**1.** 忙等待：程序仍然占有cpu运行，但是在等待锁或者互斥量或者其他资源，不停地检测和判断而没有实际操作

其他类型等待：阻塞等待，释放cpu资源的等待直到等待时间发生才会继续执行；准备等待：在准备队列中的等待，排队等待CPU资源; 休眠等待

避免忙等：可以，在暂时获得不到需要的资源时，将进程放入该资源的等待队列，并且将其挂起转为阻塞等待，释放CPU资源。直到资源可用时将其唤醒

**2.** 设同时调用`withdraw`取出a元，调用`desposit()`存入b元（分为两个进程执行），银行账户中有`count`元。且取款和存款不是原子操作。

```
//取款a元 p1
register1=count       //(1)
register1=register1-a //(2)
count=register1       //(3)
//存款b元 p2
register2=count       //(4)
register2=register2+b //(5)
count=register2       //(6)
```

竞争：p1和p2进程执行过程会被打断，p1和p2竞争`count`，且p1和p2的执行顺序的随机的，导致最终`count`可能为`count-a`或者`count+b`的错误结果

```
T0: 执行p1(1):        register1=count
T1: 执行p1(2):        register1=count-a
T2: 切换进程执行p2(4):register2=count
T3: 执行p2(5):        register2=count+b

若T4切换进程：
T4：切换P1(3):       count=register1=count-a
T5: 切换P2(6):        count=register2=count+b
最终：count=count+b

若T4没有切换进程：
T4: 执行P2(6)      count=register2=count+b
T5: 执行P1(3)      count=register1=count-a
最终：count=count-a
```

竞争状态的防止：

使用同步机制将p1的(1)(2)(3)过程和p2的(4)(5)(6)过程用互斥锁或者信号量保护，使p1和p2执行过程并不会被"打断",不会有其他进程获得`count`。以下是用互斥锁保护的, 进程执行前要获得锁，如果锁被其他进程持有，进程要空转或者挂起等待锁的释放，无法继续执行。p1在执行过程中，`mutex`被p1持有，即使中途切换进程p2,p2会在`mutex.acquire()`处等待，然后切换回p1,直到p1执行完释放`mutex`，p2才能获得`mutex`继续执行

```
//取款a元 p1
mutex.acquire();
register1=count       //(1)
register1=register1-a //(2)
count=register1       //(3)
mutex.release();
//存款b元 p2
mutex.acquire();
register2=count       //(4)
register2=register2+b //(5)
count=register2       //(6)
mutex.release();
```

**3.**

a. 竞争条件：++和--操作并不是原子执行的，而是分成三步的，需要将变量的值从内存加载到寄存器中，然后在寄存器上实现加减运算，再将其存回到内存变量地址中。

`allocate_process`和`release_process`共享`number_of_process`变量，且两个函数对其的写操作都不是原子性的。当这两个函数并发运行，同时访问和写`number_of_process`时，由于执行顺序是不确定的，可能会得到错误的值。

错误情况举例：两个线程p1和p2分别同时执行`allocate_process`和`release_process`时，T0-T3的过程中，p1和p2都执行完(1)(2)。若T4时刻执行P1的(3),T5执行P2的(3),那么`number_of_process`最终结果为`number_of_process-1`。

```
++number_of_process:p1
register1=number_of_process  //(1)
register1=register1+1        //(2)
number_of_process =register1 //(3)

--number_of_process:p2
register2=number_of_process  //(1)
register2=register2-1        //(2)
number_of_process =register2 //(3)
```

b. ==++ --都不是原子操作，但是都不能被打断

```
int allocate process(){
    int new pid;
    mutex.acquire()//锁的获取
    if (number of processes == MAX PROCESSES)
    return -1;
    else{
     ++number_of_processes;
    }
    mutex.release()//锁的释放
}
void release process() {
    mutex.acquire()//锁的获取
    --number_of_process;
    mutex.release()//锁的释放
}
```

```
16
```

c.不能，还要使用原子整数类中特殊的自增自减函数和其他函数来保证对`number_of_acquire`的读写是原子性的

**4.**

违反了互斥原则

- 假如有进程`i,j(i<j)`取得的`number[i]=number[j]=3`是一样的且为`number[]`中最大的
- 现在有进程`k(i<j<k)`正在执行`number[k]=1+getmax(number[],n);`取号，在比较中已经得到`number[i]`是最大的，但是还没赋值的时候切换进程`i`。
- 设进程`i`前面已经没有人排队等待，进程`i`跳出循环来到临界区，执行`number[i]=0`
- 此时切换回进程`k`,`number[k]=number[i]+1=1`,
- `number[k]`应该等于4，但是由于`number[i]=1+getmax(number[],n);`不满足互斥性而导致错误赋值。在面包师算法中意味着进程`k`插了进程`j`的队

**5.**

不正确：

- 信号量的初始化：

  `car_avail`是空闲的车辆数，应该初始化为`m`；

  `car_taken`是游客上车,应该初始化为0（二元信号量）；

  `car_filled`是游客要乘坐的车是否唤醒,应该初始化为`0`（二元信号量）；`passenger_released`等待的游客数，应该初始化为`n`

- 进程中对信号量的释放和获取修改如下

```
process passenger(i:1 to num_passenfers)
p(car_avail) //查询是否有空车
v(car_filled) //唤醒
v(car_taken)//上车
do true->nap()//
p(passenger_released//开车游玩
v(car_avail)//游玩结束空车增加

process car(j:=1 to num_cars)
p(car_filled)//唤醒车
p(car_taken)//接上游客
do true->nap()//开车游玩
v(passenger_released())//开车门下车
```

**6.**

缺陷：只要有一个读者还在读，任意读者都可以读共享资源，所以可能会导致写者饥饿问题

**7.**

a.

T2: 2 4 1 3+2 2 2 4=4 6 3 7>2 5 3 3　　可用 4 6 3 7

T3　4 1 1 0+4 6 3 7=8 7 4 7>6 3 3 2　　可用 8 7 4 7

T1　2 1 0 2+8 7 4 7=10 8 4 9>4 2 3 2　　可用10 8 4 9

T0　3 1 4 1+10 8 4 9=13 9 8 10>6 4 7 3　可用13 9 8 10

T4　2 2 2 1+13 9 8 10=15 11 10 11>5 6 7 5

安全序列为：T2，T3, T1，T0，T4

b.

T4 (2,2,2,4)的请求不可以立即批准

　　如果申请了，可用 0 0 0 0

　　T4 2 2 2 1+2 2 2 4=4 4 4 5<5 6 7 5 没有安全序列

c.

T2 申请(0,1 1,0) 可以 安全序列同a 为T2,T3,T1,T0.T4

d.

T3申请（2 2 1 2）可以,可用（0，0，1，2）

T3: 4 1 1 0+ 2 2 1 2 +0，0 1 2=4 1 1 0+2 2 2 4=6 3 3 4>6 3 3 2

T2

T1

T0

T4

安全序列为：T3,T2,T1,T0,T4

**8.**

```
sem_deer=1//这个是鹿（二元信号量）
sem_santa=0//唤醒圣诞老人数（计数信号量）
sem_jinglin=1//精灵信号量（二元信号量）
共享资源 count_deer=0
共享资源 problem_count=0

//九个鹿的进程
for(i:=9)
sem_deer.p()//
count_deer++;//回去的鹿数量++
if (count_deer==9)sem_santa.v()//唤醒圣诞老人
sem_deer.v()

//精灵的进程
sem_jinlin.p()
```

```
problem_count++;
if (problem_count==3){
sem_santa.V()
}
else{
sem_jinlin.v()//没满三个精灵问题，还可以继续有精灵产生问题
}

//圣诞老人进程
sem_santa.P()
if(count_deer==9 && problem_count==3){
    //准备雪橇
}
else if(promble_count==3 && count_deer!=9){
     //解决精灵问题
     problem_count=0
     sem_jinlin.V()
  }
else if(count_deer==9){
    wake for deer
}
```