(1条消息)多线程中的锁机制 - 是蛋筒啊的博客 - CSDN博客

由于多线程之间是并发执行的，而系统调度又是随机的，因此在写多线程程序时会出现很多问题，这时就免不了要用到各种锁机制来保证线程安全且按我们的意愿正确执行。

互斥锁

1.定义一个互斥量

```
pthread_mutex_t mutex;
```

- 1

2.初始化互斥量

- 静态分配

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZED
```

- 1

- 动态分配

```
int pthread_mutex_init  (  pthread_mutex_t *restrict  mutex,
          const pthread_mutexattr_t *restrict attr);
```

- 1
- 2

restrict：C中的一个关键字（只能用在函数形参处来修饰一个指针），和volatile的作用相反。它将一个地址放在寄存器中，每次都要从寄存器中取值，目的是做优化。

3.上锁

```
pthread_mutex_lock(&mutex);
```

- 1

4.解锁

```
pthread_mutex_unlock(&mutex);
```

- 1

5.销毁

```
pthread_mutex_destroy(&mutex);
```

- 1

使用互斥量解决售票系统的问题：

```
#include <iostream>
#include <unistd.h>
#include <pthread.h>
using namespace std;
int piaopiao=10;
pthread_mutex_t mutex;

void *ticket(void* arg)
{
  char* str=static_cast<char*>(arg);
  while(1)
  {
    pthread_mutex_lock(&mutex);
    if(piaopiao<=0)
    {
      pthread_mutex_unlock(&mutex);
      break;
    }
    piaopiao--;
    cout<<str<<"sale:"<<piaopiao+1<<endl;
    pthread_mutex_unlock(&mutex);
    usleep(10000);
  }
}

int main()
{
  pthread_t t1,t2,t3,t4;
  pthread_mutex_init(&mutex,NULL);
  pthread_create(&t1,NULL,ticket,(void*)"pthread t1");
  pthread_create(&t2,NULL,ticket,(void*)"pthread t2");
  pthread_create(&t3,NULL,ticket,(void*)"pthread t3");
  pthread_create(&t4,NULL,ticket,(void*)"pthread t4");

  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
  pthread_join(t3,NULL);
  pthread_join(t4,NULL);
  pthread_mutex_destroy(&mutex);
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12

- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41

运行结果：

```
pthread t1sale:10
pthread t2sale:9
pthread t3sale:8
pthread t4sale:7
pthread t1sale:6
pthread t2sale:5
pthread t4sale:4
pthread t3sale:3
pthread t1sale:2
pthread t2sale:1
```

某些时候，线程为了访问临界资源而为其加上锁，但在访问过程中被外界取消，若线程处于响应取消状态，且采用异步方式响应，或者在打开互斥锁之前的运行路径上存在取消点，则该临界资源就永远处于锁定状态得不到释放，外界取消操作难以预见，所以需要一个机制来释放该锁。

//清理函数（用于释放资源）

```
void pthread_cleanup_push(void ,
                                  void *arg);
```

```
void pthread_cleanup_pop(int execute);
```

- 1
- 2
- 3
- 4

回调函数执行时机：

- pthread_exit()
- pthread_cancel()
- pthread_cleanup_pop()的参数不为0时，且执行到这行代码时会触发

这两个函数是以宏的方式实现的do{ }while(0) 的形式，因此这两个函数必须成对出现

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
pthread_mutex_t mutex;
void callback(void* arg)
{
    printf("callback\n");
    sleep(1);
    pthread_mutex_unlock(&mutex);
}
void* even(void* arg)
{
    int i=0;
    for(i=0;;i+=2)
    {
        pthread_cleanup_push(callback,NULL);
        pthread_mutex_lock(&mutex);
        printf("%d \n",i);
        pthread_mutex_unlock(&mutex);
        pthread_cleanup_pop(0);
    }
}
void* odd(void* arg)
{
    int i=0;
    for(i=1;;i+=2)
    {
        pthread_mutex_lock(&mutex);
        printf("%d \n",i);
        pthread_mutex_unlock(&mutex);
    }
}
int main()
{
    pthread_t t1,t2;
    pthread_mutex_init(&mutex,NULL);
    pthread_create(&t1,NULL,even,NULL);
    pthread_create(&t2,NULL,odd,NULL);

    sleep(2);
```

```
    pthread_cancel(t1);
    pthread_mutex_unlock(&mutex);

    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    pthread_mutex_destroy(&mutex);
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49

回旋锁：

一般用于实时性较高的场合

```
pthread_spin_t spin;
pthread_spin_init();
pthread_spin_lock();
pthread_spin_unlock();
pthread_spin_destroy();
```

- 1
- 2
- 3
- 4
- 5

实际工作中用的比较少

读写锁：

读读共享，读写互斥，读优先级高
应用于大量读进程，少量写进程（读者写者模型）

```
pthread_rwlock_t rwlock;
pthread_rwlock_init(&rwlock,NULL);
pthread_rwlock_rdlock(&rwlock);/pthread_rwlock_wrlock(&rwlock);
pthread_rwlock_unlock(rwlock);
pthread_rwlock_destroy(rwlock);
```

- 1
- 2
- 3
- 4
- 5

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
int count=0;
pthread_rwlock_t rwlock;
void* route_read(void *arg)
{
    int i=*(int *)arg;
    free(arg);
    while(1)
    {
        pthread_rwlock_rdlock(&rwlock);
        printf("%d read %d\n",i,count);
        pthread_rwlock_unlock(&rwlock);
        usleep(1000);
    }
}
void* route_write(void *arg)
{
    int i=*(int *)arg;
    free(arg);
```

```
    while(1)
    {
        pthread_rwlock_wrlock(&rwlock);
        printf("%d write %d\n",i,++count);
        pthread_rwlock_unlock(&rwlock);
        usleep(1000);
    }
}

int main()
{
    pthread_t id[8];
    int i=0;
    pthread_rwlock_init(&rwlock,NULL);
    for(i=0;i<5;i++)
    {
        int *p=(int *)malloc(sizeof(int));
        *p=i;
        pthread_create(&id[i],NULL,route_read,(void *)p);
    }
    for(i=5;i<8;i++)
    {
        int *p=(int *)malloc(sizeof(int));
        *p=i;
        pthread_create(&id[i],NULL,route_write,(void *)p);
    }

    for(i=0;i<8;i++)
    {
        pthread_join(id[i],NULL);
    }
    pthread_rwlock_destroy(&rwlock);
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56

运行结果：

```
1 read 7353
5 write 7354
2 read 7354
7 write 7355
6 write 7356
3 read 7356
4 read 7356
0 read 7356
1 read 7356
5 write 7357
2 read 7357
0 read 7357
```

乐观锁/悲观锁

悲观锁

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度(悲观)，因此，在整个数据

处理过程中，将数据处于锁定状态。 悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）

乐观锁

乐观锁（ Optimistic Locking ）相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。

相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。