线程同步思想

互斥量(互斥锁)

1. 互斥锁类型: 创建一把锁:pthread_mutex_t muext;
2. 互斥锁的特点: ○ 多个线程访问共享数据的时候是串行的
3. 使用互斥锁缺点? ○ 效率低
4. 互斥锁的使用步骤:
    - 创建互斥锁:pthread_mutex_t mutex;
    - 初始化这把锁:pthread_mutex_init(&mutex, NULL); -- mutex = 1 -- NULL是一个锁属性,一般保持为空就好.
    - 寻找共享资源:
        - 操作共享资源的代码之前加锁
            - pthread_mutex_lock(&mutex); -- mutex = 0
    - 。。。。。。
    - 。。。。。。 临界区(会访问到共享资源的就是临界区)
    - 

      注意这个临界区的情况.
    - pthread_mutex_unlock(&mutex); -- mutex = 1
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

叔叔例子:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>

#define MAX 10000
// 全局变量
int number;
//创建一把互斥锁
pthread_mutex_t mutex;
// 线程处理函数
void* funcA_num(void* arg)
{
    for(int i=0; i<MAX; ++i)
    {
        //枷锁,访问之前
        //如果Mutex被锁上了,,代码阻塞在当前位置
        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);//不让睡就会主动让cpu
    }
```

```
    return NULL;
}

void* funcB_num(void* arg)
{
    for(int i=0; i<MAX; ++i)
    {
        pthread_mutex_lock(&mutex);
        int cur = number;
        cur++;
        number = cur;
        printf("Thread B, id = %lu, number = %d\n" , pthread_self(), number);
        pthread_mutex_unlock(&mutex);
        usleep(10);
    }

    return NULL;
}

int main(int argc, const char* argv[])
{
    pthread_t p1, p2;
    //初始化互斥锁
    pthread_mutex_init(&mutex,NULL);
    // 创建两个子线程
    pthread_create(&p1, NULL, funcA_num, NULL);
    pthread_create(&p2, NULL, funcB_num, NULL);

    // 阻塞，资源回收
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    //SHIFANG
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

原子操作

死锁

如何解决:

- 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
- trylock (访问会失败,会做相应的处理动作)

读写锁

1. 读写锁是几把锁?
    - 一把锁
    - pthread_rwlock_t lock;
2. 读写锁的类型:
    - 读锁 - 对内存做读操作
    - 写锁 - 对内存做写操作
3. 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
        - 读共享 - 并行处理
    - 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
        - 写独占
    - 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
        - 读写不能同时
        - 写的优先级高,写和读有个优先级的比较问题
        - 这个也可以按照现实意义理解,就是你没有写完成之前,读是不能进行,因为,你要读你写完的内容
4. 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
5. 读写锁的适用场景?
    - 互斥锁 - 读写串行
    - 读写锁:
        - 读:并行
        - 写:串行

- 程序中的读操作>>写操作的时候
  - 这样能提高程序的效率

6. 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);

- 阻塞:之前对这把锁加的写锁的操作

   - 尝试加读锁

pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

- 加锁成功:0

- 失败:错误号

   - 加写锁

pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

   - 上一次加锁写锁,还没有解锁的时候
     - 上一次加读锁,没解锁
   - 尝试加写锁

pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

   - 解锁

pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

7. 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
     - 先不加锁

```
//
// Created by bruce on 18-5-20.
//

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>

int number=0;

pthread_rwlock_t  lock;
```

```c
void* write_func(void*arg)
{
    while(1)
    {
        pthread_rwlock_wrlock(&lock);
        number++;
        printf("===write:%lu,%d\n",pthread_self(),number);
        pthread_rwlock_unlock(&lock);
        usleep(500);
    }
    return NULL;
}
void* read_func(void*arg)
{
    while(1)
    {
        //加读锁
        pthread_rwlock_rdlock(&lock);
        printf("===read:%lu,%d\n",pthread_self(),number);
        pthread_rwlock_unlock(&lock);
        usleep(500);
    }
    return NULL;
}
int main()
{
    pthread_t p[8];
    pthread_rwlock_init(&lock,NULL);
    for(int i=0;i<3;++i)
    {
        pthread_create(&p[i],NULL,write_func,NULL);
    }
    for(int i=3;i<8;++i)
    {
        pthread_create(&p[i],NULL,read_func,NULL);
    }
    //阻塞回收
    for(int i=0;i<8;++i)
    {
        pthread_join(p[i],NULL);
    }
    pthread_rwlock_destroy(&lock);
    return 0;
}
```

8. 读写锁,互斥锁
    - 阻塞线程
    - 不是什么时候都能阻塞线程
    - 链表 Node *head = NULL

```
    - while(head == NULL)
    {
    // 我们想让代码在这个位置阻塞
    // 等待链表中有了节点之后再继续向下运行
    // 使用了后边要讲的条件变量 - 阻塞线程
}
// 链表不为空的处理代码
。。。。。。
。。。。。。
```

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(临时不满足条件的阻塞)
       - 生产者和消费者模型
2. 条件变量的两个动作?
   - 条件不满足, 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 cond;
4. 主要函数:
   - 初始化一个条件变量 - condtion

   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**);
   - 阻塞线程

        - 将已经上锁的**mutex**解锁
        - 该函数解除阻塞,会对互斥锁加锁
    - 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); // 阻塞一段时间就解除阻塞了

    - 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);

生产者消费者模型


```
//
// Created by bruce on 18-5-20.
//

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>
//节点结构
typedef struct node
{
    int data;
    struct node* next;
}Node;

//永远指向链表头部指针
Node *head = NULL;

//XIANCHENG同步,互斥锁
pthread_mutex_t mutex;
//线程阻塞-条件变量类型的变量
pthread_cond_t cond;

void* produce(void *arg)
{
    while(1)
    {
        Node *pnew = (Node*)malloc(sizeof(Node));

        //节点初始化
        pnew->data = rand()%1000;

        //使用互斥锁保护共享数据
```

```
        pthread_mutex_lock(&mutex);
        //指针域
        pnew->next = head;
        head = pnew;
        printf("======prodece:%lu,%d\n",pthread_self(),pnew->data);
        pthread_mutex_unlock(&mutex);

        //tongzhi 阻塞的消费者进程,解除阻塞
        pthread_cond_signal(&cond);

        sleep(rand()%3);
    }
    return NULL;
}

void* customer(void *arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        //判断链表是否为空
        if(head == NULL)
        {
            //阻塞函数
            pthread_cond_wait(&cond,&mutex);//注意这里有个互斥锁的加进去
            //解除阻塞之后,对互斥锁做枷锁操作

        }
        //部位空
        Node *pdel = head;
        head = head->next;
        printf("=====customer:%ld,%d\n",pthread_self(),pdel->data);
        free(pdel);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
int main()
{
    pthread_t p1,p2;
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&cond,NULL);
    //创建生产者线程
    pthread_create(&p1,NULL,produce,NULL);
    pthread_create(&p2,NULL,customer,NULL);


    //阻塞回收子线程
    pthread_join(p1,NULL);
    pthread_join(p2,NULL);

    pthread_mutex_destroy(&mutex);

    pthread_cond_destroy(&cond);
```

```
}
```

信号量

1. 头文件 - semaphore.h mutex = 1
2. 信号量类型 lock() mutex = 0
    - sem_t sem;

   unlock() mutex = 1
    - 加强版的互斥锁

   mutex实现的同步都是串行的
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 - 5
    - 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

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

#include <pthread.h>
```

```c
#include <semaphore.h>


sem_t produce_sem;
sem_t custom_sem;

typedef struct node
{
    int data;
    struct node* next;
}Node;

Node* head = NULL;

void* producer(void* arg)
{
    while(1)
    {
        sem_wait(&produce_sem);     // porduce_sem --  == 0,  阻塞
        Node * node = (Node*)malloc(sizeof(Node));
        node->data = rand() % 1000;
        node->next = head;
        head = node;
        printf("+++++ 生产者：%lu, %d\n", pthread_self(), node->data);
//        print();
        sem_post(&custom_sem);  // custom_sem++

        sleep(rand()%5);
    }

    return NULL;
}

void* customer(void* arg)
{
    while(1)
    {
        sem_wait(&custom_sem);
        Node* del = head;
        head = head->next;
        printf("----- 消费者：%lu, %d\n", pthread_self(), del->data);
        free(del);
        sem_post(&produce_sem);

        sleep(rand()%5);
    }
    return NULL;
}

int main(int argc, const char* argv[])
{
    pthread_t thid[2];
```

```
    // 初始化信号量
    sem_init(&produce_sem, 0, 4);    // 初始化生产者线程信号量
    sem_init(&custom_sem, 0, 0);     // 初始化消费者线程信号量

    pthread_create(&thid[0], NULL, producer, NULL);
    pthread_create(&thid[1], NULL, customer, NULL);

    for(int i=0; i<2; ++i)
    {
        pthread_join(thid[i], NULL);
    }

    sem_destroy(&produce_sem);
    sem_destroy(&custom_sem);

    return 0;
}
```

哲学家就餐问题