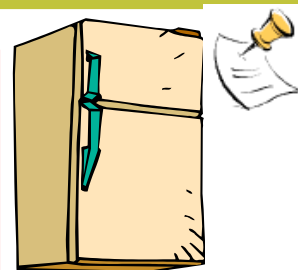
进入临界区的又一个尝试 – 标记法

```
if(noMilk){  
    if(noNote){  
        leave Note;  
        buy milk;  
        remove note;  
    }  
}
```

计算机考虑
问题的方式



```
leave Note;  
if(noMilk){  
    if(noNote){  
        buy milk;  
    }  
}  
remove note;
```



```
flag[0] = true;  
while (flag[1]) ;
```

临界区

```
flag[0] = false;
```

剩余区

进程P₀

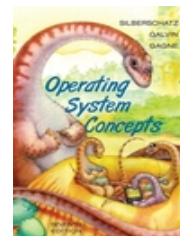
```
flag[1] = true;  
while (flag[0]) ;
```

临界区

```
flag[1] = false;
```

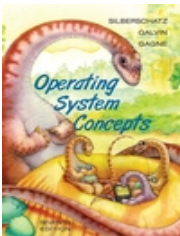
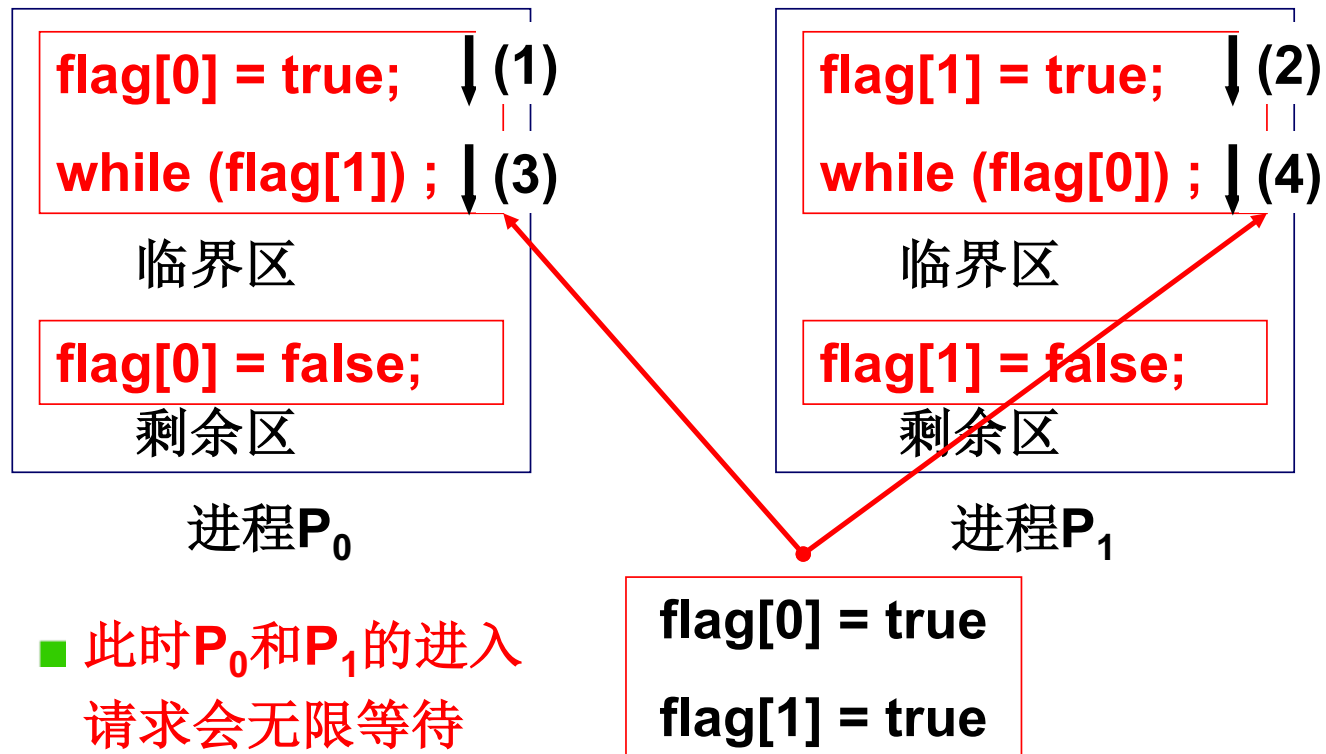
剩余区

进程P₁



标记法能否解决问题？

■ 考虑下面的执行顺序



进入临界区的再一次尝试 – 非对称标记

- 带名字的便条 + 让一个人更加勤劳

丈夫(A)

```
leave note A;  
while (note B)  
{ //X  
  do nothing;  
}  
if (noMilk) {  
  buy milk;  
}  
remove note A;
```

妻子(B)

```
leave note B;  
if (noNote A)  
{ //Y  
  if (noMilk) {  
    buy milk;  
  }  
}  
remove note B;
```

丈夫是那个更勤劳的人

各种情况都正确

- 关键: 选择一个进程进入, 另一个进程循环等待



进入临界区Peterson算法

- 结合了标记和轮转两种思想

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

临界区

```
flag[0] = false;
```

剩余区

进程 P_0

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

临界区

```
flag[1] = false;
```

剩余区

进程 P_1



Peterson算法的正确性

■ 满足互斥进入:

如果两个进程都进入，则
flag[0]=flag[1]=true，
turn==0==1，矛盾！

■ 满足有空让进:

如果进程 P_1 不在临界区，则
flag[1]=false，或者**turn=0**，
都 P_0 能进入！

```
flag[i] = true;
turn = j;
while (flag[j] && turn == j) ;
```

临界区

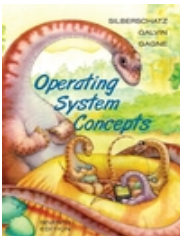
```
flag[i] = false;
```

剩余区

进程 P_i

■ 满足有限等待:

P₀要求进入，flag[0]=true；后面的P₁不可能一直进入，因为P₁执行一次就会让turn=0。



多个进程怎么办？ – 面包店算法

■ 仍然是标记和轮转的结合

- **如何轮转:** 每个进程都获得一个序号，序号最小的进入
- **如何标记:** 进程离开时序号为0，不为0的序号即标记
- **面包店:** 每个进入商店的客户都获得一个号码，号码最小的先得到服务；号码相同时，名字靠前的先服务。

进程
 P_i

```
choosing[i] = true; num[i] = max(num[0], ..., num[n-1])+1;  
choosing[i] = false; for(j=0; j<n; j++) { while(choosing[j]);  
while ((num[j] != 0) && (num[j], j)<(num[i], i)); }
```

临界区

```
num[i] = 0;
```

剩余区



面包店算法的正确性

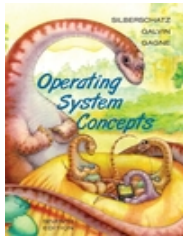
进程
 P_i

```
choosing[i] = true; num[i] = max(num[0], ..., num[n-1])+1;  
choosing[i] = false; for(j=0; j<n; j++) { while(choosing[j]);  
while ((num[j] != 0) && (num[j], j)<(num[i], i)); }
```

临界区

```
num[i] = 0;
```

- **互斥进入:** P_i 在临界区内, P_k 试图进入, 一定有 $(num[i], i) < (num[k], k)$, P_k 循环等待。
- **有空让进:** 如果没有进程在临界区中, 最小序号的进程一定能够进入。
- **有限等待:** 离开临界区的进程再次进入一定排在最后(**FIFO**), 所以任一个想进入进程至多等 n 个进程



一个显然的感觉是：太复杂了，有
没有简单一些的...



临界区保护的另一类解法...

- 再想一下临界区：只允许一个进程进入，进入另一个进程意味着什么？
 - 被调度：另一个进程只有被调度才能执行，才可能进入临界区，如何阻止调度？

cli();

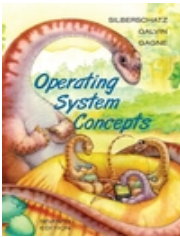
临界区

sti();

剩余区

进程 P_i

- 什么时候不好使？
- 多CPU(多核)...
- 问题：为什么不好使？



临界区保护的硬件原子指令法

```
boolean  
  TestAndSet (boolean &x)  
{  
    boolean rv = x;  
    x = true;  
    return rv;  
}
```

一次
执行
完毕

```
while(TestAndSet(&  
lock)) ;
```

临界区

```
lock = false;
```

剩余区

进程 P_i

■ 想一想: 多CPU情况好不好使?

