实现临界区互斥的基本方法（都无法实现让权等待）

① 软件实现方法

（1）单标志法

- 算法思想 — 两个进程在访问完临界区后会把使用临界区的权限转交给另一个进程。也就是说每个进程进入临界区的权限只能被另一个进程赋予
- 可以实现"同一时刻最多只允许一个进程访问临界区"
- 关键 — turn=i 表谦让，表示允许 i 进入临界区 — i表示i号进程
- **违背"空闲让进"原则，应该也违反【有限等待】**
  - 若某个进程不再进入临界区，则另一个进程也将无法进入临界区
  - 某进程A【让出】临界区，而另一个进程根本没有进入临界区的需求，如果A还想进入临界区就永远进不去
  - **改进:下面几种用flag表示进入临界区的意愿**
- **可能导致饥饿**

```
int turn = 0; //turn 表示当前允许进入临界区的进程号
```

turn 变量背后的逻辑：表达“谦让”

P0 进程:		P1进程:	
while (turn != 0);	①	while (turn != 1);	⑤ //进入区
critical section;	②	critical section;	⑥ //临界区
turn = 1;	③	turn = 0;	⑦ //退出区
remainder section;	④	remainder section;	⑧ //剩余区

①是否轮到自己用？（检查）
②访问临界资源，啦啦啦~
③下次让老渣用（表达谦让）
④做其他事情

只用了turn

（2）双标志先检查法

都进去了

- 算法思想
  - 设置一个布尔型数组 flag[]，数组中各个元素用来标记各进程想进入临界区的意愿 — “flag[0] = ture”意味着 0 号进程 P0 现在想要进入临界区
  - 每个进程在进入临界区之前先检查当前有 没有别的进程想进入临界区，如果没有，则把自身对应的标志 flag[i] 设为 true，之后开始访问临界区
- 关键 — flag [i] =l表达 i 想进入临界区意愿 — 上锁
- **检查操作：while(flag [1])**
- **上锁操作：flag [0] =true**

同下

```
bool flag[2];          //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;       //刚开始设置为两个进程都不想进入临界区
```

理解背后的含义：“表达意愿”

P0 进程:		P1 进程:	
while (flag[1]);	①	while (flag[0]);	⑤ //如果此时 P0 想进入临界区，P1 就一直循环等待
flag[0] = true;	②	flag[1] = true;	⑥ //标记为 P1 进程想要进入临界区
critical section;	③	critical section;	⑦ //访问临界区
flag[0] = false;	④	flag[1] = false;	⑧ //访问完临界区，修改标记为 P1 不想使用临界区
remainder section;		remainder section;	

进入区

①老渣是否想用（检查）
②我小渣要用（表达意愿，上锁）
③访问临界资源，啦啦啦~
④小渣不用了（解锁）

只用了flag

- **违反"忙则等待"原则**
  - 按照 ①⑤②⑥⑧⑦....的顺序执行，P0 和 P1 将会同时访问临界区
  - **（不能实现资源互斥访问）都进去了**
  - 原因：进入区的"检查"和"上锁" 两个处理不是一气呵成的。"检查"后，"上锁"前可能发生【进程切换】

同

（3）双标志后检查法

都没进去

- 算法思想 — 先检查法的改版，先"上锁"后"检查"的方法，来避免上述问题 — 先表达意愿
- 按照 ①⑤②⑥...的顺序执行，P0 和 P1 将都无法进入临界区 — 会因各进程都长期无法访问临界资源而产生【饥饿】
- 虽解决了"忙则等待"
  - **都没进去**
  - 不可能有两个进程同时进入临界区
- **但违背了【空闲让进】和【有限等待】** — 都没进去 — **可能产生饥饿**

```
bool flag[2];          //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;       //刚开始设置为两个进程都不想进入临界区
```

理解背后的含义：“表达意愿”

P0 进程:		P1 进程:	
flag[0] = true;	①	flag[1] = true;	⑤ //标记为 P1 进程想要进入临界区
while (flag[1]);	②	while (flag[0]);	⑥ //如果 P0 也想进入临界区，则 P1 循环等待
critical section;	③	critical section;	⑦ //访问临界区
flag[0] = false;	④	flag[1] = false;	⑧ //访问完临界区，修改标记为 P1 不想使用临界区
remainder section;		remainder section;	

进入区

①我小渣要用（表达意愿，上锁）
②老渣是否想用（检查）
③访问临界资源，啦啦啦~
④小渣不用了（解锁）

只用了flag

（4）Peterson算法

- 算法思想 — 结合双标志法、单标志法的思想，如果双方都争着想进入临界区，那可以让进程尝试"孔 融让梨"（谦让）
- 关键
  - turn=i 表谦让，表示允许 i 进入临界区
  - flag [i] =l表达 i 想进入临界区意愿 — 上锁

  都有
- **先上锁→再谦让→while(本算法的核心)**
- 谁最后说了"客气话"，谁就失去了行动的优先权

```
bool flag[2];          //表示进入临界区意愿的数组，初始值都是false
int turn = 0;          //turn 表示优先让哪个进程进入临界区
P0 进程:
flag[0] = true;                    ①
turn = 1;                          ②
while (flag[1] && turn==1);        ③
critical section;                  ④
flag[0] = false;                   ⑤
remainder section;
P1 进程:
flag[1] = true;                    ⑥ //表示自己想进入临界区
turn = 0;                          ⑦ //可以优先让对方进入临界区
while (flag[0] && turn==0);        ⑧ //对方想进，且最后一次是自己“让梨”，那自己就循环等待
critical section;                  ⑨
flag[1] = false;                   ⑩ //访问完临界区，表示自己已经不想访问临界区了
remainder section;
```

背后的含义：“表达意愿”

表达“谦让”

进入区：1. 主动争取；2. 主动谦让；3. 检查对方是否也想使用，且最后一次是不是自己说了“客气话”

谁最后说了“客气话”，谁就失去了行
Eg: 过年了，某阿姨给你发压岁钱。

场景二
阿姨：乖，收下阿姨的心意~
你：不用了阿姨，您的心意我领了
阿姨：对阿姨来说你还是个孩子，你就
你：真的不用了阿姨，我已经成年了
结局…

```
P0 进程:
flag[0] = true;                    ①
turn = 1;                          ②
while (flag[1] && turn==1);        ③
critical section;                  ④
flag[0] = false;                   ⑤
remainder section;
```

①我小渣要用（表达意愿）
②我愿意优先让老渣用（谦让）
③老渣想用？且最后是自己表示了谦让？我就等待（检查）
④访问临界资源，啦啦啦~
⑤我小渣不想用了

turn flag都用了

- **未遵循让权等待（应该是都没遵循）**
  - 遵循了空闲让进、忙则等待、有限等待 三个原则
  - 都未遵循让权等待，因为进不去的时候，while都在一直转

  硬件软件都不满足让权等待
- **不会饥饿** — 2010

② 硬件实现方法（低级方法、元方法）

1.中断屏蔽方法

- **利用"开/关中断指令"实现**
  - 与原语的实现思想相同
  - 原语的执行具有原子性，即执行过程只能一气呵成，不允许被中断。
  - 可以用"关中断"指令和"开中断"指令这两个特权指令实现原子性（只能内核程序使用）

    ```
    ...
    关中断;
    临界区;
    开中断;
    ```
  - 原语是一种特殊的程序段
- 优点：简单、高效
- 缺点
  - ①不适用于多处理机 — 因为关中断指令只对执行关中断指令的那个处理机有用
  - ②只适用于操作系统内核进程，不适用于用户进程 — 因为开/关中断指令只能运行在内核态

2.硬件指令方法

无法实现让权等待

- **❶ TestAndSet（TSL）指令**
  - TSL 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成 — **只是逻辑描述实际是硬件实现的原子操作**
  - 相比软件实现方法，TSL 指令把"上锁"和"检查"操作用硬件的方式变成了【一气呵成】的原子操作 — 对比双标志先检查法

    ```
    //以下是使用 TSL 指令实现互斥的算法逻辑
    while (TestAndSet (&lock)); //"上锁"并"检查"
    临界区代码段...
    lock = false;                 同//"解锁"
    剩余区代码段...
    ```

    while后面是；；；；号，　　都是δ

    指的只是这个函数

    ```
    //布尔型共享变量 lock 表示当前临界区是否被加锁
    //true 表示已加锁，false 表示未加锁
    bool TestAndSet (bool *lock){
      bool old;
      old = *lock;   //old用来存放lock 原来的值
      *lock = true;  //无论之前是否已加锁，都将lock设为true
      return old;    //返回lock原来的值
    }
    ```

    - 不是交换
    - **先保存旧lock→把lock设为true→再return old**

    lock前都带*，都是*

    感觉*和&一样

- **❷ Swap（XCHG）指令**
  - 共同点
    - 相当于指令一直在执行while操作，lock的值也一直在传入 — **lock的值是由系统传入的**
    - **都用old保存lock的原值(系统传入的)，然后把lock设为ture(上锁)**
    - 如果传入的参数lock=true（已上锁）
      - TSL — 设置新lock=true后返回值old还是true，将继续循环...
      - swap — 交换后old还是true，将继续循环...

      不是用交换实现的
    - **当某一进程使用完临界区把lock设为false时 → 系统传入参数lock→ 交换后old=false → 该进程可以进入临界区**
    - **不会处于阻塞态，只可能处于就绪态或运行态**
      - 就绪态是因为时间片用完
      - 退出临界区的进程负责唤醒就绪态进程
      - 只有信号量才引入阻塞

      2016
    - 逻辑上来看 Swap 和 TSL 并无太大区别 — 都是由硬件实现的原操作，不是原语! — 2023
    - **细节Swap 和 TSL指令指定的只是那个被调用的函数【名字TSL(),和Swap()】** — 2016
    - 硬件方法的优缺点
      - 优点
        - **1.适用于多处理机环境** — 中断屏蔽方法不适合多处理机
        - 2.简单、容易验证其正确性 — 实现简单，无需像软件实现方法那样严格检查是否会有逻辑漏洞
        - 3.可以支持进程内有多个临界区，只需为每个临界区设立一个布尔变量old1、old2
      - 缺点
        - **不满足"让权等待"原则** — 暂时无法进入临界区的进程会占用CPU并循环执行TSL指令，从而导致"忙等"
        - 2.从等待进程中随机选择一个进入临界区，有的进程可能一直选不上，从而导致【饥饿】现象 — **饥饿**

```
//以下是用 Swap 指令实现互斥的算法逻辑
//lock 表示当前临界区是否被加锁
bool old = true;
while (old == true)
  Swap (&lock, &old);
临界区代码段...
lock = false;                 同
剩余区代码段...
```

while后无；；号　　都是δ

```
//Swap 指令的作用是交换两个变量的值
Swap (bool *a, bool *b) {
  bool temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

a，b前都带*，都是*

- 就像平时写交换一样只不过把δ换成*
- **第一步** — 第一个old=true确保第一次while能够进去
- 第一次进入时的old的含义只是为进入循环
- **之后的old的含义是系统传入的lock的值**
- **第二步** — **把lock和old交换值，再判断old是否=true**

23真题
把while改成了if
退出临界区lock错写成了true

- Swap 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成

都是先记录下此时临界区是否已经被上锁（记录在 old 变量上），再将上锁标记 lock 设置为 true，最后检查 old，如果 old 为 false 则说明之前没有别的进程对临界区上锁，则可跳出循环，进入临界区

Presented with xmind