Semaphore 信号量

1. 并发

并发重点考虑的两方面：
（1）互斥（mutual exclusion）：某一资源同时只允许一个访问者对其进行访问，具有排它性。比如临界区的资源。可以用锁或信号量实现。一般取名为mutex（锁和信号量都可以）。
（2）同步（ordering）：在互斥的基础上，访问者对资源有序访问。也就是任务之间有依赖关系，某个任务的运行依赖于另一个任务。可用条件变量或信号量实现。同步的目的是让异步的任务相互配合，按希望的顺序进行

互斥是一种特殊的同步。互斥可以理解为该资源只有一个，只能被分配给一个进程使用，只有这个进程释放了，才能被其他进程使用。

2. 信号量

一个整数变量，或者整数加等待队列的结构，表示某种资源的数量。
通过两个原语（不可中断，例如需要用TestAndSet实现）对信号量s进行操作。

- wait(s)， 或者P(s)、sem_wait(s)： 信号量减1（减前为0时，根据是否允许负数，实现略有不同）
- signal(s)，或者V(s)、sem_post(s)：信号量加1，唤醒一个等待线程

提醒：信号量的值允许负数吗？不同信号量实现是不同的。有些实现限制信号量只能是0和正整数(例如POSIX信号量，sem_wai(),sem_post()是POSIX标准API，因此不允许负数，s大于0时，sem_wait()会减去1，当s为0时，sem_wait()直接让调用线程等待，而不去减s）；有些信号量机制允许负数。 如果信号量值为-N，表示有N个线程在等待该信号量。

根据取值范围，信号量分为

- Binary Semaphore 二进制信号量 （只能取值0，1，一般用于mutex），wait本质是控制临界区的入口，signal是访问临界区结束，释放资源。
- Counting Semaphore 计数信号量（可以取整数） （一般用于表示实际的资源数量）

信号量功能非常强大，可以实现互斥/同步，但也有缺点
（1）wait和signal分散在代码的各个地方，大规模使用会导致代码的模块化不好；编程逻辑复杂，而且容易出错。
（2）优先级反转问题。（高优先级任务A通过信号量访问共享资源，但信号量被低优先级

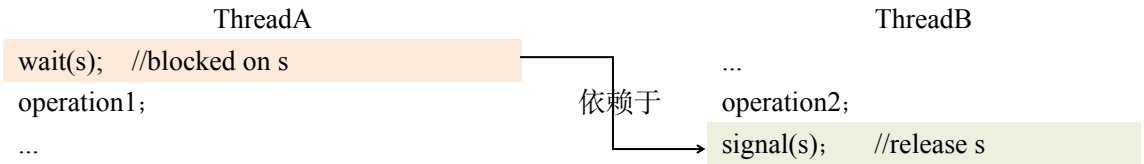
任务C占有，导致A被阻塞。如果只有A、C两种任务，问题不严重，因为临界区通常很短，C访问完很快就让出给A，但是如果A、C之间还存在长的中优先级任务B，且B不访问共享资源，那么B就会中断C的运行，导致C长时间不能运行，不能释放资源，从而A也不能获得被C占用的信号量运行。A、B优先级反转，A实时性无法保证）

3. 信号量实现互斥/同步

信号量实现互斥的方法，按照临界区前后成对出现原则。
竞争者的代码是对称的。
wait(mutex);
访问临界区；
signal(mutex);

信号量实现同步。A线程中的operation1操作需要在B线程的operation2操作后执行。按“一前一后”原则进行。

- 在优先级较高的operation2操作的后面执行 signal 操作，释放资源
- 在优先级较低的operation1操作的前面执行 wait操作，申请占用资源。



实际执行 threadA会在执行到wait时等待threadB执行完signal，因此operation2先于operation1执行。

信号量用于互斥/同步的三个典型例子。一定要完全理解代码。

（1）Producer-consumer
（2）Reader-writer
（3）Dining philosopher

4. 信号量分析问题“四步法”

分析同步互斥问题的常见思路： （信号量问题“四步法”）
（1）找线程：分析有几类不同的线程/进程，并写出每类线程基本操作。例如producer-consumer中有两类线程producer()，consumer()。先不考虑同步/互斥，把线程的基本操作写出来。在此基础上还要进一步分析出每类线程有多少实例。
例如：生产者的基本操作就两步。

```
Producer()
{
    While (1)
     {produce an item;
      put it to buffer;
      }
}
```
producer可以多个实例。

如果多实例：进一步为后续同步互斥关系考虑

a. 同类线程之间的互斥/同步问题。

b. 多实例线程之间是否有操作的差异性。（例如读者问题，第一个读者、中间读者、最后一个读者，在操作上有差异性）

（2）找互斥关系：互斥在同类和不同类线程间都有。分析有哪些资源，哪些需要互斥访问。

这里的互斥资源主要包括两类：(a) 所有线程需要互斥访问的实际资源，例如缓冲区资源；(b)一些同类线程之间的状态变量，例如读者数量，readcount。

每个互斥资源都要定义一个互斥信号量mutex，常见逻辑是wait,signal在同一个线程内配套使用。

wait(mutex);

修改互斥变量/访问临界区；

signal(mutex);

（3）找同步关系：同步一般是不同类线程间（同一个线程内操作的先后顺序就是按在自己代码中的先后顺序，不需要额外同步）。分析不同类线程间的操作有哪些依赖关系。定义同步信号量。一种依赖关系就需要一个信号量。常见同步逻辑是依赖方（需要后执行的操作）使用wait，被依赖方（需要先执行的操作）使用signal。例如，producer中的放置操作需要等待consumer的取出操作腾空位置，那么就需要一个信号量empty，producer在放入前执行wait(empty)， consumer在取出后执行signal(empty)。 同时consumer也要等待producer放入物品，那么就需要另一个信号量full，consumer取出前执行wait(empty)， producer放入后执行signal(empty)。

（4）补充代码：定义信号量与初值，在基本操作上补充同步互斥代码。

信号量赋初值（mutex一般为1，同步信号量根据资源情况，一般是0或者最大值）。

在线程基本操作基础上加入对信号量的wait/signal代码。

注意最后写完了多个进程代码后，要写一个并发运行。例如

```
main()
{
cobegin{
producer();
consumer();
} coend;
}
```

在这四部的基础上，进一步考虑：公平、死锁问题。
（1）公平：不同类线程之间是否会饥饿（也就是是否某一类线程独占资源）
（2）死锁：有没有存在环状等待的情况（特别申请多种资源时，也就是某线程内连续多个wait时，要考虑死锁问题）

关于同步和互斥的区别：
同步->不同任务间，有中间数据生成。典型场景：A任务的运行，依赖于B任务产生的数据。
互斥->同类，不同类线程之间都可能按序访问互斥资源。互斥的资源一般是已有的资源（不是某个任务生成的中间数据）或者线程间共享的状态。

5. 典型例子

生产-消费者类

1. 桌上有一空盘，允许存放N个水果。妈妈可向盘中放苹果，也可向盘中放桔子，儿子专等吃盘中的桔子，女儿专等吃盘中的苹果。请用信号量实现妈妈、儿子、女儿三个并发线程的同步。

分析：
（1）首先是判定几类进程/线程。显然，有3个不同的任务，妈妈，儿子，女儿。基本操作是 {take a fruit; put it to plate;}，{take a fruit from plate; eat it;}。理论上可以多实例，例如多个儿子。一般是单实例。
（2）分析互斥的资源。三类线程都要使用盘子，需要互斥使用。盘子互斥信号量mutex。
（3）分析同步依赖的关系。女儿（取苹果）->妈妈（放苹果），儿子（取桔子）->妈妈（放桔子）。 妈妈（放水果）->儿子/女儿（取出空位） （A->B表示A依赖于B，也就是），所以需要3个同步信号量。以“女儿（取苹果）->妈妈（放苹果）”为例，信号量apple初始为0，女儿pick apple 前执行wait(apple)，妈妈put apple后执行signal(apple)。
（4）信号量初值。mutex肯定是1，空位是N。水果是0。

代码
```
semaphore plate=N, apple=0, orange=0,    //同步用
sempahore mutex=1;   //互斥用

mother() {   //
    while (1) {
        prepare a fruit;
        wait(plate) ;   //是否有空位
        wait(mutex);   //向盘中放水果，互斥使用盘子。
        put the fruit on the plate;
```

```
            signal(mutex);
            if (fruit=apple)
                  signal(apple);
            else
                signal(orange);
            }
}

son() {
      while (1)
      {
         wait(orange) ;
         wait(mutex);
         pick an orange from the plate;
         signal(mutex);
         signal(plate);
         eat the orange;
}

daughter() {
      while (1)
      {
         wait(apple) ;
         wait(mutex);
         pick an apple from the plate;
         signal(mutex);
         signal(plate);
         eat the apple;
}
```

注意：上述代码是正确的代码，在mother，son，daughter都有多个实例的情况下可以正常工作（如多个儿子，多个女儿）。如果只有三个线程（也就是三类线程都只有一个实例），那么mutex的操作可以省略（所有wait(mutex)，signal(mutex)去掉）。因为三个同步信号量关系可以保证互斥（同类之间无竞争，只有一个实例；不同类线程之间靠同步信号量顺序执行，即使son和daughter同时从盘子取，他们取的也是不同位置资源，不冲突)。

水果问题是生产-消费者问题的变形版本，实际包含多类生产和消费者。

类似问题练习：

（1）玩具自行车组装问题：玩具自行车有2个轮子和1个车身构成，生产轮子的工人每做好一个轮子，就放到轮子的工作台，工作台可以放N1个轮子，满的时候暂停生产；生产车身的工人把车身做好，放到放车身的工作台，可以放N2个车身，满的时候暂停生产；组装工人从两个工作台分别取2个轮子和1个车身，然后组装成一台玩具自行车。试着用信号量来描述这个过程。

读者-写者类

I. 在A、B两地中间有一段路，路宽只能容纳一辆车通行，假设汽车不能倒车，试用信号量解决两个方向车通行中的同步互斥问题。

分析：这是一个改进版的读者-写者问题。

（1）首先分析几类线程。两类车：A到B，和B到A。每类里面都有多个实例。不考虑同步互斥，车的基本操作：{drive through road AB or BA;}

（2）分析互斥共享资源：AB之间的路，对于同向都是读者锁，对于反向都是写者锁。 也就是同一方向的车可连续通过，当某一方有车时，另一方向的车必须等待。因此需要一个AB间道路的资源互斥锁mutex。

与读写问题类似， 因为同向多辆车之间的操作有所不同（第一辆，中间，最后一辆），需要记录同向车数量的状态变量（countAB，countBA），也就需要两个状态变量修改的互斥锁SemCountAB，SemCountBA。

（3）分析依赖关系。没有依赖关系（没有中间数据生成）。

AB向的车->BA向的车（释放路权）。BA向的车->AB向的车（释放路权）。因为AB间的路只有1份，所以直接就是互斥关系可以表达这种关系。

代码：

```
Semaphore mutex=1, SemCountAB=1, SemCountBA=1;  //全部是互斥。
int countAB=0, countBA=0; //两个方向车的数量

LeftCar()
{
wait(SemCountAB);    // 修改AB方向车的数量需要互斥
countAB++;
if (countAB=1)
      wait(mutex);  // 本方向第一辆车需要获得AB间路的使用权（与另一方向车竞争）
signal(SemCountAB);
Drive through the road AB;
Wait(SemCountAB);
```

```
countAB--;
if (countAB=0)
   signal(mutex);   // 本方向最后一辆车需要释放AB间路的使用权
signal(SemCountAB);
}
```

RightCar()的代码完全对称。就不写了。

这个实现没有考虑公平性，也就是某一方向一直有车，另外一个方向就会一直等下去。

公平版本

对两个方向的车都在外层加一个更大的锁：两个方向的车，无论哪个方向，一方提出通行请求后，另一方的未上路的车必须等待，已上路的车可以继续通行，通行结束后切换到提出请求的方向。

（实际就是进行两个方向全局排序，两个方向都申请全局token，按申请顺序先后通过）

增加一个上路请求的信号量 token=1，wait(token)表示发起使用道路资源的请求。本质上就是维护一个全局队列，按wait的先后顺序通过。

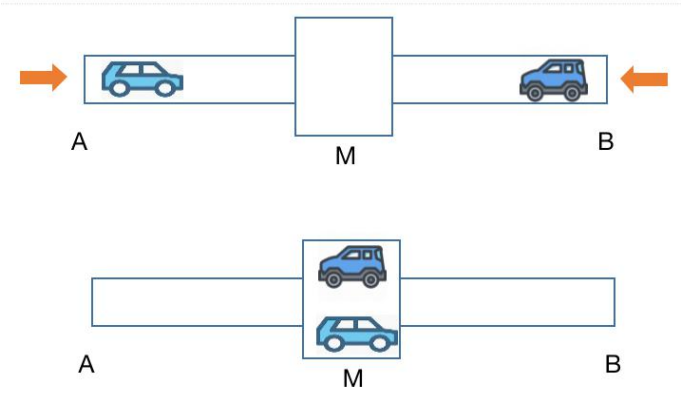
```
Semaphore mutex=1, SemCountAB=1, SemCountBA=1; //mutex是道路资源使用权
Semaphore token=1;   //token是两个方向公平竞争的令牌。
int countAB=0, countBA=0; //两个方向车的数量

LeftCar()
{
wait(token);   // 请求上路，第一个获得token的车会先尝试进入道路，优先上路；
wait(SemCountAB);    // 修改AB向车的数量需要互斥
countAB++;
if (countAB=1)
      wait(mutex);   // 本方向第一辆车需要获得AB间路的使用权（与另一方向车竞争）
signal(SemCountAB);
signal(token);
Drive through road AB;
Wait(SemCountAB);
countAB--;
if (countAB=0)
   signal(mutex);   // 本方向最后一辆车需要释放AB间路的使用权
signal(SemCountAB);
}
```

//释放token，这样其他车还能获得token，并尝试上路。如果后续是同方向的车获得token，可以直接上路（countAB大于1不会阻塞在mutex）。如果释放后被Right方向的第一辆车获得token，那么那辆车会被阻塞在mutex（countBA=1），因为Left方向还有已上路的车，没有释放mutex。

右边车代码类似。

II. 在A、B两地中间有一段路，路宽只能容纳一辆车通行，只有在A、B之间的M处有一个平台，可以实现两个方向的车错车。假设汽车不能倒车，试用信号量解决两个方向车通行中的同步互斥问题。



分析：这个问题在前面例子基础上复杂了一些，主要是允许不同方向的车同时进入AB区间。

（1）首先分析几类线程。两类车：A到B，和B到A。每类里面都有多个实例。
AB方向车的基本操作 {drive through AM; drive through MB;}
也就是A->B方向：先经过AM路段，到达M；再经过MB路段，到达B。

（2）分析互斥共享资源。AM之间的路，MB之间的路，对于同向都是读者锁，对于反向都是写者锁。

（3）分析依赖关系。无。

先分析第一种简单情况：为了保障双向的公平性且不死锁，同一个方向只能1辆车进入。这样两个方向相对独立，同方向的车按申请本方向入口信号量的顺序先后排队。例如所有AB方向的车竞争进入AB的入口权，并按顺序通过。获得进入权限的两辆车在内部竞争使用AM和BM。

Semaphore mutexAM=1, mutexBM=1 //互斥，道路AM使用权，道路BM使用权。
Semaphore tokenAB=1, tokenBA=1;　　//入口权限互斥，同方向只有1辆进入的权限。

```
LeftCar()
{
    wait(tokenAB);    //获得AB方向的入口权；
    wait(mutexAM);    //此时在入口等，获得AM路段使用权；
    drive through AM；
    signal(mutexAM);
    wait(mutexBM);    //此时在M处等，为了通过下一段，需要获得MB路段使用权；
```

```
    drive through MB;
    signal(mutexBM);
    signal(tokenAB);
}
```

RightCar()是对称的。

此方案是正确可行的，但是效率略低（A，B区间最大同时只容纳两辆车）。

再分析第二种情况：同方向允许多辆车进入。多辆车的一方具有优先级（也就是多辆车的一方连续通过，另一方允许进入的车只能在入口或者M等待）。
死锁分析：为了不死锁。要保证当一个方向有多车的时候，另一个方向最多有一辆车可以进入（这辆车可以在入口，或者M）。
有权进入AB区间的车之间竞争：两段路AM，BM。两段都是读写锁。

```
Semaphore mutexAM=1, mutexBM=1 //互斥，道路AM使用权，道路BM使用权。
Semaphore priority=1; //同方向超过2辆必须获得优先权，优先权一方可以连续上路。
Semaphore SemCountAB=1, SemCountBA=1;  //修改两个方向车数量的互斥。
int countAB=0, countBA=0; //两个方向车的数量;
Semaphore SemCountLeftAM=1, SemCountLeftBM=1, SemCountRightAM=1,
SemCountRightBM=1; ;  //两个方向想上两段路的车数量修改信号量。
int countLeftAM=0, countLeftBM=0,  countRightAM=0, countRightBM=0; //两个方向想上两段路车的数量;
```

//两个方向的第一辆车都可以直接上路，第一个到2辆车的方向，获得多车的优先权。（上一个版本是同方向竞争本方向的一个上路权tokenAB，现在是两个方向各自至少一个上路权，但竞争多车通过权。）

```
LeftCar()
{
wait(SemCountAB);    //
countAB++;  //AB方向的车数量
if (countAB=2)
    wait(priority);    // 两个方向谁先到达2辆车，谁获得优先权。
signal(SemCountAB);
//now at A
wait(SemCountLeftAM);    // 修改左边方向AM段车的数量需要互斥
countLeftAM++;
if (countLeftAM=1)
    wait(mutexAM);    // AM段第一辆车需要获得AM间路的使用权
signal(SemCountLeftAM);
drive through AM;
wait(SemCountLeftAM);    // 修改左边方向AM段车的数量需要互斥
countLeftAM--;
```

```
if (countLeftAM=0)
      signal(mutexAM); ;   // 最后一辆车需要释放AM间路的使用权
signal(SemCountLeftAM);
//now at M
wait(SemCountLeftBM);    // 修改左边方向BM段车的数量需要互斥
countLeftBM++;
if (countLeftBM=1)
      wait(mutexBM); ;   // 第一辆车需要获得BM间路的使用权
signal(SemCountLeftBM);
drive through MB;
wait(SemCountLeftBM);    // 修改左边方向BM段车的数量需要互斥
countLeftBM--;
if (countLeftBM=0)
      signal(mutexBM); ;   // 最后一辆车需要释放BM间路的使用权
signal(SemCountLeftBM);
//now at B
wait(SemCountAB);    //
countAB--;   //AB方向的车数量减少
if (countAB=1)
      signal(priority);    // 如果本方向只有一辆车，那主动让出优先权。
signal(SemCountAB);


}
```

RightCar() 对称。

注意：一定要先写出不带同步互斥的基础代码。然后再加代码。

主要要点：

（1）限制双向参与竞争上单行道的车的数量，避免死锁（如左>1，那么右<=1；如右>1，那么左<=1)。

理论上countAB=countLeftAM+countLeftBM，可以通过countLeftAM+countLeftBM=2来判定加优先锁，也就是把wait(priority)放到获取AM路段权限里面，把signal放到释放BM路权里面呢，但是这个操作要两个wait()，而且在两段路都要判定，所以如果在前后两段路两个信号量顺序不对，会导致两辆车各获得一个，导致死锁。

例如

```
wait(SemCountLeftAM);
countLeftAM++;
wait(SemCountLeftBM);
    if ((countLeftAM+countLeftBM)=2)
      wait(priority);
```

```
   if (countLeftAM=1)
        wait(mutexAM);
signal(SemCountLeftBM);
signal(SemCountLeftAM);

wait(SemCountLeftBM);     // 修改左边方向BM段车的数量需要互斥
countLeftBM--;
if (countLeftBM=0)
      signal(mutexBM); ;   // 最后一辆车需要释放BM间路的使用权
wait(SemCountLeftAM);
     if ((countLeftAM+countLeftBM)=1)
      signal(priority);
signal(SemCountLeftAM);
signal(SemCountLeftBM);
```

所以在AM段和BM如果两个wait顺序一致，是可以的。

（2）确定了双向的车数量后，有权利上路的车再内部竞争两段路。（两段路都是读写锁）

实际考试一般不会涉及太复杂的设计。能理解第一种情况就可以。