学习目标

1. 熟练掌握互斥量的使用
2. 说出什么叫死锁以及解决方案
3. 熟练掌握读写锁的使用
4. 熟练掌握条件变量的使用
5. 理解条件变量实现的生产消费者模型
6. 理解信号量实现的生产消费者模型

1 - 互斥量(互斥锁)

1. 互斥锁类型:
   创建一把锁：pthread_mutex_t mutex;
2. 互斥锁的特点:
3. 使用互斥锁缺点?
   - 串行
4. 互斥锁的使用步骤:
   - 创建互斥锁: pthread_mutex_t mutex;
   - 初始化: pthread_mutex_init(&mutex, NULL);
   - 找到线程共同操作的共享数据
     - 加锁: pthread_mutex_lock(&mutex); // 阻塞线程
     - pthread_mutex_trylock(&mutex); // 如果锁上锁，直接返回，不阻塞
       - .....共享数据操作
     - 解锁: pthread_mutex_unlock(&mutex);
       - 阻塞在锁上的线程会被唤醒
   - 销毁: pthread_mutex_destroy(&mutex);
5. 互斥锁相关函数:
   - 初始化互斥锁
     ```
     pthread_mutex_init(
         pthread_mutex_t *restrict mutex,
         const pthread_mutexattr_t *restrict attr
     );
     ```
   - 销毁互斥锁
     ```
     pthread_mutex_destroy(pthread_mutex_t *mutex);
     ```
   - 加锁
     ```
     pthread_mutex_lock(pthread_mutex_t *mutex);
     ```
     - mutex：
       - 没有被上锁，当前线程会将这把锁锁上
       - 被锁上了：当前线程阻塞
         - 锁被打开之后，线程解除阻塞
   - 尝试加锁，失败返回，不阻塞
     ```
     pthread_mutex_trylock(pthread_mutex_t *mutex);
     ```
     - 没有锁上：当前线程会给这把锁加锁
     - 如果锁上了：不会阻塞，返回

```
if(pthread_mutex_trylock(&mutex)==0)
{
    // 尝试加锁，并且成功了
    // 访问共享资源
}
else
{
    // 错误处理
    // 或者 等一会，再次尝试加锁
}
```

- 解锁

  pthread_mutex_unlock(pthread_mutex_t *mutex);

如果我们想使用互斥锁同步线程:

所有的线程都需要加锁

2 - 死锁

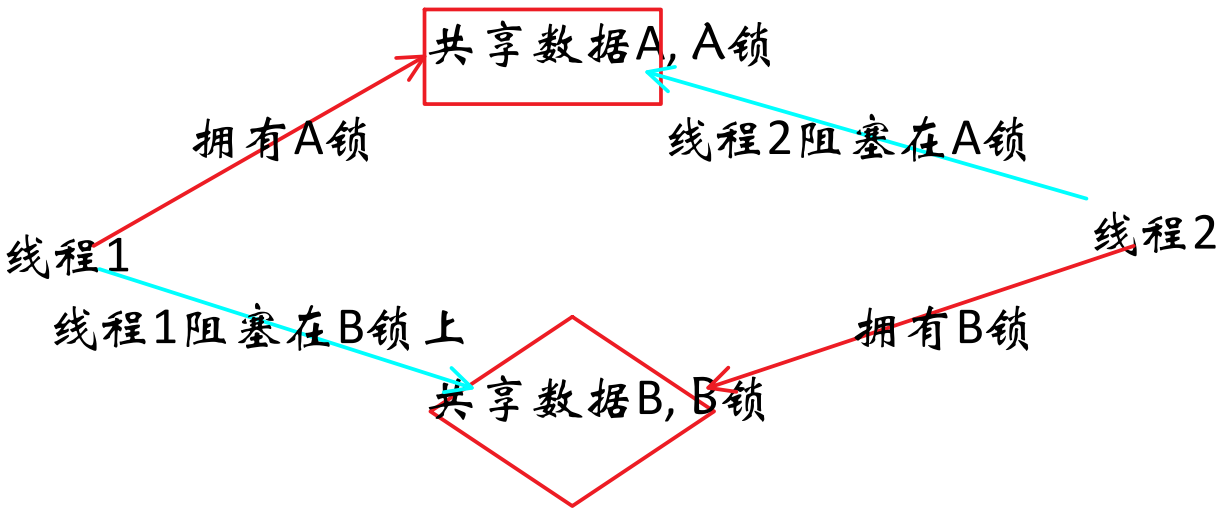
造成死锁的原因:

1. 自己锁自己

```
for(int i=0; i<MAX; ++i)
{
    // 加锁
    pthread_mutex_lock(&mutex);
    pthread_mutex_lock(&mutex);
    int cur = number;
    cur++;
    number = cur;
    printf("Thread A, id = %lu, number = %d\n", pthread_self(), number);
    // 解锁
    pthread_mutex_unlock(&mutex);
    usleep(10);
}
```

操作做完成之后,一定要解锁

2.



1. 在访问其他共享数据的时候,先把自己的锁解锁
2. 所有的线程顺序的去访问共享数据

3 - 读写锁

1. 读写锁是几把锁?
    - 一把锁
2. 读写锁的类型:
    - 锁读操作
    - 锁写操作
    - 不加锁
3. 读写锁的状态:
4. 读写锁的特性:
    - 线程A加读锁成功,又来了三个线程,做读操作,可以加锁成功
        - 加读锁,可以并行
    - 线程A加写锁成功,又来了三个线程,做读操作,三个线程阻塞
        - 写操作的时候不能读,写的时候是串行的
    - 线程A加读锁成功,又来了B线程加写锁阻塞,又来了C线程加读锁阻塞
        - 写的优先级高
    - 读共享,写独占,写的优先级高
5. 读写锁场景练习:
    - 线程A加写锁成功,线程B请求读锁
        - 线程b阻塞
    - 线程A持有读锁,线程B请求写锁
        - 线程B阻塞
    - 线程A拥有读锁,线程B请求读锁
        - 线程B加锁成功
    - 线程A持有读锁,然后线程B请求写锁,然后线程C请求读锁
        - BC阻塞
        - A解锁B加锁成功,C阻塞
        - B解锁C加锁成功
    - 线程A持有写锁,然后线程B请求读锁,然后线程C请求写锁
        - BC阻塞
        - A解锁,C加锁成功,B阻塞
        - C解锁,B加锁
6. 读写锁的适用场景?
    - 读 - 并行
    - 写 - 串行

- 读的操作>写操作的

7. 主要操作函数
   - 初始化读写锁

     ```
     pthread_rwlock_init(
         pthread_rwlock_t *restrict rwlock,
         const pthread_rwlockattr_t *restrict attr
     );
     ```

   - 销毁读写锁

     ```
     pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
     ```

   - 加读锁

     ```
     pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
     ```

   - 尝试加读锁

     ```
     pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
     ```

   - 加写锁

     ```
     pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
     ```

   - 尝试加写锁

     ```
     pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
     ```

   - 解锁

     ```
     pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
     ```

8. 练习：
   - 3个线程不定时写同一全局资源，5个线程不定时读同一全局资源

4 - 条件变量

1. 条件变量是锁吗?
   - 不是锁,但是条件变量能够阻塞线程
   - 使用条件变量 + 互斥量
     - 互斥量: 保护一块共享数据
     - 条件变量: 引起阻塞
       - 生产者和消费者模型
2. 条件变量的两个动作?
   - 条件不满,阻塞线程
   - 当条件满足,通知阻塞的线程开始工作
3. 条件变量的类型: pthread_cond_t;
4. 主要函数:
   - 初始化一个条件变量

     ```
     pthread_cond_init(

         pthread_cond_t *restrict cond,
         const pthread_condattr_t *restrict attr
     );
     ```
   - 销毁一个条件变量

     ```
     pthread_cond_destroy(pthread_cond_t
     *cond);
     ```
   - 阻塞等待一个条件变量

     ```
     pthread_cond_wait(
                     pthread_cond_t *restrict cond,
          pthread_mutex_t *restrict mutex
     );
     ```
   - 限时等待一个条件变量

```
pthread_cond_timedwait(
    pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime
);
```

- 唤醒至少一个阻塞在条件变量上的线程
  `pthread_cond_signal(pthread_cond_t *cond);`
- 唤醒全部阻塞在条件变量上的线程
  `pthread_cond_broadcast(pthread_cond_t *cond);`

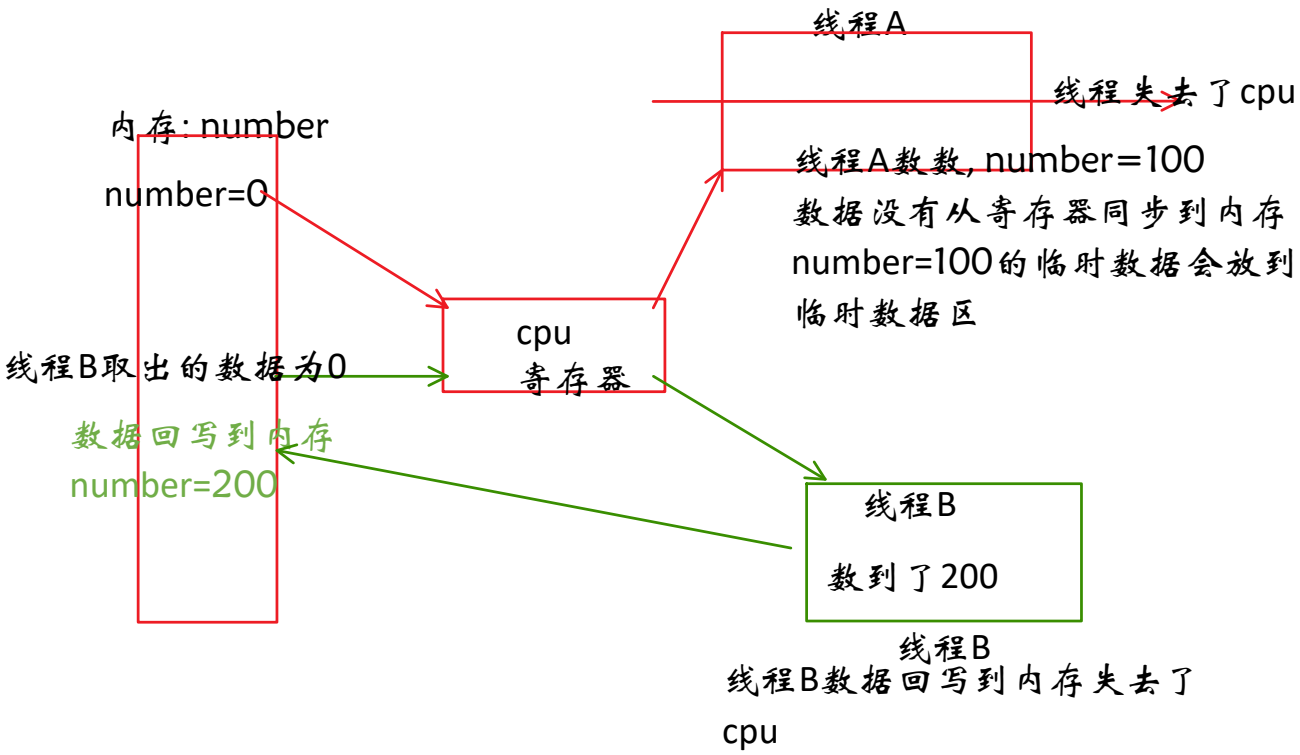
5. 练习
   - 使用条件变量实现生产者,消费者模型

5 - 信号量(信号灯)

1. 头文件 - semaphore.h
2. 信号量类型
   - sem_t sem; | | int
   - 加强版的互斥锁
3. 主要函数
   - 初始化信号量
     sem_init(sem_t *sem, int pshared, unsigned int value);
     - 0 - 线程同步
     - 1 - 进程同步
     - value - 最多有几个线程操作共享数据
   - 销毁信号量
     sem_destroy(sem_t *sem);
   - 加锁 --
     sem_wait(sem_t *sem);
     调用一次相当于对sem做了--操作
     如果sem值为0, 线程会阻塞
   - 尝试加锁
     sem_trywait(sem_t *sem);
     - sem == 0, 加锁失败, 不阻塞, 直接返回
   - 限时尝试加锁
     sem_timedwait(sem_t *sem, xxxxx);
   - 解锁 ++
     sem_post(sem_t *sem);
     对sem做了++操作
4. 练习:
   - 使用信号量实现生产者,消费者模型

6-哲学家就餐模型

五个哲学家,围着一张桌子吃饭,每个哲学家只有一根筷子,需要使用旁边人的筷子才能把饭吃到嘴里.
抢到筷子的吃饭,没抢到的思考人生.
使用多线程实现多线程实现哲学家交替吃饭的模型



造成数据混乱的原因：

- 多个线程操作了共享数据
- cpu的调度文件
- 提供一套同步机制

什么叫线程同步？

让多个线程协同步调，先后处理某件事情

2017年1月9日 10:23