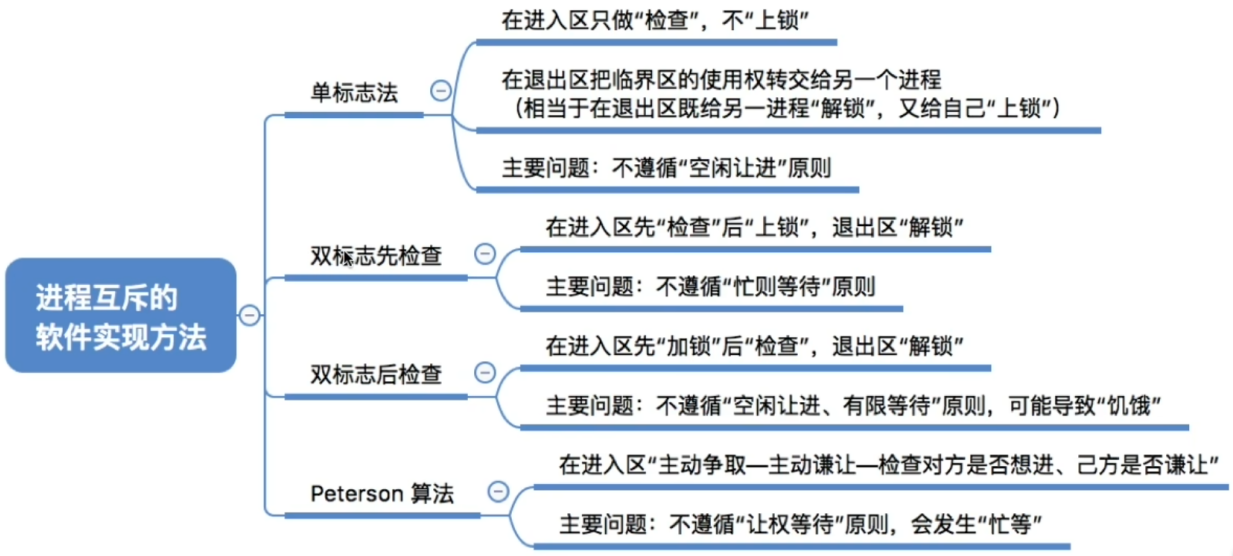
实现临界区互斥的基本方法

进程互斥的软件实现方法



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

```
int turn = 0;//turn表示当前允许进入临界区的进程号
//P0进程
while(turn!=0);//1 进入区
critical section;//2 临界区
turn = 1;//3 退出区
remainder section;//4 剩余区
//P1进程
while(turn!=1);//5
critical section;//6
turn = 0;//7
remainder section;//8
```

turn的初值为0，即刚开始只允许0号进程进入临界区。

若P1先上处理机运行，则会一直卡在5。直到P1的时间片用完，发生调度，切换P0上处理机运行。

因此，该算法可以实现”同一时刻最多只允许一个进程访问临界区“

只能按P0->P1->P0->P1->...这样轮流访问。这种必须”轮流访问“带来的问题是，如果此时允许进入临界区的进程是P0，而P0一直不访问临界区，那么虽然此时临界区空闲，但是并不允许P1访问。

因此，单标志法存在的主要问题是：违背”空闲让进“原则。

双标志先检查

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，比如"flag[0]=true"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。

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

若按照152637...的顺序执行，P0和P1将会同时访问临界区。

因此，双标志先检查法的主要问题是：违反"忙则等待"原则。

原因在于，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。

双标志后检查

算法思想：双标志先检查法的改版。前一个算法的问题是先"检查"后"上锁"，但是这两个操作又无法一气呵成，因此导致了两个进程同时进入临界区的问题。因此，人们又想到先"上锁"后"检查"的方法，来避免上述问题。

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

若按照1526...的顺序执行，P0和P1将都无法进入临界区

因此，双标志后检查法虽然解决了"忙则等待"的问题，但是又违背了"空闲让进"和"有限等待"原则，会因各进程都长期无法访问临界资源而产生"饥饿"现象。

两个进程都争着想进入临界区，但是谁也不让谁，最后谁都无法进入临界区。

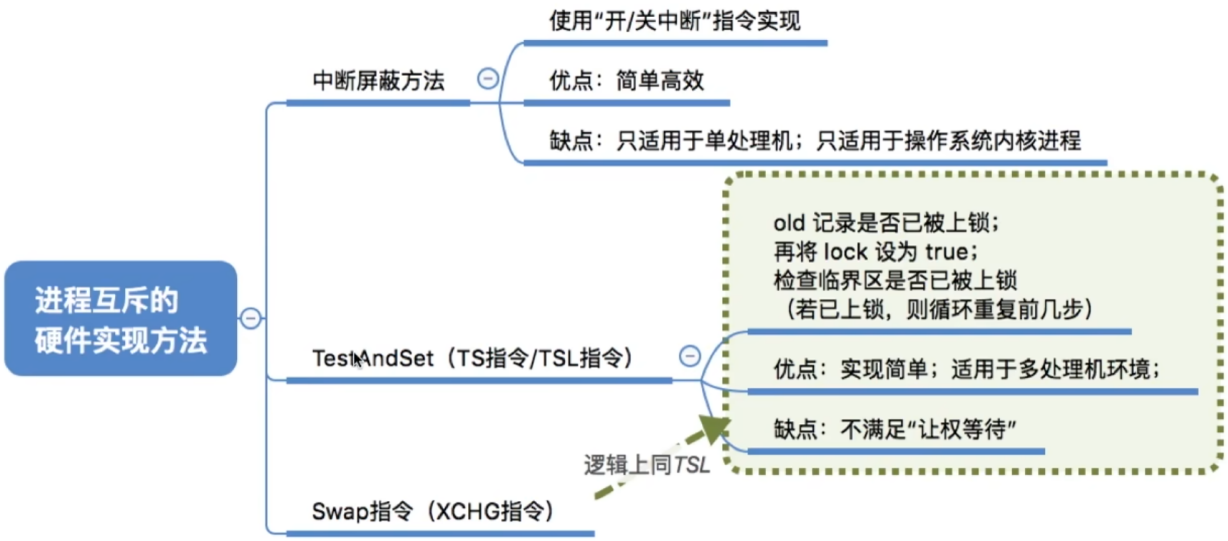
Peterson算法

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）。做一个有礼貌的进程。

```
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;
```

Peterson算法用软件方法解决了进程互斥问题，遵循了空闲让进、忙则等待、有限等待三个原则，但是依然未遵循让权等待的原则。

进程互斥的硬件实现方法



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，即在某进程开始访问临界区到结束访问为止都不允许被中断，也就不能发生进程切换，因此也不可能发生两个同时访问临界区的情况）

- ...
- 关中断；

  关中断后即不允许当前进程被中断，也必然不会发生进程切换
- 临界区；
- 开中断；

  直到当前进程访问完临界区，再执行开中断指令，才有可能有别的进程上处理机并访问临界区
- ...

优点：简单、高效

缺点：不适用于多处理机；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，这组指令如果能让用户随意使用会很危险）

TestAndSet（TS指令/TSL指令）

简称TS指令，也有地方称为TestAndSetLock指令，或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。

```
//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
}
//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;//"解锁"
//剩余区代码段...
```

若刚开始lock是false，则TSL返回的old值为false，while循环条件不满足，直接跳过循环，进入临界区。若刚开始lock是true，则执行TLS后old返回的值为true，while循环条件满足，会一直循环，直到当前访问临界区的进程在退出区进行“解锁”。

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。

优点：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；适用于多处理机环境

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”。

Swap指令（XCHG指令）

有的地方也叫Exchange指令，或简称XCHG指令。

Swap指令使用硬件实现的，执行的过程不允许被中断，只能一气呵成。

```
//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;
}
//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;
//剩余区代码段...
```

逻辑上来看Swap和TSL并无太大区别，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。

优点：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；适用于多处理机环境

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”。