**实验二 线程与同步**

操作系统实验报告

李波 22920202204570

# 一、实验内容

本次实验的目的在于将nachos中的锁机制和条件变量的实现补充完整，并利用这些同步机制实现几个基础工具类。实验内容分三部分：实现锁机制和条件变量，并利用这些同步机制将实验一中所实现双向有序链表类修改成线程安全的；实现一个线程安全的表结构；实现一个大小受限的缓冲区。

# 二、实验设计

## （一）用Thread::Sleep实现锁机制和条件变量

### 1.问题描述

这部分实现主要参考Semaphore中Semaphore::P()和Semaphore::V()的实现，在必要的时候应关中断，关中断的方法可以参考Semaphore::P()，阻塞进程应该挂到相应的阻塞队列中，不同的等待事件应对应不同的阻塞队列，因此Lock 类和 Condition 类中应包含相应的队列首指针。

### 2.具体实现

（1）用Thread::Sleep实现锁机制

① Lock类定义与初始化

·思想描述：锁机制要满足对于互斥的要求。首先可以很容易的通过一个表示状态的变量mutex的取值不同进而表示锁的开关状态（分别对应临界区内无、有线程）来实现，在这里使用0表示开状态，1表示关状态。其次，在获得锁（Lock::Acquire）和释放锁（Lock::Release）的函数中的始末开关中断，而不是令用户来控制开关中断，只要保证这两个函数的中间过程的正确性就可以保证一个线程不会无限的执行下去（会被中断）。另外，可以通过一个等待线程队列queue和Thread::Sleep方法来实现当一个线程无法进入队列就将其放入等待线程的队列，然后令其休眠将执行权交给别的线程，由于队列先进先出的特性，进程会轮流按一定顺序进行执行，而不会出现线程被无限延迟的情况。

根据文档中3.2.5节的提示，应该保证以下两个要求：已获得锁的线程不可再次要求获得锁；未获得锁的线程不能要求释放锁。因此设置一个变量heldThread用来保存锁的拥有者。

在Lock类的构造函数中需要根据传入参数设置name属性，将mutex置为0（打开），给queue分配内存，将heldThread置为NULL（没有线程拥有锁）；在析构函数中则需释放队列的内存，防止内存泄漏。

·代码实现：

文本

描述已自动生成

文本

描述已自动生成

② Lock::Acquire()函数实现

·思想描述：首先需要在Lock::Acquire函数的始末打开和关闭系统中断，来保证对mutex判断和操作的一致性。当mutex不为0时，将当前线程加入等待队列，令其休眠，以此来保证其他线程可以切换到执行态且当前线程可以在稍后被正确唤醒。当mutex为0时则将heldThread置为当前线程，并将mutex置为1。在Acquire函数的最开始处，需要通过调用ASSERT函数来确保锁被正确地使用，不能重复要求获得锁。

·代码实现：

文本

描述已自动生成

③ Lock::Release()函数实现

·思想描述：和Acquire函数相似，在Lock::Release函数的始末也需要通过开关中断来保证中间过程的正确性。在释放锁的过程中需要通过从等待队列取出一个当前正在等待的线程，若当前队列中存在线程则将其取出并唤醒，否则不需要唤醒线程。然后需要将heldThread置为NULL，mutex置为0，表示没有线程占有锁。此外，还需要在Release函数的最开始处使用ASSERT函数来保证锁被正确地使用，不能释放非本线程拥有的锁。

·代码实现：

文本

描述已自动生成

④ Lock::isHeldByCurrentThread()函数实现

·思想描述：获得和释放锁的过程中保证了对heldThread的正确修改，通过与系统提供的当前线程指针currentThread进行比较，返回拥有锁的线程是否为当前线程的判断结果。

·代码实现：

文本

描述已自动生成

（2）用Thread::Sleep实现条件变量

① Condition类定义与初始化

·思想描述：根据定义，条件变量需在条件不成立时释放锁，并令当前线程休眠。为了保证线程在稍后被正确唤醒，需要有一个队列queue用来记录当前在等待唤醒的线程，防止其无限休眠。

在实现构造函数时需要根据传入参数初始化name，并且将heldLock置为NULL，并给queue分配内存；在析构函数Condition::~Condition中则需释放queue的内存空间。

·代码实现：

文本

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

② Condition::Wait()函数实现

·思想描述：首先调用ASSERT函数来保证条件变量被正确使用，然后再决定是否调用Condition::Wait。需要先释放锁，再将当前线程加入等待队列，然后令进程休眠。最后，当进程被唤醒时需要再重新获得锁。

·代码实现：

文本

描述已自动生成

③ Condition::Signal函数实现

·思想描述：首先调用ASSERT函数来保证条件变量被正确使用，然后在Condition::Signal中需要从等待队列中取出一个休眠的线程，若队列中存在正在休眠的线程，则唤醒其中一个。

·代码实现：

文本

描述已自动生成

④ Condition::Broadcast函数实现

·思想描述：和Signal函数类似，不同是当等待队列中存在线程则将所有线程唤醒。

·代码实现：

文本

描述已自动生成

## （二）用Semaphore实现锁机制和条件变量

### 1.问题描述

不需要自行考虑关中断和阻塞队列维护等问题。在用 Semaphore 实现条件变量时并不那么直接！这里，请特别注意 Semaphore 与条件变量的区别：如果在调用 Semaphore::P()前调用 Semaphore::V()，则 V 操作的效果将积累下来；而如果在调用 Condition::Wait()前调用 Condition::Signal()，则 Signal 操作的效果将不积累。

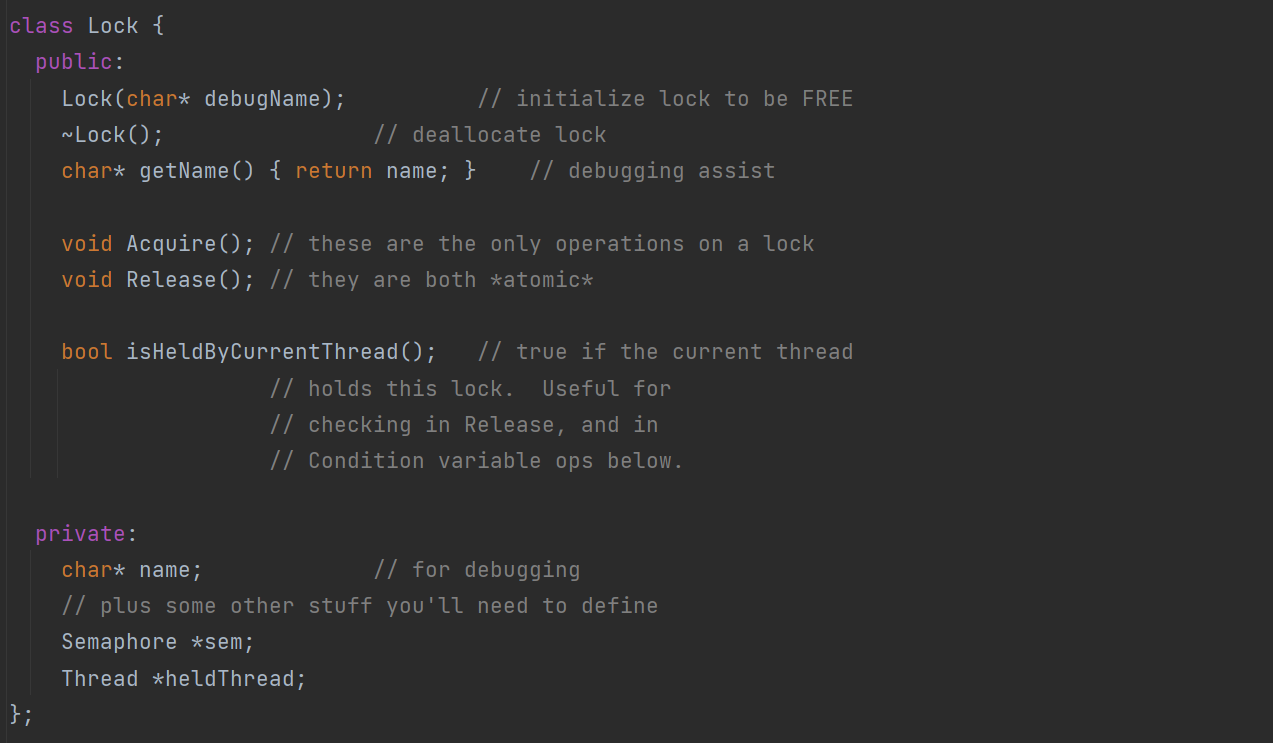
### 2.具体实现

（1）用Semaphore实现锁机制

① Lock类定义与初始化

·思想描述：互斥锁可以用一个值为1的信号量实现，因此类定义中定义一个信号量并在将其值初始化为1。另外，由于互斥锁要求为其加锁的进程和为其解锁的线程必须为同一个线程，因此定义heldThread来记录持有锁的进程，并将其初始化为NULL表示没有线程持有该锁。

·代码实现：



文本

描述已自动生成

② Lock::Acquire()函数实现

·思想描述：首先调用ASSERT函数保证信号量的正确使用，然后对信号量执行P操作，信号量值减1变为0，并将当前线程设置为该锁的拥有者。当另一个线程请求锁时，再次执行P操作，由于此时信号量值为0，线程被阻塞。这样就保证了只有一个线程拥有锁。

·代码实现：

图形用户界面, 文本, 应用程序

描述已自动生成

③ Lock::Release()函数实现

·思想描述：首先调用ASSERT函数保证信号量的正确使用。当锁的所有者释放锁时，对信号量执行V操作，唤醒一个阻塞线程，并将信号量的值加1变回1，锁的拥有者置为NULL，回到初始状态等待下一次再被请求。

·代码实现：

图形用户界面, 文本

描述已自动生成

④ Lock::isHeldByCurrentThread()函数实现

·思想描述：获得和释放锁的过程中保证了对heldThread的正确修改，通过与系统提供的当前线程指针currentThread进行比较，返回拥有锁的线程是否为当前线程的判断结果。

·代码实现：

文本

描述已自动生成

（2）用Semaphore实现条件变量

① Condition类定义与初始化

·思想描述：条件变量可以用一个值为0的信号量sem实现，同时定义waitingNum记录阻塞线程数。此外，还需要定义一个信号量atom保证操作是原子的。

·代码实现：

文本

描述已自动生成

文本

描述已自动生成

② Condition::Wait()函数实现

·思想描述：首先调用ASSERT函数保证信号量的正确使用。然后释放相应的互斥锁并执行P操作等待条件发生变化。一旦其它的某个线程改变了条件变量则唤醒一个阻塞线程并重新上锁。其中，释放锁，修改waitingNum和P操作要放在atom的P和V操作之间保证操作不被中断。

·代码实现：

文本

描述已自动生成

③ Condition::Signal()函数实现

·思想描述：验证当前互斥锁与环境变量相关联之后，如果存在阻塞进程，则执行V操作通知相应的条件变量唤醒线程。

·代码实现：

文本

描述已自动生成

④ Condition:: Broadcast()函数实现

·思想描述：与Condition::Signal()函数类似，只是通过循环唤醒所有被wait函数阻塞在某个条件变量上的线程。

·代码实现：

文本

描述已自动生成

## （三）用锁机制和条件变量修改双向有序链表

### 1.问题描述

实验一里在 nachos 系统中运行了自己编写的链表程序并演示了一些并发错误，其原因是测试程序未考虑互斥。现在根据所实现的锁和条件变量机制重写文件，并确保修改后的多线程并发程序是正确互斥的。再次注意：条件变量应和锁配合使用。在使用过程中应注意，不能出现一个条件变量与两个不同的锁对应的情况。

### 2.具体实现

（1）dllist定义修改

·思想描述：实现线程安全的双向链表需定义一个锁和一个用于判断链表非空的条件变量。

·代码实现：

文本

描述已自动生成

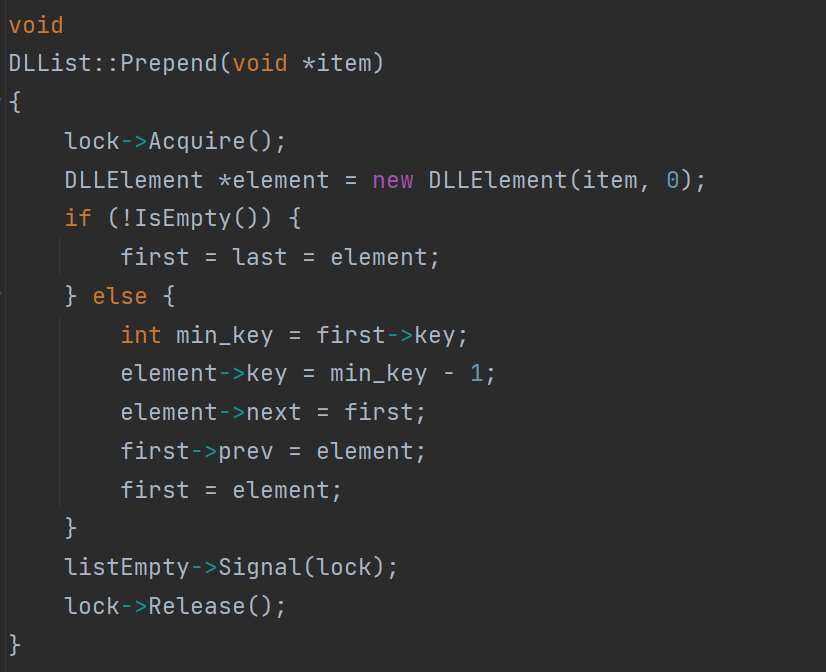
文本

描述已自动生成

（2）dllist.cc中Prepend函数和Append函数以及SortedInsert函数实现

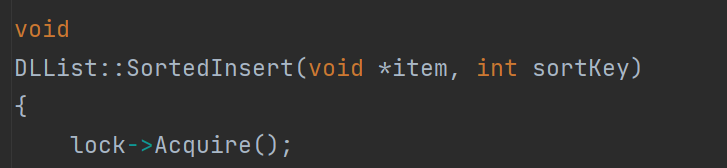
·思想描述：实现线程安全的链表结点添加，一是保证链表互斥访问，为此需要加锁；二是在添加节点后要调用Signal函数唤醒被阻塞的线程。

·代码实现：



文本

描述已自动生成



文本

描述已自动生成

（3）dllist.cc中Remove函数和SortedRemove函数实现

·思想描述：实现线程安全的链表结点删除，需要加锁保证链表互斥访问。此外，若链表为空，则要调用wait函数使线程阻塞。

·代码实现：

文本

中度可信度描述已自动生成

图片包含 文本

描述已自动生成

图形用户界面, 文本

描述已自动生成

文本

描述已自动生成

（4）dllist-driver.cc、threadtest.cc修改

·思想描述：dllist-driver.cc中的RemoveItems()函数不再需要判断链表非空。

·代码实现：

文本

描述已自动生成

### 3.实验结果

（1）用Thread::Sleep实现锁机制和条件变量

文本

描述已自动生成 文本

描述已自动生成

（2）用Semaphore实现锁机制和条件变量

文本

描述已自动生成 文本

描述已自动生成

从实验结果可见，通过修改双向链表，先前设计的并发错误不会发生。

## （四）实现一个线程安全的表结构

### 1.问题描述

头文件Table.h包含了相应的接口声明，实现应该放在Table.cc里，并注意修改 Makefile 文件里的有关部分（具体项目可以参考实验一的说明）Table.h中class Table的构造函数声明有问题，应该修改。此外，应该在其中补充析构函数的声明，并在 Table.cc 中实现。为实现线程安全，需要用到锁机制。

### 2.具体实现

（1）Table类定义与初始化

·思想描述：Table类中主要的函数除构造与析构函数外还有分配表空间函数Alloc()、按索引查询表值的函数Get()、释放指定索引位置空间的函数Release()。在构造函数中，设置tableSide为表容量，实例化一个锁用于保证线程安全，初始化每个位置的标志位为0表示没有数据。

·代码实现：

文本

描述已自动生成

文本

描述已自动生成

（2）Table::Alloc()函数实现

·思想描述：Alloc函数功能是为用户数据分配存储位置并返回存储位置索引，若分配失败则返回-1。实现线程安全的方法是在每次调用Alloc的时候在进入函数后马上加锁使当前进程不会被其他进程打断，整个在table中寻找空闲位置的过程和存储数据的过程中，都有锁的保护。无论成功或失败，在返回值前释放锁避免死锁。

·代码实现：

文本

描述已自动生成

（3）Table::Get函数实现

·思想描述：Get函数功能为获取Table指定索引位置存储的内容。，首先要调用ASSERT函数保证index参数的合理性，然后实现线程安全的方法是加锁保护当前线程不会被其他线程打断，随后找到table指定位置，若有值则返回该值，否则返回NULL。注意锁需要在返回值之前释放否则会造成死锁。

·代码实现：

文本

描述已自动生成

（4）Table::Release ()函数实现

·思想描述：Release函数功能为释放table中指定索引位置存储空间。实现线程安全的方法是每次调用Release时在进入函数后马上加锁使得整个执行过程不会被其他进程打断，随后找到指定位置，删除其中的数据，修改其value值为0表示此空间没有数据。

·代码实现：

文本

描述已自动生成

（5）测试函数实现

·思想描述：首先在定义一个指定大小的table。线程先执行Alloc，存入的内容为随机生成的整数值，根据返回值判断空间是否分配成功，若分配成功则继续按顺序执行Acquire和Release，否则直接结束。

·代码实现：

文本

描述已自动生成

文本

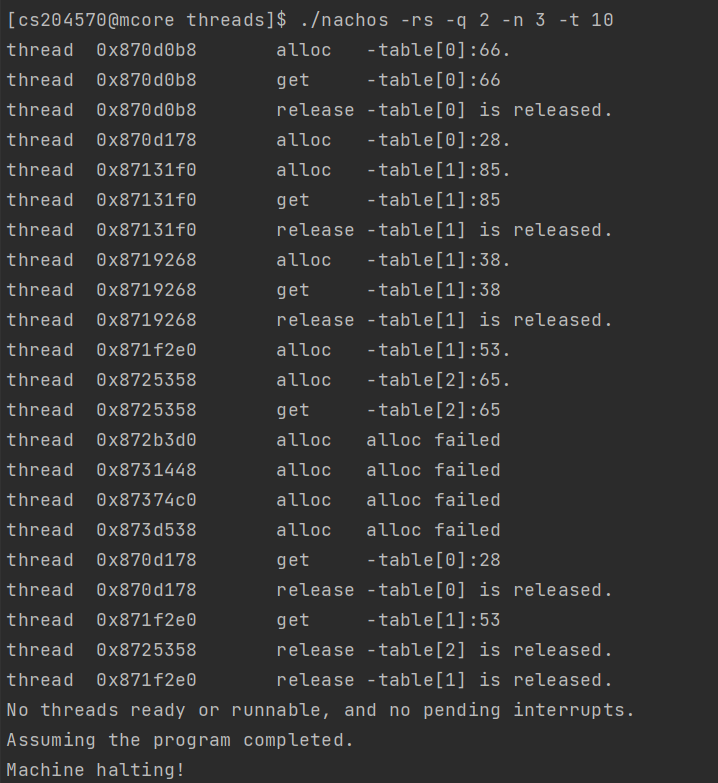
描述已自动生成

### 3.实验结果

**命令格式：**./nachos -rs [seed] -q 2 [-t <threadNum>] [-n <itemNum>]

其中，rs参数表示线程随机切换、q为2表示对Table进行测试，后面两个参数分别表示线程数和表的容量。

（1）用Thread::Sleep实现锁机制和条件变量



·由上图可见：在没有线程遇到Table已满的情况时，能保证线程安全和正常执行。在分配时遇到alloc failed即Table已满的情况时，也能保证线程安全和正常执行。

## （五）实现一个大小受限的缓冲区

### 1.问题描述

头文件BoundedBuffer.h包含了相应的接口声明，实现应该放在 BoundedBuffer.cc 里，并注意修改Makefile文件里的有关部分（具体项目可以参考实验一的说明），请在 BoundedBuffer.h中class BoundedBuffer的声明中补充析构函数的声明，并在 BoundedBuffer.cc 中实现之。这个问题实际上是“生产者－消费者”问题。实现时候可以用 Semaphore 机制，也可以 使用锁机制配合条件变量。如果使用 Semaphore 机制，则需要三个信号量；如果用锁机制配合条件变量，则需要一个锁和两个条件变量。

### 2.具体实现

（1）类定义与初始化

·思想描述：既然是生产者消费者问题，那么就要做好生产（Write）和消费（Read）函数的设计。这里我采用锁机制配合条件变量来实现缓冲区。数据域方面，除了定义了缓冲区指针、缓冲区大小，一个锁和两个条件变量（notEmpty、notFull）外，还需要readFrom、writeTo表示读出和写入的索引位置。构造函数中，可以设置缓冲区大小，初始化锁和条件变量，初始化readout、writein为0，表示缓冲区为空。

·代码实现：

文本

描述已自动生成

文本

描述已自动生成

（2）BoundedBuffer::Read()函数实现

·思想描述：通过条件变量notEmpty等待缓冲区有资源可以使用，通过锁lock来保证read时不会有其他线程访问缓冲区。在完成read后，通过条件变量notFull通知消费者缓冲区有资源可以取走，并且释放互斥锁。

·代码实现：

文本

描述已自动生成

（3）BoundedBuffer::Write()函数实现

·思想描述：通过条件变量notFull等待缓冲区有空闲位置，通过锁lock来保证write时不会有其他线程访问缓冲区。在完成write后，通过条件变量notEmpty通知缓冲区有空闲位置，并且释放互斥锁。

·代码实现：

文本

描述已自动生成

（5）测试函数实现

·思想描述：测试函数主要测试的是read、write两个函数，首先在threadtest函数中开辟一个指定大小的buffer。执行时，生产者每次写入一个数字（数字从0到9循环生成），而消费者每次读出一个数字。

·代码实现：

文本

描述已自动生成

文本

描述已自动生成

### 3.实验结果

**命令格式：**./nachos -rs [seed] -q 3 [-t <threadNum>]

其中，最后一个参数表示线程数。

文本

描述已自动生成

# 三、实验总结

线程同步与互斥是操作系统课程中难度较大的一部分，之前单纯地通过课本学习时会需要很长时间理解，一些难点很难深入理解。通过本次实验，我亲手完成了信号量、互斥锁和条件变量的实现，并且尝试实现了一些经典的线程安全数据结构，让我对线程同步和互斥有了深入的理解，更好地理解了课本上的内容。

在本次实验中，难点是各个机制实际运行过程的思考，运行时总是会出现一些匪夷所思的实验现象，需要找出代码实现中的漏洞。通过不懈的努力和思考，我最终成功解决了这些问题。这样的过程让我对操作系统的实现和设计有了更深入的认识和理解。