

信号量

抽象数据结构

- 一个整型(sem) 两个原子操作
- P() sem减1, 如果sem<0,等待, 否则继续
- V() sem加1, 如果sem<=0,唤醒一个等待的p

特征

- 信号量是整数
- 信号量是被保护的变量
 - 初始化完成后, 唯一改变一个信号量的办法就是通过P(),V()
 - 操作必须是原子
- P()能够阻塞, V()不会阻塞
- 我们假定信号量是"公平的"
 - FIFO先进先出经常被使用

两种类型信号量

- 二进制信号量: 可以0或1
 - 一般/计数信号量: 可取任何非负数
- 以上两种相互表现(给定一个可以实现另一个)

作用

- 互斥
- 条件同步, 一个线程等待另一个线程

互斥:

与lock的作用完全一样, 通过pv操作来实现加锁解锁

条件同步场景:

生产者消费者场景

- ◆ 一个线程等待另一个线程处理事情
 - 比如生产东西或消费东西
 - 互斥（锁机制）是不够的
- ◆ 例如：有界缓冲区的生产者 - 消费者问题
 - 一个或多个生产者产生数据将数据放在一个缓冲区里
 - 单个消费者每次从缓冲区取出数据
 - 在任何一个时间只有一个生产者或消费者可以访问该缓冲区



在此场景中则有三个要求：

1. 在任意一个时间只能有一个线程操作缓冲区(互斥)
2. 当缓冲区为空,消费者必须等待生产者(条件同步)
3. 当缓冲区已满，生产者必须等待消费者(条件同步)

所以我们可以使用三个信号量来满足以上三个要求

- 二进制信号量来进行互斥
- 一般信号量fullBuffer
- 一般信号量emptyBuffer

信号量实现

伪代码

```
class Semaphore {  
    int sem; //信号量  
    WaitQueue q; //等待的进程  
}
```

使用硬件原语

- 禁用中断
- 原子指令(test-and-set)

类似锁

例如使用“禁止中断”

```
Semaphore::P(){
    sem--;
    if (sem < 0){
        Add this thread t to q;
        block(p);
    }
}
```

```
Semaphore::V(){
    sem++;
    if (sem <= 0){
        Remove a thread t from q;
        wakeup(t);
    }
}
```

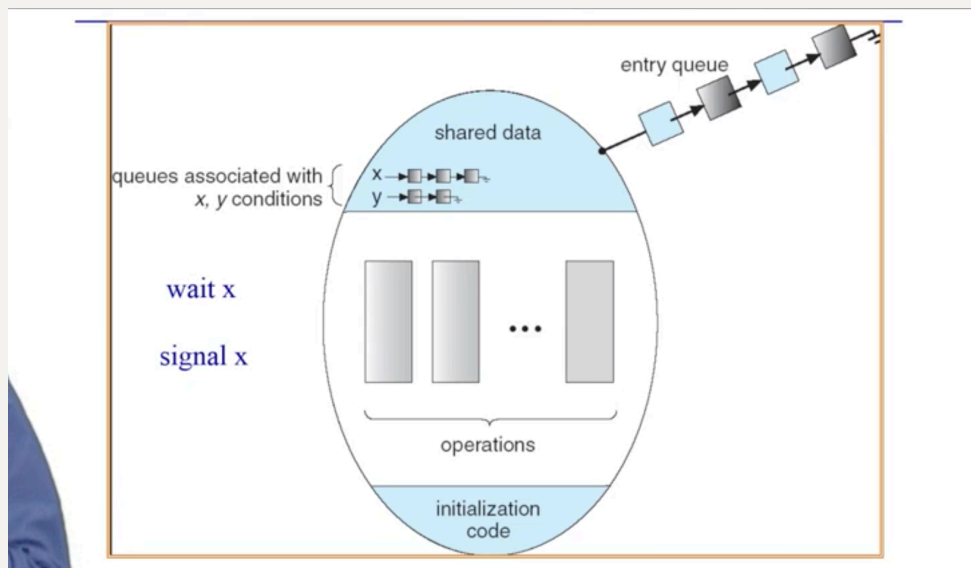
可能有死锁问题

管程

分离互斥和条件同步的关注

什么是管程

- 一个锁：指定临界区
- 0或者多个条件变量：等待/通知信号量用于管理并发访问共享数据



◆ Lock

- `Lock::Acquire()` - 等待直到锁可用，然后抢占锁
- `Lock::Release()` - 释放锁，唤醒等待者如果有

◆ Condition Variable

- 允许等待状态进入临界区
 - ❖ 允许处于等待（睡眠）的线程进入临界区
 - ❖ 某个时刻原子释放锁进入睡眠
- `Wait()` operation
 - ❖ 释放锁，睡眠，重新获得锁返回后
- `Signal()` operation (or `broadcast()` operation)
 - ❖ 唤醒等待者（或者所有等待者），如果有

◆ 实现

- 需要维持每个条件队列
- 线程等待的条件等待`signal()`

```
Class Condition {  
    int numWaiting = 0;  
    WaitQueue q;  
}
```

```
Condition::Wait(lock){  
    numWaiting++;  
    Add this thread t to q;  
    release(lock);  
    schedule(); //need mutex  
    require(lock);  
}
```

```
Condition::Signal(){  
    if (numWaiting > 0) {  
        Remove a thread t from q;  
        wakeup(t); //need mutex  
        numWaiting--;  
    }  
}
```

这个思想在java的juc包下的 `LinkedBlockingDeque` 被使用到

了, `LinkedBlockingDeque` 是使用的 `ReentrantLock` 的 `Condition` 来实现的(基于AQS) AQS 是操作系统管程的实现? (猜测)

```

class BoundedBuffer {
    ...
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}

```

```

BoundedBuffer::Deposit(c) {
    lock->Acquire();
    while (count == n)
        notFull.Wait(&lock);
    Add c to the buffer;
    count++;
    notEmpty.Signal();
    lock->Release();
}

```

```

BoundedBuffer::Remove(c) {
    lock->Acquire();
    while (count == 0)
        notEmpty.Wait(&lock);
    Remove c from buffer;
    count--;
    notFull.Signal();
    lock->Release();
}

```

- ◆ Hansen-style

- Signal is only a "hint" that the condition may be true
- Need to check again

- ◆ Benefits

- Efficient implementation

```

Hansen-style :Deposit(){
    lock->acquire();
    while (count == n) {
        notFull.wait(&lock);
    }
    Add thing;
    count++;
    notEmpty.signal();
    lock->release();
}

```

- ◆ Hoare-style

- Cleaner, good for proofs
- When a condition variable is signaled, it does not change

- ◆ But

- Inefficient implementation

```

Hoare-style: Deposit(){
    lock->acquire();
    if (count == n) {
        notFull.wait(&lock);
    }
    Add thing;
    count++;
    notEmpty.signal();
    lock->release();
}

```