CSE_lab2

Exercise 1

思考分析Semaphore和Mutex的异同，举例说明二者适用的场景，在何种情况下可以达到相同的功能，在何种情况下具有各自的独特性，可以从二者考虑问题的方向、解决问题的表现、线程和资源的关系等角度分析，500字以内。

区别

semaphore本质是调度线程，即一些线程生产同时另一些线程消费，semaphore可以让生产和消费保持合乎逻辑的执行顺序

例如：a来自一个线程、b来自另一个线程，计算c=a+b也是一个线程

```
int a, b, c;
void geta()
{
    a = calculatea();
    semaphore_increase();
}
void getb()
{
    b = calculateb();
    semaphore_increase();
}
void getc()
{
    semaphore_decrease();
    semaphore_decrease();
    c = a + b;
}
t1 = thread_create(geta);
t2 = thread_create(getb);
t3 = thread_create(getc);
thread_join(t3);
```

从这个例子可以看出，只有当a和b都执行之后，c才能够执行，因为C需要消费由a、b供给的两个线程，保证了a和b均在C之前执行，且只要有1方未执行就不会继续执行，保证了**线程的执行顺序**

mutex是一种二元的锁，其最典型的用途是防止不停地去循环以判断一个共享资源是否满足某个条件。

例如：线程1：卖票，线程2：当票数为0时挂出“已售光”的牌子，线程3：申请更多的票

在正常情况下，每次执行循环的时候，就必须去判断当前票数是否为0，对线程2和线程3执行unlock -> lock操作，但实际上，大多数情况下，票数都不为0，因此会在占用了大量CPU资源的情况下，却什么都没有做。mutex可以避免这种情况，只有当票数为0时，mutex才会将线程2和线程3加入线程队列，执行unlock操作，大幅度的减少了CPU的浪费

相似点

mutex和semaphore在**保护共享资源时**可以达到相同的功能，此时需要让**semaphore的最大值为1**，即任意时刻都只有一个线程在运行，那么共享资源自然得到了保护。mutex从某种意义上来说可以看作是semaphore为1的情况。

结论

从根本来说，两者面向的设计需求是不同的，semaphore**更注重线程的有序执行，即调度线程，保护多个线程的执行的逻辑顺序**，而mutex更注重**保护共享资源、减少无意义的CPU损耗**

Exercise2

考虑下述程序

```
int x = 0, y = 0, z = 0;
sem lock1 = 1, lock2 = 1;
process foo{
    z = z + 2;
    P(lock1);
    x = x + 2;
    P(lock2);
    V(lcok1);
    y = y + 2;
    V(lock2);
}
process bar{
    P(lock2);
    y = y + 1;
    P(lock1);
    x = x + 1;
    V(lock1);
    V(lock2);
    z = z + 1;
}
```

1、何种情况下，程序会产生死锁

当 foo 进程执行完 `z = z+2` 后，会把lock1锁上，继续执行完 `x = x+2` 后，foo进程需要在占用 lock1 的情况下继续请求 lock2 的资源才能继续执行，且此时不会释放 lock1

另外，bar 进程在执行`y = y+1`后会把lock2锁上，而之后又需要lock1的资源

此时会发生死锁，foo进程执行到`x = x+2`, bar进程执行到 `y = y+1`。

2、在死锁状态下，x, y, z 的最终值可能是多少

x = 2

y =1

z =2

3、如果程序正常中值，x, y, z的最终值可能是多少（对z的赋值操作是不具有原子性的）

z = 1，2，3

y = 3

x = 3

由于z不具有原子性，因而在读取z的值的情况有3种

`z = z+1`读取了 `z = 0`，`z = z+2` 也读取了`z = 0`，前者先发生，后者后发生时，答案为`2`，反之为`1`

`z = z+1`,`z = z+2` 依序正常读取（先后无关），答案为`3`

Exercise 3:

多个进程共享U个资源，1个进程1次可以获取（request）1个资源，但是可能释放（release）多个。使用信号量做同步，合理进行P，V操作，实现request和release方法的伪代码，确保request和release操作是原子性的。提前声明和初始化你用到的变量。

```
int free = U;
# your variables
# 使用一个多元锁来保证资源的访问和释放具有原子性
# locks数组的所有初始值都为1
int[] locks = new int[U];

#假设request所请求的就是最外侧的资源
#<await (free > 0) free = free-1>
request(){
    # your code
    if free <= 0:
        return;
    # 当进程请求资源时，请求当前free数的空闲资源，并在lock数组中对这个值上锁，这样在其他进程再请求资源后释放资源时，不会存在这一资源被请求了却同时被释放的情况
    P(locks[free-1]);
    # 请求free处的资源
    self.requestResources(free);
    free = free-1;
    # 将锁打开，此处虽然该资源已被获取，但由于free的值小于锁的位置，因此，这个位置的锁开放表示该位置可以释放资源，同时也可以避免死锁的出现
    V(locks[free]);
}

# <free = free + number;>
release(int number){
 # your code
    if number == 0:
        return;
    for i = 0:number:
        # 在释放资源时对这一资源的访问上锁，防止对象请求资源发生变化，同时也保证了这一资源被请求时不会发生变化
        P(locks[free]);
        # 释放准备释放的第i个资源到free的位置
        self.releaseResources(i);
        free ++;
        # 将锁打开，保证这一资源可以被访问
        V(locks[free-1]);
}
```

Exercise 4

哲学家吃饭问题：

5个哲学家围坐在一起吃饭（eating）和思考（thinking）。有5只叉子可以供他们共享，每个哲学家需要拿起身旁的2只叉子进行吃饭，吃完之后会放下叉子，进行思考，然后叉子会被别的哲学家使用。 使用Java语言模拟上述场景，保证每个哲学家都能吃到饭，同时避免死锁。

具体代码见src文件

部分代码解释

```
    @Override
    public void run() {
        try {
            while (true) {
                doAction(System.nanoTime() + ": Thinking"); // thinking
                // your code
                /* 思路如下：
                 *    产生所有哲学家都进入想要别人的叉子的死锁状况，存在于当每个哲学家都拿了下一位/上一位哲学家空着的手的叉子，即所有人都拿了左手/右手的叉子。为了避免这种情况，我们可以将哲学家分类为奇数和偶数，这样一来，奇数和偶数的哲学家交替，分别先拿左手/右手的叉子，便不会出现连续的哲学家拿了相同手的叉子而渴望对方拿了的叉子而产生的死锁
                 */
                if (num % 2 == 0) {
                    //对于编号为偶数的哲学家，他们先举起右手的叉子
                    synchronized (leftFork) {
                        pick_left();
                        synchronized (rightFork) {
                            pick_right();
                            eat();
                        }
                    }
                } else {
                    //对于编号为奇数的哲学家，他们先举起左手的叉子
                    synchronized (rightFork) {
                        pick_right();
                        synchronized (leftFork) {
                            pick_left();
                            eat();
                        }
                    }
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
```