09-线程同步

学习目标：

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

1 互斥锁

1.1 互斥锁的使用步骤

- 第 1 步：创建一把互斥锁
  - pthread_mutex_t mutex;
- 初始化互斥锁
  - pthread_mutex_init(&mutex);---相当于 mutex=1
- 在代码中寻找共享资源（也称为临界区）

  pthread_mutex_lock(&mutex); -- mutex = 0

  [临界区代码]

  pthread_mutex_unlock(&mutex); -- mutex = 1
- 释放互斥锁资源

  pthread_mutex_destroy(&mutex);

注意：必须在所有操作共享资源的线程上都加上锁否则不能起到同步的效果。

1.2 练习

- 编写思路：

1 定义一把互斥锁，应该为一全局变量

  pthread_mutex_t mutex;

2 在 main 函数中对 mutex 进行初始化

  pthread_mutex_init(&mutex, NULL);

3 创建两个线程，在两个线程中加锁和解锁

4 主线程释放互斥锁资源

  pthread_mutex_destroy(&mutex);

```c
void *mythread1(void *args)
{
    while(1)
    {
        //加锁
        pthread_mutex_lock(&mutex);

        printf("hello ");
        sleep(rand()%3);
        printf("world\n");

        //解锁
        pthread_mutex_unlock(&mutex);
        sleep(rand()%3);
    }

    pthread_exit(NULL);
}
```

```c
void *mythread2(void *args)
{
    while(1)
    {
        //加锁
        pthread_mutex_lock(&mutex);

        printf("HELLO ");
        sleep(rand()%3);
        printf("WORLD\n");

        //解锁
        pthread_mutex_unlock(&mutex);
        sleep(rand()%3);
    }

    pthread_exit(NULL);
}
```

1.3 死锁

死锁并不是 linux 提供给用户的一种使用方法，而是由于用户使用互斥锁不当引起的一种现象。

❁ 常见的死锁有两种：

- 第一种：自己锁自己，如下图代码片段

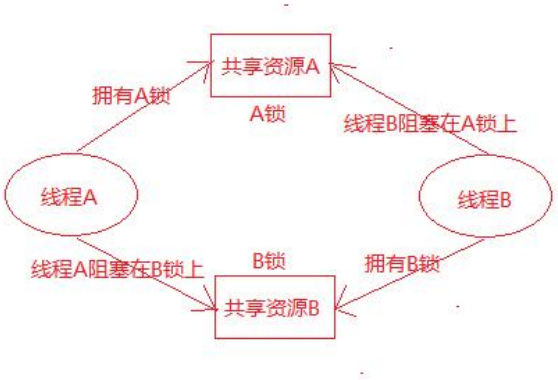
```c
void *mythread1(void *args)
{
    while(1)
    {
        //加锁
        pthread_mutex_lock(&mutex);
        pthread_mutex_lock(&mutex);

        printf("hello ");
        sleep(rand()%3);
        printf("world\n");

        //解锁
        pthread_mutex_unlock(&mutex);
        sleep(rand()%3);
    }

    pthread_exit(NULL);
}
```

- 第二种 线程 A 拥有 A 锁，请求获得 B 锁；线程 B 拥有 B 锁，请求获得 A 锁，这样造成线程 A 和线程 B 都不释放自己的锁，而且还想得到对方的锁，从而产生死锁，如下图所示：



- 如何解决死锁：
  - 让线程按照一定的顺序去访问共享资源
  - 在访问其他锁的时候，需要先将自己的锁解开
  - 调用 pthread_mutex_trylock，如果加锁不成功会立刻返回

2 读写锁

- 什么是读写锁
  - 读写锁也叫共享-独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。**写独占、读共享。**
- 读写锁使用场合
  - 读写锁非常适合于对数据结构读的次数远大于写的情况。
- 读写锁特性
  - 读写锁是“写模式加锁”时，解锁前，所有对该锁加锁的线程都会被阻塞。
  - 读写锁是“读模式加锁”时，如果线程以读模式对其加锁会成功；如果线程以写模式加锁会阻塞。
  - 读写锁是“读模式加锁”时， 既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求。优先满足写模式锁。读锁、写锁并行阻塞，写锁优先级高
- 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

- B 阻塞，c 阻塞 - 写的优先级高
- A 解锁，B 线程加写锁成功，C 继续阻塞
- B 解锁，C 加读锁成功

- 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 加写锁成功，B 继续阻塞
  - C 解锁，B 加读锁成功

- 读写锁总结

  读并行，写独占，当读写同时等待锁的时候写的优先级高

- 读写锁主要操作函数

  - 定义一把读写锁
    - pthread_rwlock_t rwlock;
  - 初始化读写锁
    - int pthread_rwlock_init(
      pthread_rwlock_t *restrict rwlock,
      const pthread_rwlockattr_t *restrict attr);
    - 函数参数
      - rwlock-读写锁
      - attr-读写锁属性，传 NULL 为默认属性

  - 销毁读写锁

    int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
  - 加读锁

    int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
  - 尝试加读锁

    int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
  - 加写锁

    int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
  - 尝试加写锁

    int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
  - 解锁

    int pthread_rwlock_unlock(&pthread_rwlock_t *rwlock);

- 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。

3 条件变量

- 条件本身不是锁！但它也可以造成线程阻塞。通常与互斥锁配合使用。给多线程提供一个会合的场所。
  - 使用互斥量保护共享数据;
  - 使用条件变量可以使线程阻塞，等待某个条件的发生，当条件满足的时候解除阻塞.

- 条件变量的两个动作:
  - 条件不满足，阻塞线程
  - 条件满足，通知阻塞的线程解除阻塞，开始工作.

- 条件变量相关函数
  - pthread_cond_t cond;
    - 定义一个条件变量
  - int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
    - 函数描述:初始化条件变量
    - 函数参数:
      cond: 条件变量
      attr: 条件变量属性，通常传 NULL
    - 函数返回值:成功返回 0，失败返回错误号
  - int pthread_cond_destroy(pthread_cond_t *cond);
    - 函数描述: 销毁条件变量
    - 函数参数: 条件变量
    - 返回值: 成功返回 0，失败返回错误号
  - int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
    - 函数描述: 条件不满足，引起线程阻塞并解锁;
      条件满足，解除线程阻塞，并加锁
    - 函数参数:
      cond: 条件变量
      mutex: 互斥锁变量
    - 函数返回值: 成功返回 0，失败返回错误号
  - int pthread_cond_signal(pthread_cond_t *cond);
    - 函数描述: 唤醒至少一个阻塞在该条件变量上的线程
    - 函数参数: 条件变量
    - 函数返回值: 成功返回 0，失败返回错误号

4 使用条件变量的代码片段

```c
void *producer(void *args)
{
    NODE *pNode = NULL;
    while(1)
    {
        pNode = (NODE *)malloc(sizeof(NODE));
        if(pNode==NULL)
        {
            exit(1);
        }
        pNode->data = rand()%1000;

        pthread_mutex_lock(&mutex);

        pNode->next = head;
        head = pNode;
        printf("P:[%d]\n", head->data);

        pthread_mutex_unlock(&mutex);

        pthread_cond_signal(&cond);

        sleep(rand()%2);
    }

    pthread_exit(NULL);
}
```

```c
void *consumer(void *args)
{
    NODE *pNode = NULL;
    while(1)
    {
        pthread_mutex_lock(&mutex);

        if(head==NULL)
        {
            pthread_cond_wait(&cond, &mutex);
        }

        printf("C:[%d]\n", head->data);
        pNode = head;
        head = head->next;
        free(pNode);
        pNode = NULL;

        pthread_mutex_unlock(&mutex);
        sleep(rand()%3);
    }

    pthread_exit(NULL);
}
```

上述代码中，生产者线程调用 pthread_cond_signal 函数会使消费者线程在 pthread_cond_wait 处解除阻塞。

4 信号量

1 信号量介绍

信号量相当于多把锁，可以理解为是加强版的互斥锁

2 相关函数

- 定义信号量 sem_t sem;
- int sem_init(sem_t *sem, int pshared, unsigned int value);
  - 函数描述：初始化信号量
  - 函数参数：

    sem：信号量变量

    pshared：0 表示线程同步，1 表示进程同步

    value：最多有几个线程操作共享数据
  - 函数返回值:成功返回 0，失败返回-1，并设置 errno 值
- int sem_wait(sem_t *sem);
  - 函数描述：调用该函数一次，相当于 sem--，当 sem 为 0 的时候，引起阻塞
  - 函数参数：信号量变量
  - 函数返回值：成功返回 0，失败返回-1，并设置 errno 值
- int sem_post(sem_t *sem);
  - 函数描述：调用一次，相当于 sem++
  - 函数参数：信号量变量
  - 函数返回值：成功返回 0，失败返回-1，并设置 errno 值
- int sem_trywait(sem_t *sem);
  - 函数描述：尝试加锁，若失败直接返回，不阻塞
  - 函数参数：信号量变量
  - 函数返回值：成功返回 0，失败返回-1，并设置 errno 值
- int sem_destroy(sem_t *sem);

- 函数描述：销毁信号量
- 函数参数：信号量变量
- 函数返回值：成功返回 0，失败返回-1，并设置 errno 值

3 信号量代码片段：

```c
//生产者线程处理函数
void *producer(void *args)
{
    NODE *pNode = NULL;

    while(1)
    {
        pNode = (NODE *)malloc(sizeof(NODE));
        if(pNode==NULL)
        {
            perror("malloc error\n");
            exit(1);
        }
        pNode->data = rand()%1000;

        //sem_producer--, 若为0则阻塞
        sem_wait(&sem_producer);

        pNode->next = head;
        head=pNode;
        printf("P:[%d]\n", head->data);

        //sem_consumer++
        sem_post(&sem_consumer);

        sleep(rand()%3);
    }
}
```

```c
//消费者线程处理函数
void *consumer(void *args)
{
    NODE *pNode = NULL;
    while(1)
    {
        //sem_consumer--, 若为0则阻塞
        sem_wait(&sem_consumer);

        printf("C:[%d]\n", head->data);
        pNode = head;
        head = head->next;

        //sem_producer++
        sem_post(&sem_producer);

        free(pNode);
        pNode = NULL;

        sleep(rand()%3);
    }
}
```