基于 Pthread 的同步机制研究

周宝航 2120190442

一、问题简介

Pthread 是线程的 POSIX 标准。该标准定义了创建和操纵线程的一整套 API，编写程序时需要人工编写控制线程管理和协调与分解并行任务并管理任务调度的逻辑。但其在系统级代码开发中广泛使用，也用于某些类型的应用程序。Pthread 提供了一整套完善的线程控制接口。但是在并发编程中，我们的程序对于资源的访问控制涉及到同步机制的实现。比如：两个线程都需要对某一个数据结构对象（如：链表）进行操作，可是如果不加以限制的话，并发操作对象的结果会与串行程序逻辑产生偏差。因此，我们需要设计一套同步机制来保证程序并发运行的同时运行结果是正确的。本文主要讨论读写锁机制的实现以及使用 Pthread 其他同步机制来实现读写锁算法，将我们实现的方法与标准 API 的实现进行结果对比。以此来验证我们的同步机制的正确性与有效性。

二、算法描述与实现

首先我们考虑读写锁算法的实现。读写锁即共享-独占（shared-exclusive）锁，其实际是一种特殊的自旋锁，它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。

读写锁和互斥量（互斥锁）很类似，是另一种线程同步机制，可以用来同步同一进程中的各个线程。和互斥量不同的是，互斥量会把试图进入已保护的临界区的线程都阻塞；然而读写锁会视当前进入临界区的线程和请求进入临界区的线程的属性（读/写）来判断是否允许线程进入。

总的来说，读写锁有以下规则：

- 临界区加上读锁时，新的线程可以给临界区继续加读锁，但不能加写锁；
- 临界区加上写锁时，新的线程不可以给临界区加读锁或者写锁；
- 同一时刻可以有多个读锁加给临界区；
- 同一时刻只可以有一个写锁加给临界区；

因此从读写锁非常适合读数据的频率远大于写数据的频率从的应用中。这样可以在任何时刻运行多个读线程并发的执行，给程序带来了更高的并发度。

读写锁是演化于操作系统的读者写者问题，这是从系统中资源调配问题抽象而来。其内容是：写者要等到没有读者时才能去写文件、所有读者要等待写者完成写文件后才能去读文件。这就是读写锁要解决的问题。因此研究读写锁问题就是分析读者写者问题。

读者写者问题中根据具体的需求不同，制定出读者优先和写者优先两种策略。考虑到 Pthread 原生的读写锁是遵循读者优先的策略，本次实验实现的读写锁也都基于读者优先策略。策略如下：

- 读者进程执行：
  - 无其他读者写者，直接执行
  - 有写者等，但其他读者在读，直接读
  - 有写者写，等待
- 写者进程执行：
  - 无其他读写者，直接执行
  - 有其他读写者，等待

伪代码如下：

读者代码

```
//R: 信号量，初值为 1 ，用于同步读者之间
//mutex: 信号量，初值为 1，用于互斥读者与写者，或者写者与写者
//rc: 用于统计当前有多少读者进程
void reader()
{
    while (TRUE)
    {
        P(R);
            rc = rc + 1;
            if (rc == 1)
                P(mutex);
        V(R);
            读操作
        P(R);
            rc = rc - 1;
            if (rc == 0)
                V(mutex);
        V(R);
        其他操作
    }
}
```

写者代码

```
void writer()
{
    while (TRUE)
    {
        P(mutex);
           写操作
        V(mutex);
    }
}
```

读者需要对多读者访问同一临界区进行控制，并且需要控制与写者之间的访

问冲突问题。写者则进行简单的临界区加解锁即可。基于此伪代码，可通过 Pthread 其他同步机制实现读写锁算法。

除此之外，我们还考虑使用 Pthread 的其他同步机制也来实现读写锁。因为考虑到不同的同步机制也会影响控制线程访问资源的效率，我们采用不同的机制来实现读写锁这种机制也是对同步机制的研究。下面介绍几种常用的同步机制方式：

信号量：

信号量机制用于保证两个或多个共享资源被线程协调地同步使用，信号量的值对应当前可用资源的数量。

信号量机制通过信号量的值控制可用资源的数量。线程访问共享资源前，需要申请获取一个信号量，如果信号量为 0，说明当前无可用的资源，线程无法获取信号量，则该线程会等待其他资源释放信号量（信号量加 1）。如果信号量不为 0，说明当前有可用的资源，此时线程占用一个资源，对应信号量减 1。

互斥量：

Mutex（互斥量）是“mutual exclusion”的缩写，互斥量最主要的用途是在多线程中对共享数据同进行写操作时同步线程并保护数据。

互斥量在保护共享数据资源时可以把它想象成一把锁，在 Pthreads 库中互斥量最基本的设计思想是在任一时间只有一个线程可以锁定（或拥有）互斥量，因此，即使许多线程尝试去锁定一个互斥量时成功的只会有一个，只有在拥有互斥量的线程开锁后其它的线程才能锁定，线程必须排队访问被保护的数据。

本次实验需要分析对整个链表加互斥量与加读写锁的性能比较。其主要区别是对于链表读的操作：读写锁可以让线程并行执行读操作，但互斥量让线程必须串行逐步执行读操作。

条件变量：

条件变量是线程使用的一种同步机制。条件变量给多个线程提供了会合的场所。条件变量和互斥量一起用的时候，允许线程以无竞争的方式等待特定的条件发生。条件是受互斥量保护的。线程在改变条件状态时必须首先对互斥量加锁。

条件变量机制弥补了互斥机制的缺陷，允许一个线程向另一个线程发送信号（这意味着共享资源某种条件满足时，可以通过某个线程发信号的方式通知等待的线程），允许阻塞等待线程（当线程等待共享资源某个条件时，可让该线程阻塞，等待其他线程发送信号通知）。

条件变量机制在处理等待共享资源满足某个条件问题时，具有非常高的效率，且空间消耗相比互斥机制也有优势。

三、代码实现

实验代码主要包括：各种同步机制实现的读写锁算法以及链表数据结构。本部分主要介绍各算法的实现程序。

1. 互斥量的读写锁

```
class RWLockMutex
{
public:
    RWLockMutex();
    void wlock();   //写锁
    void uwlock();  //解除写锁
    void rlock();   //读锁
    void urlock();  //解除读锁

private:
    int mutexReaders = 0;
    //基于互斥锁的读写锁
    //定义和初始化互斥锁
    pthread_mutex_t r_mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_t w_mutex = PTHREAD_MUTEX_INITIALIZER;
};

void RWLockMutex::wlock()
{
    pthread_mutex_lock(&w_mutex);
}

void RWLockMutex::uwlock()
{
    pthread_mutex_unlock(&w_mutex);
}

void RWLockMutex::rlock()
{
    pthread_mutex_lock(&r_mutex);
    if (mutexReaders == 0)
        pthread_mutex_lock(&w_mutex);
    mutexReaders++;
```

```
    pthread_mutex_unlock(&r_mutex);
}

void RWLockMutex::urlock()
{
    pthread_mutex_lock(&r_mutex);
    mutexReaders--;
    if (mutexReaders == 0)
        pthread_mutex_unlock(&w_mutex);
    pthread_mutex_unlock(&r_mutex);
}
```

这里使用 2 个互斥锁+1 个整型变量来实现互斥量的读写锁。其中写锁通过简单写的互斥量设置其访问临界区的方式；而读锁通过读的互斥量实现读者访问的控制，通过写的互斥量实现与写者的关系。

2. 信号量的读写锁

```
class RWlockSem
{
public:
    RWlockSem();
    void wlock();
    void uwlock();
    void rlock();
    void urlock();

private:
    //基于信号量的读写锁
    sem_t r_sem;     //定义信号量
    sem_t w_sem;     //定义信号量
    int semReaders = 0;

};

RWlockSem::RWlockSem()
{
    sem_init(&r_sem, 0, 1);     //初始化信号量
    sem_init(&w_sem, 0, 1);     //初始化信号量
}

void RWlockSem::wlock()
{
    sem_wait(&w_sem);
}
```

```
28.
29. void RWlockSem::uwlock()
30. {
31.     sem_post(&w_sem);
32. }
33.
34. void RWlockSem::rlock()
35. {
36.     sem_wait(&r_sem);
37.     if (semReaders == 0)
38.         sem_wait(&w_sem);
39.     semReaders++;
40.     sem_post(&r_sem);
41. }
42.
43. void RWlockSem::urlock()
44. {
45.     sem_wait(&r_sem);
46.     semReaders--;
47.     if (semReaders == 0)
48.         sem_post(&w_sem);
49.     sem_post(&r_sem);
50. }
```

这里使用 2 个信号量+1 个整型变量来实现。令信号量的初始数值为 1，那么信号量的作用就和互斥量等价了。简单来说，信号量的实现与互斥量相似，只是信号量的初始化与互斥量不同，需设置信号量数。

3. 条件变量的读写锁

```
1.  class RWlockCond
2.  {
3.  public:
4.      RWlockCond();
5.      void wlock();
6.      void uwlock();
7.      void rlock();
8.      void urlock();
9.
10. private:
11.     //基于条件变量的读写锁
12.      pthread_mutex_t   rwlock_cond_mutex = PTHREAD_MUTEX_INITIALIZER;       //
    定义和初始化互斥锁
13.      pthread_cond_t   rwlock_cond = PTHREAD_COND_INITIALIZER;        //定义和初
    始化条件变量
```

```
14.     int condReaders = 0;      //记录读者的个数
15.     int condWriters = 0;      //记录写者的个数
16. };
17.
18. void RWlockCond::wlock()
19. {
20.     pthread_mutex_lock(&rwlock_cond_mutex);      //加锁
21.     while (condWriters != 0 || condReaders > 0)
22.     {
23.         pthread_cond_wait(&rwlock_cond, &rwlock_cond_mutex);       //等待条件变量的成立
24.     }
25.     condWriters = 1;
26.
27.     pthread_mutex_unlock(&rwlock_cond_mutex);
28. }
29.
30. void RWlockCond::uwlock()
31. {
32.     pthread_mutex_lock(&rwlock_cond_mutex);
33.     condWriters = 0;
34.     pthread_cond_broadcast(&rwlock_cond);       //唤醒其他因条件变量而产生的阻塞
35.     pthread_mutex_unlock(&rwlock_cond_mutex);     //解锁
36. }
37.
38. void RWlockCond::rlock()
39. {
40.     pthread_mutex_lock(&rwlock_cond_mutex);
41.     while (condWriters != 0)
42.     {
43.         pthread_cond_wait(&rwlock_cond, &rwlock_cond_mutex);       //等待条件变量的成立
44.     }
45.     condReaders++;
46.     pthread_mutex_unlock(&rwlock_cond_mutex);
47. }
48.
49. void RWlockCond::urlock()
50. {
51.     pthread_mutex_lock(&rwlock_cond_mutex);
52.     condReaders--;
53.     if (condReaders == 0)
```

```
54.         pthread_cond_broadcast(&rwlock_cond);        //唤醒其他因条件变量而产生的阻塞
55.     pthread_mutex_unlock(&rwlock_cond_mutex);    //解锁
56. }
```

这里用条件变量+互斥锁来实现。条件变量必须和互斥锁一起使用，等待、释放的时候都需要加锁。条件变量需要在读者写者程序中都进行控制，因为其等待和唤醒程序是相互的，读者完成时需唤醒所有其他线程竞争临界区，写者亦然。因为条件变量的使用比互斥量和信号量麻烦一些，但其的单线程唤醒、广播唤醒等操作可更适用于其他情形的问题处理。

4. 测试算法正确性

为了测试读写锁程序的正确性，本次实验主要通过测试读写锁的最终结果链表是否与串行执行的链表各元素相同。在链表初始化和执行增删操作时，两组链表都通过相同的数据进行完全相同的操作。实验默认读锁共同读的正确性，主要测试写锁的正确性。

四、实验结果与分析

为分析读写锁较于互斥量加锁的效果，实验对比了串行程序、互斥量加锁程序、原生读写锁程序、互斥量的读写锁程序、信号量的读写锁程序和条件变量的读写锁程序的性能。并且分别比较了不同线程数下，加锁程序的并行效果。为了进一步研究读写锁的效用，实验还设置了两组不同比例操作数的实验，分别是 Read:Insert:Delet(0.8:0.1:0.1)和 Read:Insert:Delet(99.9%:0.05%:0.05%)。以此来分析读写锁程序对读操作的并行优化。

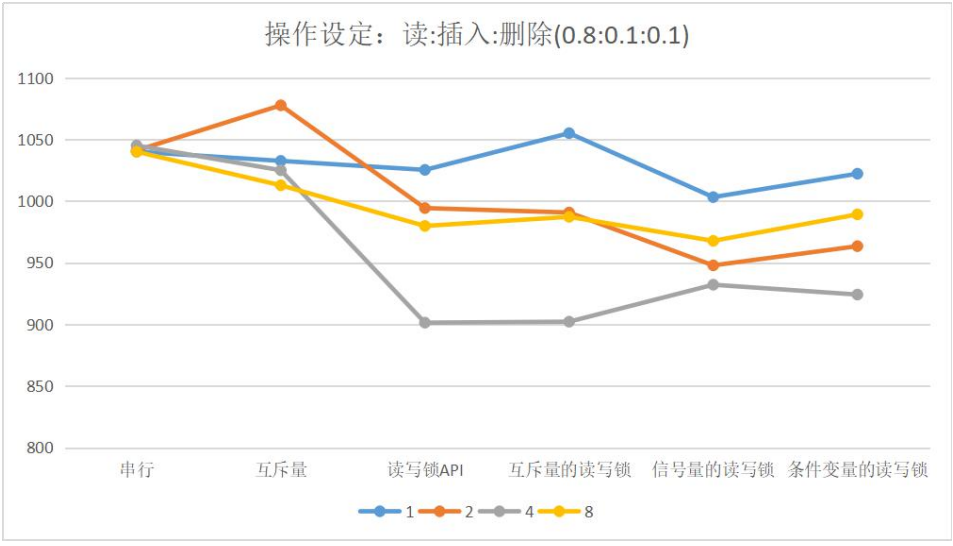

图 1 总操作数 100000；操作比例 Read:Insert:Delet(0.8:0.1:0.1)

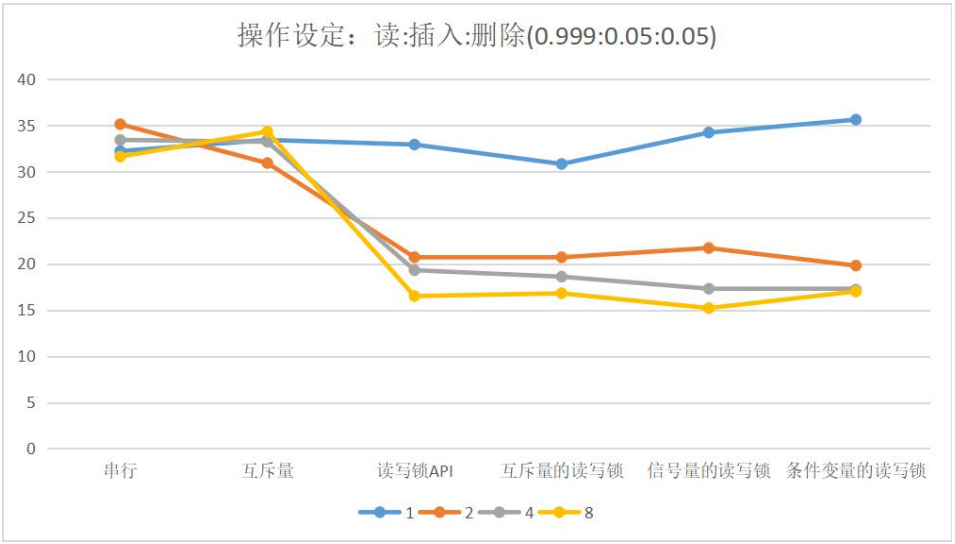

图 2 总操作数 100000；操作比例 Read:Insert:Delet(99.9%:0.05%:0.05%)

从上面的实验结果可以做出以下分析。

对比不同的操作数比例。首先读写操作本身消耗的计算资源就不同，写操作耗费的资源远高于读操作，并且这也与实现的算法相关。因此读操作变多时，整体运算时间会下降。由于读写锁时会并行读操作，因为当读的比例更高时，读写锁程序能提高更多的性能。即读比例高的程序中更多的执行部分可被并行化。

对比不同的线程数量。当线程数为 1 时，无论加什么锁都是只有一个线程执行程序，因此运行时间与线程差不多。当线程数增加时，读写锁的运行效率明显提高，互斥量程序则依然与串行程序相似。随着线程数的增加，并行效果会有所增加，但当到达系统的核心数时，并行效果趋于稳定。

对比不同的实现程序。互斥量加锁的效果与串行程序差不多，因为互斥量会让线程顺序执行临界区代码，其相当于就是串行执行程序，因此运行时间与串行程序相似。对于不同的读写锁实现，其并行效果相差不大，因为其底层的实现都是调用操作系统的原子操作，并且都是基于读者优先的策略。但其都会优于串行和互斥量的效果。因为程序中的读操作可以被并行化，这样可以减少读操作部分的运行时间。

总而言之，读写锁更加适合读操作频繁的应用，因为可以通过并行化更多的读操作使程序性能提高。

六、总结

本次实验研究分析了基于 Pthread 的同步机制。通过研究线程同步的相关的控制函数，分析读写锁的特点与其使用，并且基于 Pthread 中的互斥量、信号量、条件变量实现了三种读写锁。之后比较了不同读写操作比例下的各类锁的性能，

发现当读操作所占比例变大时，对整个链表加读写锁比加互斥锁的性能变高。因此读写锁非常适合读操作更加频繁的应用，这能提高程序的整体执行效率。

通过这次实验，我学习了 Pthread 的同步机制，对各类型锁有了一定了解。并且掌握了读写锁的相关使用，联系操作系统中的读者写者问题，更加深刻认识了读写锁的效用与性能。