Lock

概念

临界区（Critical Section）：访问和操作共享数据的代码段

比如：用10个线程计算Sum=1+2+…+100,

- 线程1计算S1=1+2+…+10, Sum+=S1;
- 线程2计算S2=11+12+…+20, Sum+=S2;
- …
- 线程10计算S10=90+91+…+100, Sum+=S10;

概念

- **原子操作**：执行过程不能被打断的操作；
- **粒度**：在本topic中理解为原子操作的size；

为何要加锁？

假设你的银行账号有1050块钱，你要在ATM取100块，同一时刻你女票在买买买要付款1000。

理想情况：你或女票操作失败；

实际情况：（有可能）你们同时提交了请求，线程1看到你余额为1050，给你吐出100块，然后更新余额为950；线程2同时看到你余额为1050，你女票支付成功，更新余额为50。银行：？？？

加锁：让{请求，查询余额，结算，更新余额}变成原子操作

锁的作用

- 让某些操作所在的代码块具有原子性；
- 保证临界区最多只有一个线程（或者有限个读线程）在访问；
- 保护的是数据而不是代码；
- Linux的几种锁：**自旋锁**（Spin Lock）、**信号量**（Semaphore）、**互斥锁**（Mutex）

自旋锁（Spin Lock）

- 内核中最常见的锁，最多只能被一个可执行线程持有；
- 如果当前自旋锁不可用（被占用），线程会进行忙循环（占用CPU），直到锁可用；
- 不应被长时间占用，适用于短期间内的轻量级加锁；
- 不可递归（死锁）。

函数：

spin_lock()、spin_unlock()

```
DEFINE_SPINLOCK(lock);
spin_lock(&lock);
/*
 *临界区
*/
spin_unlock(&unlock);
```

读写自旋锁

锁的用途细化为read和write两种场景：

- 写操作时，不能有其他线程并发地写同一套被保护的数据；
- 读操作时，允许多线程并发地读同一套被保护的数据，但不允许并发写；

函数：

read_lock()、read_unlock();

write_lock()、write_unlock();

信号量（Semaphore）

- 一种睡眠锁，线程在等待时睡眠，不占用CPU；
- 上下文切换、睡眠、维护等待队列以及唤醒等会造成额外开销；
- 线程在睡眠，只能靠进程上下文唤醒；
- 等待信号量锁时不能占用自旋锁，因为会陷入睡眠；
- 并非只允许一个线程访问临界区，可以设置count；

函数：down()、up()

读写信号量

- 与读写自旋锁相似；
- count被固定为1（对写者）；

函数：

down_read()、up_read();

down_write()、up_write();

互斥锁（Mutex）

- 信号量的简化版，count被设置为1；
- 特性跟信号量基本一样；

函数：

mutex_lock()、mutex_unlock()

读写锁实例

- 读者优先（写线程饥饿）
- 写者优先

```
pthread_mutex_t rd = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t wr = PTHREAD_MUTEX_INITIALIZER;

int readCount = 0;
void* writer(void *arg)
{
    ...
    while (n--)
    {
        ...
        pthread_mutex_lock(&wr);
        //写数据
        pthread_mutex_unlock(&wr);
    }
}
```

```
void* reader(void *arg)
{
    ...
    while (n--)
    {
        pthread_mutex_lock(&rd);
        readCount++;
        if( readCount == 1)
        {
            pthread_mutex_lock(&wr);
        }
        pthread_mutex_unlock(&rd);
        //读数据
        pthread_mutex_lock(&rd);
        readCount--;
        if (readCount == 0)
        {
            pthread_mutex_unlock(&wr);
        }
        pthread_mutex_unlock(&rd);
    }
}
```

```
pthread_mutex_t rd = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t wr = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t priority = PTHREAD_MUTEX_INITIALIZER;

void* writer(void *arg)
{
    int n = M;
    while (n--)
    {
        pthread_mutex_lock(&priority);
        pthread_mutex_lock(&wr);
        //写数据
        pthread_mutex_unlock(&wr);
        pthread_mutex_unlock(&priority);
    }
}
```

```
void* reader(void *arg)
{
    ...
    while (n--)
    {
        pthread_mutex_lock(&priority);
        pthread_mutex_lock(&rd);
        readCount++;
        if( readCount == 1)
        {
            pthread_mutex_lock(&wr);
        }
        pthread_mutex_unlock(&rd);
        pthread_mutex_unlock(&priority);
        //读数据
        pthread_mutex_lock(&rd);
        readCount--;
        if (readCount == 0)
        {
            pthread_mutex_unlock(&wr);
        }
        pthread_mutex_unlock(&rd);
    }
}
```

死锁

- 需要有一个或多个线程和一个或多个资源；
- 每个线程都在等待其中一个资源，但所有的资源都被占用；

线程1：

获得锁A

试图获得锁B

等待锁B

线程2：

获得锁B

试图获得锁A

等待锁A

Thanks !