

同步机制实习报告

姓名：李聪 学号：1800012826

日期：2020/11/13

目录

内容一：任务完成情况.....	3
任务完成列表（Y/N）	3
具体 Exercise 完成情况	3
Exercise1	3
Exercise2	7
Exercise3	12
Exercise4	15
Challenge	23
内容二：遇到的困难以及收获.....	32
内容三：对课程或 Lab 的意见和建议.....	33
内容四：参考文献.....	33

内容一：任务完成情况

任务完成列表 (Y/N)

	Exercise1	Exercise2	Exercise3	Exercise4	Challenge
第一部分	Y	Y	Y	Y	Y

具体 Exercise 完成情况

Exercise1

在这部分练习中，我调研了 Linux 的同步机制，现将这一机制进行简要的阐述。

Linux 的同步机制种类繁多，比较常见的有：每 CPU 变量、原子操作、内存屏障、自旋锁、顺序锁、信号量等。下面我来简要介绍一下

(一) 每 CPU 变量

每 CPU 变量意为每个 CPU 都有自己的变量，各个 CPU 仅访问自己的每 CPU 变量。一般来说，每 CPU 变量的数据结构是一个长度等于 CPU 个数的数组。这个机制可以用来解决多 CPU 之间可能大声的竞争条件。

(二) 原子操作

原子操作表示把一些操作放到“一步”来进行，不可分割的一些操作。Linux 的实现手段主要有：

1. 操作码前缀为 lock 的汇编指令，表示在执行这条操作时锁住总线，进而保证不会有其他 CPU 读写内存。

2. 多处理器中，Linux 内核提供了 atomic_t 类型，封装了一系列原子操作。比如，atomic_inc(v) 表示原子地给 v 加 1。

(三) 优化和内存屏障

优化编译器为了达到最优的处理方式，会将代码重排，这样可能造成程序访问内存时的顺序不会像我们原来写的那样。Linux 为这种问题提供了一种屏障机制，让屏障之前和之后的指令不会因为重排而跨越这个屏障。

（四）自旋锁

自旋锁是一种常用的同步技术，它锁住一块临界区，进入临界区时需要先获取自旋锁，在离开临界区时需要释放自旋锁。如果已经有其他进程获取了该锁，那么当前想要获取该锁的进程只能在临界区入口自旋（做忙等待），直到获取该锁。

1.最基础的自旋锁的使用方法如下。只要在访问临界区之前上锁，访问临界区之后解锁即可。为了保证上锁和解锁的原子性，两个函数调用中会屏蔽中断。

```
static DEFINE_SPINLOCK(xxx_lock);

unsigned long flags;

spin_lock_irqsave(&xxx_lock, flags);
... critical section here ..
spin_unlock_irqrestore(&xxx_lock, flags);
```

2.读写自旋锁。对于读着-写着问题，我们允许多个读者同时读，但一旦有一个写者开始写，就不允许其他读者/写者进入缓冲区。为此，我们可以设计读写锁来解决这类问题。在现在的 Linux 同步机制中，也会用 RCU 技术来解决这一问题。

```
rwlock_t xxx_lock = __RW_LOCK_UNLOCKED(xxx_lock);

unsigned long flags;

read_lock_irqsave(&xxx_lock, flags);
.. critical section that only reads the info ...
read_unlock_irqrestore(&xxx_lock, flags);

write_lock_irqsave(&xxx_lock, flags);
.. read and write exclusive access to the info ...
write_unlock_irqrestore(&xxx_lock, flags);
```

3.互斥锁 Mutex，这个结构类似于一元信号量，在 Linux 中实现如下：

```
struct mutex {
    atomic_long_t    owner;
    spinlock_t       wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
    struct list_head  wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
    void              *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
};

struct ww_class;
struct ww_acquire_ctx;

struct ww_mutex {
    struct mutex base;
    struct ww_acquire_ctx *ctx;
#ifdef CONFIG_DEBUG_MUTEXES
    struct ww_class *ww_class;
#endif
};
```

Mutex 的核心内容是持有者和一个自旋锁。获取锁的函数内容如下：

```
void __sched mutex_lock(struct mutex *lock)
{
    might_sleep();

    if (!__mutex_trylock_fast(lock))
        __mutex_lock_slowpath(lock);
}
```

（五）顺序锁

顺序锁对某个共享数据读取的时候不加锁，写的时候加锁。为了避免读取的过程中写者修改内容导致的读取错误，这个锁还引入了 `sequence` 变量，每次读者读取前后要对这个变量进行判断，不相同则重新读取；写者写入数据的同时要更新 `sequence` 的值。

顺序锁的结构定义如下。`lock` 用于保证对 `seqcount` 的原子性操作，而 `seqcount` 即为上面提到的封装过的 `sequence` 变量：

```
typedef struct {
    struct seqcount seqcount;
    spinlock_t lock;
} seqlock_t;

typedef struct seqcount {
    unsigned sequence;
} seqcount_t;
```

写入者上锁的操作如下。主要是锁住临界区，再更新 `sequence` 的值：

```
static inline void write_seqlock(seqlock_t *sl)
{
    spin_lock(&sl->lock);
    write_seqcount_begin(&sl->seqcount);
}
```

```
static inline void write_seqcount_begin(seqcount_t *s)
{
    write_seqcount_begin_nested(s, 0);
}

static inline void write_seqcount_begin_nested(seqcount_t *s, int subclass)
{
    raw_write_seqcount_begin(s);
    seqcount_acquire(&s->dep_map, subclass, 0, _RET_IP_);
}

static inline void raw_write_seqcount_begin(seqcount_t *s)
{
    s->sequence++;
    smp_wmb();
}
```

解锁操作内容如下。可以看到，`sequence` 仍然是加 1，这样可以保证 `sequence` 是偶数就没有写者在写，奇数说明有写者正在写。

```

static inline void write_sequnlock(seqlock_t *sl)
{
    write_seqcount_end(&sl->seqcount);
    spin_unlock(&sl->lock);
}

static inline void write_seqcount_end(seqcount_t *s)
{
    seqcount_release(&s->dep_map, 1, _RET_IP_);
    raw_write_seqcount_end(s);
}

static inline void raw_write_seqcount_end(seqcount_t *s)
{
    smp_wmb();
    s->sequence++;
}

```

读操作的实现如下。注意到，repeat 内部的判断意为，当取到的 sequence 值为奇数，那么要不停地回到判断的开始阶段，重新判断，直到写者操作完毕后再返回：

```

static inline unsigned read_seqbegin(const seqlock_t *sl)
{
    return read_seqcount_begin(&sl->seqcount);
}

```

```

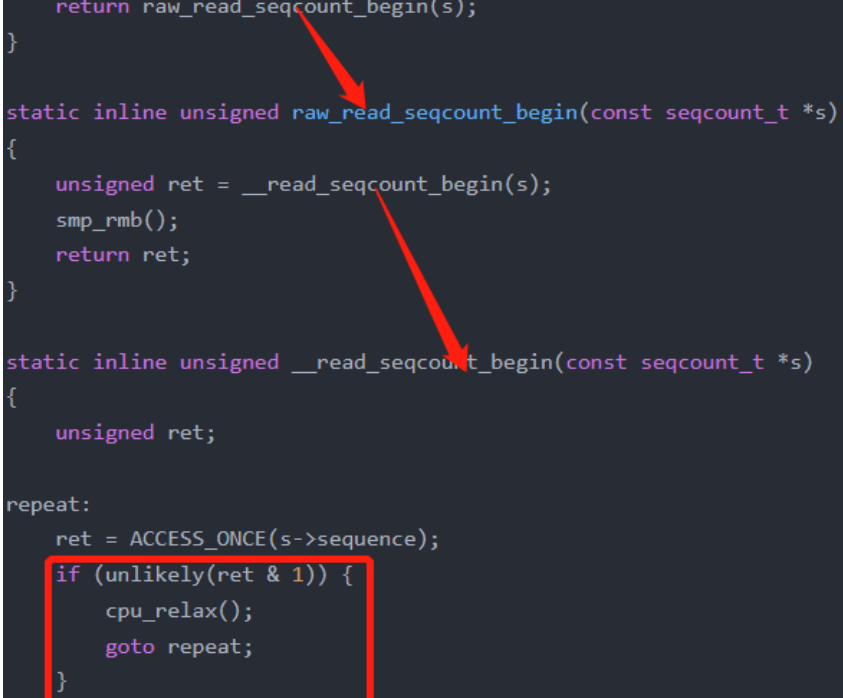
static inline unsigned read_seqcount_begin(const seqcount_t *s)
{
    seqcount_lockdep_reader_access(s);
    return raw_read_seqcount_begin(s);
}

static inline unsigned raw_read_seqcount_begin(const seqcount_t *s)
{
    unsigned ret = __read_seqcount_begin(s);
    smp_rmb();
    return ret;
}

static inline unsigned __read_seqcount_begin(const seqcount_t *s)
{
    unsigned ret;

repeat:
    ret = ACCESS_ONCE(s->sequence);
    if (unlikely(ret & 1)) {
        cpu_relax();
        goto repeat;
    }
    return ret;
}

```



（六）信号量

信号量与自旋锁类似，都用于控制临界区的互斥访问，但是二者的区别在于：自旋锁在获取锁时，不会主动切换，而是会占用 CPU 自旋，类似于 while 死循环。除此之外，信号量存在等待队列，P 操作未成功获取的进程（线程）会挂到这个队列中，主动切换至其他进程（线程）。

（七）小结

实际上，除了这几种常见的同步机制以外，Linux 还有诸如读-拷贝-更新机制（RCU 机制）、禁止本地中断机制、Futex 机制、完成变量机制等来实现同步。除此之外，管道、信号、套接字还可以实现进程间的通信。可以说，Linux 的同步和通信机制真的是种类齐全，应有尽有。

Exercise2

这一部分要求阅读四份源代码，下面我将逐一分析各部分的具体内容。总的来说，这四份源代码包含了 Nachos 中的所有同步机制相关的类和方法。

（一）code/threads/synch.h 和 code/threads/synch.cc

这两份代码文件中包含了用于实现同步的三种数据结构：Semaphore、Lock、Condition。其中的 Lock 和 Condition 只提供了函数接口，内容需要在 Exercise3 中完成。接下来我将对 Semaphore 的机制进行简要分析。

Semaphore 中的成员变量有：信号量名称 name（用于 debug）、信号量值 value（总是大于等于 0）、一个队列，用于储存等待在 P 上的线程信息。

注意到信号量结构中不允许线程直接访问信号量的 value 值，实际上这样的操作是没有意义的。因为这样只能读取当时信号量的值，而不能读取现在这一时刻信号量的值。因为在把信号量的值存到寄存器之后，可能又进行了上下文切换，其他线程可能又进行了 PV 操作修改了 value 值。

```
private:
    char* name;
    int value;
    List *queue;
    .
```

除此之外，信号量中还封装了一些公共方法：构造函数、析构函数、getName、PV 操作，

除了 getName 以外（这个方法就是简单地返回 name 成员变量），这些方法在 synch.cc 中实现。下面我们来逐一分析一下。

```
public:
    Semaphore(char* debugName, int initialValue);
    ~Semaphore();
    char* getName() { return name;}

    void P();
    void V();
```

1.构造函数和析构函数

Semaphore 的构造函数需要提供信号量名称和初始值这两个参数，它们分别用于初始化信号量对应的字段，然后一个新的等待队列也会被创建。

```
Semaphore::Semaphore(char* debugName, int initialValue)
{
    name = debugName;
    value = initialValue;
    queue = new List;
}
```

Semaphore 的析构函数就是把等待队列删除。因为在析构一个信号量时，我们会假设，已经没有任何线程还等待在这个信号量上面。

```
Semaphore::~~Semaphore()
{
    delete queue;
}
```

2.P 操作

Nachos 中假设系统运行在单处理器机器上，因此 PV 中的原子操作都采用屏蔽中断的方式来实现。

在 P 操作中，信号量首先保存原有的中断等级，然后调用中断类的 SetLevel 函数，以关闭中断。

中断关闭后，如果当前信号量值为 0，那么把这个线程加入信号量的等待队列（插入指向这个线程结构的指针），然后调用 Sleep 函数让当前线程进入阻塞状态。回顾先前已经调研过的 Sleep 函数，我们会发现，Sleep 在执行之前会

当信号量值大于 0 以后，P 操作会占据信号量，把信号量 value 减 1，然后利用先前保存的 oldLevel 变量把中断恢复到原来的状态。

```
void
Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {
        queue->Append((void *)currentThread); // sei
        currentThread->Sleep();                // so
    }
    value--;
    (void) interrupt->SetLevel(oldLevel);      // sei
                                              // coi
}
```


3.V 操作

Nachos 的 V 操作也会首先屏蔽中断，以保证原子性，接下来 V 操作会取出等待队列中的第一个线程。如果队首的线程指针不为空，那么 V 操作会把这个线程加入就绪队列中。然后，V 操作把信号量值加 1，再重新恢复中断到原来的状态。

注意，恢复中断并不代表一定要恢复到可相应的状态，因为调用 V 之前可能已经处于屏蔽中断的状态了，所以我们使用 `oldLevel` 来恢复中断。

```
void
Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    if (thread != NULL) // make thread ready, cons
        scheduler->ReadyToRun(thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

下面我们再来看一下 Lock 和 Condition 的内容。

Lock 代表互斥锁类，一个锁可以处于 BUSY 状态或 FREE 状态。每个线程可以调用 Acquire 函数获取锁，也可以调用 Release 函数来释放锁。由于原则上我们应该让获取这把锁的线程来释放锁，所以类中还加入了 `isHeldByCurrentThread` 函数来做这个判断。而且注意到，Lock 的成员中并没有 Value 值，所以我们可能还需要添加一些其他变量。

```
class Lock {
public:
    Lock(char* debugName);           // initialize lock to be FREE
    ~Lock();                         // deallocate lock
    char* getName() { return name; } // debugging assist

    void Acquire(); // these are the only operations on a lock
    void Release(); // they are both *atomic*

    bool isHeldByCurrentThread();    // true if the current thread
                                    // holds this lock. Useful for
                                    // checking in Release, and in
                                    // Condition variable ops below.

private:
    char* name;                     // for debugging
    // plus some other stuff you'll need to define
};
```

Condition 代表条件变量。条件变量内没有 value 值，但是线程可以在这个变量上列队等待。Condition 支持三个操作：Wait 操作会释放这个条件变量，直到被 Signal 通知之前，都会让出 CPU；Signal 会唤醒等待在这个条件变量上的一个线程；而 Broadcast 函数则会唤醒所有等待在这个条件变量上的线程。

除此之外，所有条件变量操作都必须在当前线程已经获取了一把锁的时候来完成。实际上，每个条件变量都应该有一把锁来保护。Nachos 中得出条件变量基于 MESA 管程的语义：当条件变量执行 Signal 或者 Broadcast 时，被唤醒的线程被添加到就绪队列中，重新获取锁。

```

class Condition {
public:
    Condition(char* debugName);           // initialize condition to
                                           // "no one waiting"
    ~Condition();                         // deallocate the condition
    char* getName() { return (name); }

    void Wait(Lock *conditionLock);       // these are the 3 operations on
                                           // condition variables; releasing the
                                           // lock and going to sleep are
                                           // *atomic* in Wait()
    void Signal(Lock *conditionLock);     // conditionLock must be held by
    void Broadcast(Lock *conditionLock); // the currentThread for all of
                                           // these operations

private:
    char* name;
    // plus some other stuff you'll need to define
};

```

(二) code/threads/synchlist.h 和 code/threads/synchlist.cc

这两份代码中主要实现了一个“同步列表”——SynchList 类。这个类可以维持下列约束：

1. 尝试移除 List 中元素的线程，如果发现 List 中没有元素，会一直等待，直到 List 中有了元素。
2. 一个时刻上，只有一个线程能获取 List 数据结构。

这个类中的成员变量包含：一个非同步的列表 list、一个锁 Lock、一个条件变量 listEmpty。

```

private:
    List *list;
    Lock *lock;
    Condition *listEmpty;

```

除此之外，这个类中还包含以下方法，它们在 synchlist.cc 中实现：

```

public:
    SynchList();
    ~SynchList();

    void Append(void *item);

    void *Remove();

```

1. 构造函数和析构函数

SynchList 的构造函数会新建一个空的 List，并生成锁和条件变量。构造函数执行完毕后，可以开始添加元素了。

```

SynchList::SynchList()
{
    list = new List();
    lock = new Lock("list lock");
    listEmpty = new Condition("list empty cond");
}

```

析构函数则是把创建的成员变量全部 delete 掉，释放内存空间。

```

SynchList::~~SynchList()
{
    delete list;
    delete lock;
    delete listEmpty;
}

```

2.Append

向队列中添加一个元素，同时唤醒一个等待读取元素的线程。由于要实现同步操作，因此在使用 Append 添加元素和通知等待在条件变量上的元素之前，要先获取锁；而且执行完两个操作后还要再释放锁。

```

void
SynchList::Append(void *item)
{
    lock->Acquire();
    list->Append(item);
    listEmpty->Signal(lock);
    lock->Release();
}

```

3.Remove

从队列的起始处移除一个元素。如果队列为空，那么线程进行等待，直到队列非空。函数首先试图获取锁，获取之后，当队列为空时，当前线程进入条件变量的就绪队列中等待，直到队列非空。然后便可从队列中取走元素，释放锁之后返回。

```

void *
SynchList::Remove()
{
    void *item;

    lock->Acquire();
    while (list->IsEmpty())
        listEmpty->Wait(lock);
    item = list->Remove();
    ASSERT(item != NULL);
    lock->Release();
    return item;
}

```

4.Mapcar

这个函数对队列中的每个元素执行相同的操作，为了保证同步，在调用 List 对象的 Mpcar 之前，要先获取锁；执行之后，还要释放锁。

```

void
SynchList::Mapcar(VoidFunctionPtr func)
{
    lock->Acquire();
    list->Mapcar(func);
    lock->Release();
}

```

Exercise3

这一部分要求我们实现锁和条件变量机制。

（一）锁机制的实现

为了实现锁机制，需要我们完成的方法有：构造函数、析构函数、Acquire 和 Release 操作、对释放者和拥有者是否相等的判断。

由于锁可以被看作一种二元信号量（因为每个时刻只有一个线程能占据这个共享变量），而且有占有者的概念，所以考虑使用信号量来实现锁机制。为此，对 Lock 结构添加两个私有成员：Owner 和 lockSemaphore:

```
private:
    char* name; // for debugging
    // plus some other stuff you'll need to define

    /* ----- added by Li cong 1800012826 ----- */
    Thread* Owner; // which thread is holding this lock now
    Semaphore* lockSemaphore; // Lock implementation
```

接下来，便可以开始完善整个锁的成员函数了。

构造函数需要对锁的成员变量进行初始化。初始时 Owner 指针应该为空，表示没有线程占有锁；而且信号量的初始值应该为 1，因为每个时刻最多只能有一个线程占据锁。

```
Lock::Lock(char* debugName)
{
    name = debugName;
    Owner = NULL;
    lockSemaphore = new Semaphore("Lock semaphore", 1); // Only allow one thread to access lock at ont time.
}
```

析构函数采取与信号量类似的思想，delete 掉这个锁中的 Semaphore 结构。

```
Lock::~Lock()
{
    delete lockSemaphore;
}
```

由于添加了 Owner 成员，isHeldByCurrentThread 方法只需要把 Owner 指针与全局的 currentThread 指针做比较

```
bool Lock::isHeldByCurrentThread()
{
    return Owner == currentThread;
}
```

Acquire 操作的思想与信号量的 P 操作类似。由于 Owner 的修改和内部锁信号量的获取需要原子性地完成，因此在这个函数的开始和结束也要进行中断的屏蔽与恢复。然后在这中间，进行信号量的获取等待（P 操作）和持有者的更新。

```

void Lock::Acquire()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    DEBUG('s', "Lock \"%s\" is Acquired by Thread \"%s\"\n", name, currentThread->getName());

    lockSemaphore->P(); // try to get semaphore, if failed, currentThread will sleep.
    Owner = currentThread;

    (void) interrupt->SetLevel(oldLevel);
}

```

Release 操作的思想也与信号量的 V 操作类似。与上面的理由相同，我们也需要对 Owner 修改和 lockSemaphore 的 V 操作进行原子化。除此之外，为了便于调试，我还加入了一些调试语句。

```

void Lock::Release()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    DEBUG('s', "Lock \"%s\" is Released by Thread \"%s\"\n", name, currentThread->getName());
    ASSERT(isHeldByCurrentThread()); // By convention, only the thread that acquired the lock may release it.
    Owner = NULL;
    lockSemaphore->V();

    (void) interrupt->SetLevel(oldLevel);
}

```

至此，Lock 的实现完成。（测试在后续问题中进行）

（二）条件变量机制的实现

为了实现条件变量机制，我们需要完成构造函数、析构函数、Wait-Signal-Broadcast 三个操作。

由于等待在这个条件变量上的线程需要排队，因此在实现上述方法前，我们需要添加一个成员变量——等待队列 waitQueue

```

private:
    char* name;
    // plus some other stuff you'll need to define

    /* ----- added by Li cong 1800012826 ----- */
    List* waitQueue;

```

下面我们开始实现条件变量的各个方法。

构造函数和析构函数与 Lock 的思路类似，就是初始化成员变量和释放成员变量，内容如下：

```

Condition::Condition(char* debugName)
{
    name = debugName;
    waitQueue = new List();
}

Condition::~Condition()
{
    delete waitQueue;
}

```

注意到，调用条件变量的方法之前，应该先获取一个锁，这个锁作为 Wait-Signal-

Broadcast 的参数传入方法中。

因此，在调用 Wait 之前，传入参数的锁一定是被占用了的，因此我们需要在正式执行 Wait 之前进行检查，且 Wait 的最开始我们需要屏蔽中断，结束后再恢复中断响应。这期间我们要进行的操作包括：释放锁——把当前线程加入等待队列——挂起当前线程——切换回来以后再重新获取锁。因此，Wait 的内容如下：

```
void Condition::Wait(Lock* conditionLock)
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff); // Ensure *atomic* operation.

    // Step 0: Check whether current thread is holding this lock.
    ASSERT(conditionLock->isHeldByCurrentThread());

    // Step 1: release the lock
    conditionLock->Release();

    // Step 2: append thread to waitQueue and block this thread
    waitQueue->Append(currentThread);
    currentThread->Sleep();

    /* Signal came */

    // Step 3: re-Acquire this lock after Awaken by a signal operation.
    conditionLock->Acquire();

    (void) interrupt->SetLevel(oldLevel);
}
```

Signal 操作会释放等待队列的队首线程（如果队列不为空），为了保证原子性，仍应进行中断的屏蔽和恢复。因此内容如下：

```
void Condition::Signal(Lock* conditionLock)
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    // Step 0: Check whether current thread is holding this lock.
    ASSERT(conditionLock->isHeldByCurrentThread());

    // Step 1: Wake up the first thread in waitQueue.
    if(!waitQueue->IsEmpty())
    {
        Thread* t = (Thread*) waitQueue->Remove();
        scheduler->ReadyToRun(t);
    }

    (void) interrupt->SetLevel(oldLevel);
}
```

Broadcast 操作会把队列中的所有线程都释放，但功能与 Signal 相似，只需要把 if 变为 while，以把所有等待线程唤醒。因此实现如下：

```
void Condition::Broadcast(Lock* conditionLock)
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    // Step 0: Check whether current thread is holding this lock.
    ASSERT(conditionLock->isHeldByCurrentThread());

    // Step 1: Wake up all threads in waitQueue.
    while(!waitQueue->IsEmpty())
    {
        Thread* t = (Thread*) waitQueue->Remove();
        scheduler->ReadyToRun(t);
    }

    (void) interrupt->SetLevel(oldLevel);
}
```

这样，整个 Condition 类便实现完成了。

Exercise4

这一部分中，我选择实现生产者-消费者问题的解决，采用信号量和锁+条件变量两种方式来解决。

生产者——消费者问题的大致情境如下：有一个有限的共享缓冲区，有生产者和消费者两类操作者。其中，生产者不断向每个空位内写入内容，消费者不断地从已写入内容的位置读取内容。我们需要避免在缓冲区满时生产者继续向缓冲区加入数据，同时避免缓冲区全部为空时消费者读取数据。

（一）信号量+锁（一元信号量）解决生产者——消费者问题

在这个问题中，临界区是对缓冲区的生产/消费操作。为了解决这个问题，我们需要让生产者在缓冲区为满时进行等待，让消费者在缓冲区为空时也进行等待。

所以，我们要设置满缓冲区个数 **Full** 和空缓冲区个数 **Empty** 两个信号量，起始时 **Full** 为 0，**Empty** 为 **N**（**N** 为缓冲区大小）。当生产者准备生产时，要用 **P** 操作获取空缓冲区，然后找到一个空缓冲区进行写操作，写完以后，再用 **V** 操作“释放”一个满缓冲区的位置。对称地，当消费者准备消费时，要用 **P** 操作获取一个满的缓冲区，然后找到一个满缓冲区进行读操作，读完以后，再用 **V** 操作“释放”一个空缓冲区的位置。

更进一步地，为了保证互斥地访问缓冲区（避免出现读写同时发生或者写写同时发生），我们再用一个锁（或者一元信号量）**Mutex** 保证对整个缓冲区的互斥访问，这样便可以解决上述问题。

Nachos 的代码实现如下：

缓冲区中的元素简化为只有一个 **int** 的类：

```
class Element
{
    public:
        int value;
};
```

缓冲区和读写操作的定义如下，简便起见约定每次写会向后扩展，而读会读走最后一个元素：

```

void WriteBuffer(Element* product)
{
    //printf("0\n");
    Empty->P();
    //printf("1\n");
    Mutex->Acquire();
    // printf("pointer is %ld\n", product);

    /* Critical Rigion */
    list[count++] = *product;
    /* Critical Region */

    Mutex->Release();
    Full->V();
}

Element* ReadBuffer()
{
    Element* item;

    Full->P();
    Mutex->Acquire();

    /* Critical Rigion */
    item = &list[count-1];
    count--;
    /* Critical Region */

    Mutex->Release();
    Empty->V();

    return item;
}

```

缓冲区的成员变量和其他函数如下，printBuffer 用于打印缓冲区：

```

#define BUF_LEN 5

class Buffer
{
private:
    Semaphore* Empty;
    Semaphore* Full;
    Lock* Mutex;
    int count;
    Element list[BUF_LEN];
}

Buffer()
{
    Empty = new Semaphore("Empty Count", BUF_LEN);
    Full = new Semaphore("Full Count", 0);
    Mutex = new Lock("Buffer Mutex");
    count = 0;
}

~Buffer()
{
    delete list;
}

void printBuffer()
{
    printf("Buffer(size: %d, count: %d): [", BUF_LEN, count);
    for(int i=0; i<count; ++i)
    {
        printf("%d, ", list[i].value);
    }
    for(int i=count; i<BUF_LEN; ++i)
    {
        printf("__, ");
    }
    printf("]\n");
}

```

生产者和消费者线程内容如下：

```

void
ProducerThread(int iters)
{
    for(int i=0; i<iters; ++i)
    {
        int TID = currentThread->getTID();
        printf("## Thread %s produces!!!\n", currentThread->getName());
        Element item = ProduceItem(i+TID);
        //printf("tmp ptr is %ld\n", & item);
        globalBuf->WriteBuffer(& item);
        printf("Produce Element with value %d\n", item.value);
        globalBuf->printBuffer();

        interrupt->SetLevel(IntOff);
        interrupt->SetLevel(IntOn);
    }
}

```



```

void
ConsumerThread(int iters)
{
    for(int i=0; i<iters; ++i)
    {
        int TID = currentThread->getTID();
        printf("## Thread %s consumes!!!\n", currentThread->getName());
        Element* item = globalBuf->ReadBuffer();
        ConsumeItem(item);
        globalBuf->printBuffer();

        interrupt->SetLevel(IntOff);
        interrupt->SetLevel(IntOn);
    }
}

```

其中的生产和消费函数内容如下：

```

Element
ProduceItem(int value)
{
    Element item;
    //printf("0");
    item.value = value;
    return item;
}

void
ConsumeItem(Element* item)
{
    printf("Consume item with value %d\n", item->value);
    // delete item;
}

```

测试代码如下，创建了 4 个线程，两个生产者两个消费者：

```

void
Lab3Test4_1()
{
    DEBUG('t', "Entering Lab3Test4_1.");

    globalBuf = new Buffer();

    Thread *producer1 = new Thread("Producer 1");
    Thread *producer2 = new Thread("Producer 2");
    Thread *consumer1 = new Thread("Consumer 1");
    Thread *consumer2 = new Thread("Consumer 2");

    producer1->Fork(ProducerThread, (void*)7);
    consumer1->Fork(ConsumerThread, (void*)2);
    consumer2->Fork(ConsumerThread, (void*)8);
    producer2->Fork(ProducerThread, (void*)3);
}

```

测试结果如下：

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_diantl/nachos-3.4/code/threads$ ./nachos -q 11
## Thread Producer 1 produces!!!
Produce Element with value 1
Buffer(size: 5, count: 1): [1, __, __, __, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 2
Buffer(size: 5, count: 2): [1, 2, __, __, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 3
Buffer(size: 5, count: 3): [1, 2, 3, __, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 4
Buffer(size: 5, count: 4): [1, 2, 3, 4, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 5
Buffer(size: 5, count: 5): [1, 2, 3, 4, 5, ]
## Thread Producer 1 produces!!!
## Thread Consumer 1 consumes!!!
Consume item with value 5
Buffer(size: 5, count: 4): [1, 2, 3, 4, __, ]
## Thread Consumer 1 consumes!!!
Consume item with value 4
Buffer(size: 5, count: 3): [1, 2, 3, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 3
Buffer(size: 5, count: 2): [1, 2, __, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 2
Buffer(size: 5, count: 1): [1, __, __, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 1
Buffer(size: 5, count: 0): [__, __, __, __, __, ]
## Thread Consumer 2 consumes!!!
## Thread Producer 2 produces!!!
Produce Element with value 2
Buffer(size: 5, count: 1): [2, __, __, __, __, ]
## Thread Producer 2 produces!!!
Produce Element with value 3
Buffer(size: 5, count: 2): [2, 3, __, __, __, ]
## Thread Producer 2 produces!!!
Produce Element with value 4
Buffer(size: 5, count: 3): [2, 3, 4, __, __, ]
Produce Element with value 6

```

```

Buffer(size: 5, count: 4): [2, 3, 4, 6, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 7
Buffer(size: 5, count: 5): [2, 3, 4, 6, 7, ]
Consume item with value 7
Buffer(size: 5, count: 4): [2, 3, 4, 6, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 6
Buffer(size: 5, count: 3): [2, 3, 4, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 4
Buffer(size: 5, count: 2): [2, 3, __, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 3
Buffer(size: 5, count: 1): [2, __, __, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 2
Buffer(size: 5, count: 0): [__, __, __, __, __, ]
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1090, idle 0, system 1090, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

```

可以看到，当缓冲区满时，生产者会让出 CPU，等待在缓冲区外，消费者便会进行消费。当缓冲区为空时，消费者让出 CPU，在缓冲区外等待。而且没有同时读写或者多生产者同时生产的现象发生。可以看到，上述信号量机制解决生产者-消费者问题的实现是正确的，而且 Exercise3 中的 Lock 实现也是正确的。（Lock 替换为一个初值为 1 的信号量，获得的效果相同）

注：由于 Nachos 没有真正的外部时钟中断，因此只能手动模拟时钟中断。如果加上-rs 模拟随机时间片，也可以得到正确的结果（但由于输出不是原子的，所以控制台的显示也会有所不同）

（二）锁+条件变量解决生产者——消费者问题

接下来尝试使用条件变量来解决这个问题。大致思路如下：

缓冲区由一个互斥锁 Mutex 和两个条件变量 Producer 和 Consumer 来控制，并且允许生产者和消费者获取当前缓冲区中的元素个数。

1.生产者的执行流程如下：

- （1）生产产品后，试图获取 Mutex。
- （2）获取到 Mutex 后，检查当前缓冲区是否为满，如果缓冲区满，调用 Producer 的 Wait，释放 Mutex，并等待在这个条件变量的队列中。
- （3）等待结束或者缓冲区没满（此时仍然会占用锁，Wait 会在重新占用锁之后返回），则把产品放入缓冲区。
- （4）此时检查缓冲区中产品个数，如果发现个数为 1（放入的是第一个），就对 Consumer 发出 Signal，把一个消费者放入就绪队列准备运行。
- （5）释放锁

2.消费者的执行流程如下：

- （1）准备消费产品时，试图获取 Mutex
- （2）获取到 Mutex 后，检查缓冲区是否为空，如果为空，调用 Consumer 的 Wait，释放 Mutex，在 Consumer 队列中等待。
- （3）等待结束或者缓冲区不为空，取出一个产品并消费。
- （4）检查缓冲区产品个数，如果取出的恰好是第 N 个产品（N 为缓冲区容量），那么向 Producer 发出 Signal，把一个生产者放入就绪队列，可以被调度运行。

代码实现如下：

Element 和先前的定义相同。

```
class Element
{
public:
    int value;
};
```

缓冲区中除了缓冲数组，还应该包含两个条件变量，和一个互斥锁，以及一个计数变量。构造函数和析构函数如下所示：

```
private:
    Condition* Producer;
    Condition* Consumer;
    Lock* Mutex;
    int count;
    Element list[BUF_LEN];

public:
    BufferCnd()
    {
        Producer = new Condition("Producer variable");
        Consumer = new Condition("Consumer variable");
        Mutex = new Lock("Buffer Mutex");
        count = 0;
    }

    ~BufferCnd()
    {
        delete list;
    }
```

缓冲区的读写操作如下，就是把上述流程代码化。但注意到 **Signal** 等价于 Mesa 管程意义中的 **notify**，因此应该进行一些修改：

```
void WriteBuffer(Element* product)
{
    Mutex->Acquire();

    while(count==BUF_LEN)
    {
        printf("#### FULL!! ####\n");
        Producer->Wait(Mutex);
    }

    /* Critical Region */
    list[count++] = *product;
    /* Critical Region */

    Consumer->Signal(Mutex);

    Mutex->Release();
}

Element* ReadBuffer()
{
    Element* item;

    Mutex->Acquire();

    while(count==0)
    {
        printf("#### EMPTY!! ####\n");
        Consumer->Wait(Mutex);
    }

    /* Critical Region */
    item = &list[count-1];
    count--;
    /* Critical Region */

    Producer->Signal(Mutex);

    Mutex->Release();

    return item;
}
```

除了这两个函数，仍然为这个缓冲区提供了打印缓冲区内容的方法：

```
void printBuffer()
{
    printf("Buffer(size: %d, count: %d): [", BUF_LEN, count);
    for(int i=0; i<count; ++i)
    {
        printf("%d, ", list[i].value);
    }
    for(int i=count; i<BUF_LEN; ++i)
    {
        printf("__, ");
    }
    printf("]\n");
}
```

生产者和消费者的操作如下：

```

void
ProducerThreadCnd(int iters)
{
    for(int i=0; i<iters; ++i)
    {
        int TID = currentThread->getTID();
        printf("## Thread %s produces!!!\n", currentThread->getName());
        Element item = ProduceItem(i+TID);
        //printf("tmp ptr is %ld\n", & item);
        printf("Produce Element with value %d\n", item.value);
        globalBufCnd->WriteBuffer(& item);
        globalBufCnd->printBuffer();

        interrupt->SetLevel(IntOff);
        interrupt->SetLevel(IntOn);
    }
}

void
ConsumerThreadCnd(int iters)
{
    for(int i=0; i<iters; ++i)
    {
        int TID = currentThread->getTID();
        printf("## Thread %s consumes!!!\n", currentThread->getName());
        Element* item = globalBufCnd->ReadBuffer();
        ConsumeItem(item);
        globalBufCnd->printBuffer();

        interrupt->SetLevel(IntOff);
        interrupt->SetLevel(IntOn);
    }
}

```

其中的 Produce 和 Consume 与之前相同。

测试代码如下。生产者和消费者的个数操作与信号量解法相同：

```

void
Lab3Test4_2()
{
    DEBUG('t', "Entering Lab3Test4_2.");

    globalBufCnd = new BufferCnd();

    Thread *producer1 = new Thread("Producer 1");
    Thread *producer2 = new Thread("Producer 2");
    Thread *consumer1 = new Thread("Consumer 1");
    Thread *consumer2 = new Thread("Consumer 2");

    producer1->Fork(ProducerThreadCnd, (void*)7);
    consumer1->Fork(ConsumerThreadCnd, (void*)2);
    consumer2->Fork(ConsumerThreadCnd, (void*)8);
    producer2->Fork(ProducerThreadCnd, (void*)3);
}

```

运行结果如下：

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/threads$ ./nachos -q 12
## Thread Producer 1 produces!!!
Produce Element with value 1
Buffer(size: 5, count: 1): [1, __, __, __, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 2
Buffer(size: 5, count: 2): [1, 2, __, __, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 3
Buffer(size: 5, count: 3): [1, 2, 3, __, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 4
Buffer(size: 5, count: 4): [1, 2, 3, 4, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 5
Buffer(size: 5, count: 5): [1, 2, 3, 4, 5, ]
## Thread Producer 1 produces!!!
Produce Element with value 6
##### FULL!! #####
## Thread Consumer 1 consumes!!!
Consume item with value 5
Buffer(size: 5, count: 4): [1, 2, 3, 4, __, ]
## Thread Consumer 1 consumes!!!
Consume item with value 4
Buffer(size: 5, count: 3): [1, 2, 3, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 3
Buffer(size: 5, count: 2): [1, 2, __, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 2
Buffer(size: 5, count: 1): [1, __, __, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 1
Buffer(size: 5, count: 0): [__, __, __, __, __, ]
## Thread Consumer 2 consumes!!!
##### EMPTY!! #####
## Thread Producer 2 produces!!!
Produce Element with value 2
Buffer(size: 5, count: 1): [2, __, __, __, __, ]
## Thread Producer 2 produces!!!
Produce Element with value 3

```

```

Buffer(size: 5, count: 2): [2, 3, __, __, __, ]
## Thread Producer 2 produces!!!
Produce Element with value 4
Buffer(size: 5, count: 3): [2, 3, 4, __, __, ]
Buffer(size: 5, count: 4): [2, 3, 4, 6, __, ]
## Thread Producer 1 produces!!!
Produce Element with value 7
Buffer(size: 5, count: 5): [2, 3, 4, 6, 7, ]
Consume item with value 7
Buffer(size: 5, count: 4): [2, 3, 4, 6, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 6
Buffer(size: 5, count: 3): [2, 3, 4, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 4
Buffer(size: 5, count: 2): [2, 3, __, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 3
Buffer(size: 5, count: 1): [2, __, __, __, __, ]
## Thread Consumer 2 consumes!!!
Consume item with value 2
Buffer(size: 5, count: 0): [__, __, __, __, __, ]
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 750, idle 0, system 750, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

```

可以看到，当缓冲区满时，会调用条件变量 `Producer` 的 `Wait`，然后 `Consumer` 开始消费。而当缓冲区为空时，会调用条件变量 `Consumer` 的 `Wait`，然后 `Producer` 开始消费。这样解决了我们提出的两个问题，且没造成死锁等现象。同时，这一结果也验证了我编写的条件变量的正确性。

注：由于 `Nachos` 没有真正的外部时钟中断，因此只能手动模拟时钟中断。如果加上 `rs` 模拟随机时间片，也可以得到正确的结果（但由于输出不是原子的，所以控制台的显示也会有所不同）

Challenge

这一部分中，我完成了 `Clallenge1` 和 `Challenge2`

（一）Barrier 的实现

`Barrier` 是一种线程协调机制，它会在所有线程都完成某个操作后，才一起释放这些线程。在 `C++`（since `C++20`）自己的实现中，`std::experimental::barrier` 类中包含了以下方法：

Member functions	
(constructor)	constructs a barrier (public member function)
(destructor)	destroys the barrier (public member function)
operator= [deleted]	not copy-assignable (public member function)
arrive_and_wait	arrive at the synchronization point and block (public member function)
arrive_and_drop	arrive at the synchronization point and remove the current thread from the set of participating threads (public member function)

因此，根据题目中的要求，我们需要实现类似 `arrive_and_wait` 的方法。

仿照这个实现，在 `synch.h` 中声明 `Barrier` 类如下：

```
class Barrier
{
public:
    Barrier(char* debugName, int thread_number);
    ~Barrier();
    char* getName() { return (name); }
    void ArriveAndWait();

private:
    char* name; // useful for debugging
    int num_threads; // total number of threads
    int num_left; // number of threads haven't reached
    Lock* Mutex; // Used together with condition variable
    Condition* barrier; // Block all threads and use Broadcast to wake them up.
};
```

在给出具体实现之前，我先来阐述一下实现这一机制的思路。

观察我们已有的同步机制，Condition 中的 Wait 和 Broadcast 操作与我们的目标比较接近。我们可以在 Barrier 类中记录线程总数和还没到达 arrive_and_wait 函数的线程个数。当最后一个线程进入 arrive_and_wait 之后（还没到达的个数变为 0），Barrier 中的条件变量可以调用 Broadcast，把所有线程放入就绪队列中；当还有线程没到达时，我们可以调用条件变量中的 Wait 来等待。为了使用条件变量，Barrier 中除了线程个数和条件变量以外，还需要有一个互斥锁。

根据上面的分析，我们的 arrive_and_wait 方法可以这样来实现：

```
void Barrier::ArriveAndWait()
{
    IntStatus oldStatus = interrupt->SetLevel(IntOff);

    Mutex->Acquire();

    num_left--;
    if(num_left==0) // all threads reach this point
    {
        barrier->Broadcast(Mutex);
        num_left = num_threads; // make this barrier reuseable
    }
    else
    {
        barrier->Wait(Mutex); // wait for other threads
    }

    Mutex->Release();

    (void) interrupt->SetLevel(oldStatus);
}
```

Barrier 的构造函数和析构函数如下：

```
Barrier::Barrier(char*debugName, int thread_number)
{
    name = debugName;
    num_threads = thread_number;
    num_left = num_threads;
    Mutex = new Lock("Lock in Barrier");
    barrier = new Condition("Condition in Barrier");
}

Barrier::~Barrier()
{
    delete Mutex;
    delete barrier;
}
```

现在我们来编写代码测试它。我们让每个线程和 main 线程执行三次 arrive_and_wait，代码如下：


```

void
BarrierThread(int num)
{
    printf("## Thread %s with num [%d] reached 1st barrier!\n", currentThread->getName(), num);
    Bar->ArriveAndWait();
    printf("## Thread %s with num [%d] crossed over barrier 1 and reached barrier 2!\n", currentThread->getName(), num);
    Bar->ArriveAndWait();
    printf("## Thread %s with num [%d] crossed over barrier 2 and reached barrier 3!\n", currentThread->getName(), num);
    Bar->ArriveAndWait();
    printf("## Thread %s with num [%d] crossed all barriers and reached deadline!\n");
}

void
Lab3Challenge1()
{
    DEBUG('t', "Entering Lab3Challenge1.");

    int num_threads = 10;
    Bar = new Barrier("Lab3 Test", num_threads+1); // main thread should also be included

    for(int i=0; i<num_threads; ++i)
    {
        Thread* t = new Thread("[Test]");
        t->Fork(BarrierThread, (void*)i);
    }

    TS();

    printf("main starts to cross barrier 1!\n");
    //currentThread->Yield();
    Bar->ArriveAndWait();
    printf("main crossed over barrier 1 and reached barrier 2!\n");
    //currentThread->Yield();
    Bar->ArriveAndWait();
    printf("main crossed over barrier 1 and reached barrier 2!\n");
    //currentThread->Yield();
    Bar->ArriveAndWait();
    printf("main crossed over all barriers and reached DDL!\n");
}

```

运行结果如下:

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_diantl/nachos-3.4/code/threads$ ./nachos -d b -q 13
UID    TID    PRI    NAME    STAT
0       0       0      main    RUNNING
0       1       0      [Test]  READY
0       2       0      [Test]  READY
0       3       0      [Test]  READY
0       4       0      [Test]  READY
0       5       0      [Test]  READY
0       6       0      [Test]  READY
0       7       0      [Test]  READY
0       8       0      [Test]  READY
0       9       0      [Test]  READY
0      10       0      [Test]  READY
main starts to cross barrier 1!
## Thread [Test] with num [0] reached 1st barrier!
## Thread [Test] with num [1] reached 1st barrier!
## Thread [Test] with num [2] reached 1st barrier!
## Thread [Test] with num [3] reached 1st barrier!
## Thread [Test] with num [4] reached 1st barrier!
## Thread [Test] with num [5] reached 1st barrier!
## Thread [Test] with num [6] reached 1st barrier!
## Thread [Test] with num [7] reached 1st barrier!
## Thread [Test] with num [8] reached 1st barrier!
## Thread [Test] with num [9] reached 1st barrier!
##### Start to broadcast! #####
## Thread [Test] with num [9] crossed over barrier 1 and reached barrier 2!
main crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [0] crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [1] crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [2] crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [3] crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [4] crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [5] crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [6] crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [7] crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [8] crossed over barrier 1 and reached barrier 2!
##### Start to broadcast! #####

```

```

## Thread [Test] with num [8] crossed over barrier 2 and reached barrier 3!
## Thread [Test] with num [9] crossed over barrier 2 and reached barrier 3!
main crossed over barrier 1 and reached barrier 2!
## Thread [Test] with num [0] crossed over barrier 2 and reached barrier 3!
## Thread [Test] with num [1] crossed over barrier 2 and reached barrier 3!
## Thread [Test] with num [2] crossed over barrier 2 and reached barrier 3!
## Thread [Test] with num [3] crossed over barrier 2 and reached barrier 3!
## Thread [Test] with num [4] crossed over barrier 2 and reached barrier 3!
## Thread [Test] with num [5] crossed over barrier 2 and reached barrier 3!
## Thread [Test] with num [6] crossed over barrier 2 and reached barrier 3!
## Thread [Test] with num [7] crossed over barrier 2 and reached barrier 3!
##### Start to broadcast! #####
## Thread [Test] with num [7] crossed all barriers and reached deadline!
## Thread [Test] with num [8] crossed all barriers and reached deadline!
## Thread [Test] with num [9] crossed all barriers and reached deadline!
main crossed over all barriers and reached DDL!
## Thread [Test] with num [0] crossed all barriers and reached deadline!
## Thread [Test] with num [1] crossed all barriers and reached deadline!
## Thread [Test] with num [2] crossed all barriers and reached deadline!
## Thread [Test] with num [3] crossed all barriers and reached deadline!
## Thread [Test] with num [4] crossed all barriers and reached deadline!
## Thread [Test] with num [5] crossed all barriers and reached deadline!
## Thread [Test] with num [6] crossed all barriers and reached deadline!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 540, idle 0, system 540, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
Cleaning up...

```

可以看到，只有当所有进程都进入 `arrive_and_wait` 方法中以后，每个线程才会继续向前运行。因此，上述实现方式是正确的。

（二）Read-Write Lock 的实现

这个锁主要用于解决读者-写者问题。我们不妨考虑第一类读者-写者问题：无其他读者、写者时，读者可以读；写者发现有读者正在读时需要等待，直到所有读者都读完；每个时刻只能有一个写者在写。

我们可以这样来设计读写锁：用一个变量来计数读者数量 `ReaderCnt`，同时用一个 `Mutex` 变量 `ReaderLock` 来保护它，再用一个 `Mutex` 变量 `WriterLock` 来保证只有一个写者在写。那么每次读者读之前，首先互斥的递增 `ReaderCnt`。在这个过程中，如果读者发现自己是第一个读者，那么同时占用写者的锁，以防止写者在读者读取时进行写操作。当读者读完以后，再次互斥地对 `ReaderCnt` 递减，如果当前读者发现递减以后值为 0，即自己是最后一个读者，那么释放写者锁。

根据上述思路，可以设计读写锁如下。注意由于 Nachos 要求 `Lock` 的持有者和释放着

应该相同，因此 WriterLock 采用一元信号量 Semaphore 实现（虽然与 Lock 等价），因为第一个来的读者和最后一个走的读者可能不相同：

```
class ReadWriteLock
{
public:
    ReadWriteLock(char* debugName);
    ~ReadWriteLock();
    char* getName() { return (name); }

    void AcquireReaderLock();
    void ReleaseReaderLock();

    void AcquireWriterLock();
    void ReleaseWriterLock();

private:
    char* name;
    Lock* ReaderLock;
    Semaphore* WriterLock;
    int ReaderCnt;
};
```

构造函数和析构函数就是初始化和销毁各个变量：

```
ReadWriteLock::ReadWriteLock(char* debugName)    ReadWriteLock::~~ReadWriteLock()
{
    name = debugName;
    ReaderCnt = 0;
    ReaderLock = new Lock("Reader lock");
    WriterLock = new Semaphore("Writer Lock", 1);
}
{
    delete ReaderLock;
    delete WriterLock;
}
```

核心方法是读写锁的获取和释放，内容如下。注意读者获取读锁也要对写者锁做操作：

```
void
ReadWriteLock::AcquireReaderLock()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ReaderLock->Acquire();
    ReaderCnt++;
    if(ReaderCnt==1)
    {
        WriterLock->P();
    }
    ReaderLock->Release();

    interrupt->SetLevel(oldLevel);
}

void
ReadWriteLock::ReleaseReaderLock()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ReaderLock->Acquire();
    ReaderCnt--;
    if(ReaderCnt==0)
    {
        WriterLock->V();
    }
    ReaderLock->Release();

    interrupt->SetLevel(oldLevel);
}
```

```

void ReadWriteLock::AcquireWriterLock()
{
    WriterLock->P();
}

void ReadWriteLock::ReleaseWriterLock()
{
    WriterLock->V();
}

```

下面编写测试代码。这里我把 **Buffer** 简化为一个整数变量。除此之外，每次在缓冲区中时，让线程调用 **Yield**，模拟占用缓冲区的情形：

```

void ReaderThread(int num)
{
    for(int i=0; i<10; ++i)
    {
        printf("Reader %d comes!\n", num);
        RWLock->AcquireReaderLock();

        printf("Reader %d is reading the buffer!\n", num);
        currentThread->Yield();
        printf("Reader %d says The Buffer is %d\n",num, Buff);

        RWLock->ReleaseReaderLock();
    }
}

void WriterThread(int num)
{
    for(int i=0; i<10; ++i)
    {
        printf("Writer %d comes!\n", num);
        RWLock->AcquireWriterLock();

        printf("Writer %d is writing the buffer!\n", num);
        currentThread->Yield();
        Buff+=num+i;

        RWLock->ReleaseWriterLock();
    }
}

void Lab3Challenge2()
{
    DEBUG('t', "Entering Lab3Challenge2.");

    RWLock = new ReadWriteLock("Lab 3 Challenge 2");
    int numReader = 2;

    Thread* t1 = new Thread("Writer");
    t1->Fork(WriterThread, (void*)0);

    for(int i=0; i<numReader; ++i)
    {
        Thread* t = new Thread("Reader");
        t->Fork(ReaderThread, (void*)i);
    }

    Thread* t2 = new Thread("Writer");
    t2->Fork(WriterThread, (void*)1);

    TS();
    printf("##### Test starts! #####\n");
}

```

运行结果如下：

[illegible]

```
Writer 1 comes!  
Writer 1 is writing the buffer!  
Writer 1 comes!  
Writer 1 is writing the buffer!  
Writer 1 comes!  
Writer 1 is writing the buffer!  
Writer 1 comes!  
Writer 1 is writing the buffer!  
Reader 0 is reading the buffer!  
Reader 1 is reading the buffer!  
Reader 0 says The Buffer is 114614  
Reader 0 comes!  
Reader 0 is reading the buffer!  
Reader 1 says The Buffer is 114614  
Reader 1 comes!  
Reader 1 is reading the buffer!  
Reader 0 says The Buffer is 114614  
Reader 0 comes!  
Reader 0 is reading the buffer!  
Reader 1 says The Buffer is 114614  
Reader 1 comes!  
Reader 1 is reading the buffer!  
Reader 0 says The Buffer is 114614  
Reader 0 comes!  
Reader 0 is reading the buffer!  
Reader 1 says The Buffer is 114614  
Reader 1 comes!  
Reader 1 is reading the buffer!  
Reader 0 says The Buffer is 114614  
Reader 0 comes!
```

```

Reader 1 comes!
Reader 1 is reading the buffer!
Reader 0 says The Buffer is 114614
Reader 0 comes!
Reader 0 is reading the buffer!
Reader 1 says The Buffer is 114614
Reader 1 comes!
Reader 1 is reading the buffer!
Reader 0 says The Buffer is 114614
Reader 0 comes!
Reader 0 is reading the buffer!
Reader 1 says The Buffer is 114614
Reader 1 comes!
Reader 1 is reading the buffer!
Reader 0 says The Buffer is 114614
Reader 0 comes!
Reader 0 is reading the buffer!
Reader 1 says The Buffer is 114614
Reader 1 comes!
Reader 1 is reading the buffer!
Reader 0 says The Buffer is 114614
Reader 0 comes!
Reader 0 is reading the buffer!
Reader 1 says The Buffer is 114614
Reader 1 comes!
Reader 1 is reading the buffer!
Reader 0 says The Buffer is 114614
Reader 1 says The Buffer is 114614
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1290, idle 0, system 1290, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

```

可以看到，当开始 Writer 占据缓冲区时，其他读者写者都会等待。而 Reader 占用缓冲区时，也允许其他 Reader 进入缓冲区来读数据。

为了测试读者在时对写者的排斥，编写测试代码如下：

```

void
Lab3Challenge2_1()
{
    DEBUG('t', "Entering Lab3Challenge2_1.");

    RWLock = new ReadWriteLock("Lab 3 Challenge 2_1");
    int numReader = 2;

    for(int i=0; i<numReader; ++i)
    {
        Thread* t = new Thread("Reader");
        t->Fork(ReaderThread, (void*)i);
    }

    Thread* t1 = new Thread("Writer");
    t1->Fork(WriterThread, (void*)0);

    Thread* t2 = new Thread("Writer");
    t2->Fork(WriterThread, (void*)1);

    TS();
    printf("##### Test starts! #####\n");
}

```

运行结果如下：

[illegible]

让我理清了 **Challenge** 中的概念，学到了课堂上没有的新知识。

除此之外，对经典同步互斥问题的解决也让我明白了如何应用这些同步数据结构，对互斥和同步的处理有了更深的了解。

内容三：对课程或 Lab 的意见和建议

希望可以把引导手册进一步完善，现在每次阅读代码的时候需要我们递归阅读和寻找其他额外的代码，但我们往往不知道应该去哪里能找到，这样浪费了许多不必要的时间。我觉得可以把某些常用函数（或者重要辅助函数）的位置标注出来，这样可以节省遍历搜索的时间。

也希望可以为调研类的作业提供更多的提示和说明。现在我们只能漫无目的地去遍历搜索百度、google 的结果，我们无法确认博客等网站表述内容的正确性和严谨性，而且有的调研题目很难找到相关博客。让我们在短时间内查询所有体系结构/操作系统的手册，对于还有许多其他课程的本科生来说在时间上不是很现实。所以希望课程可以提供更多相关资源，或者提示我们应该去看手册的哪些章节，去找什么样的权威资料。这样既可以让我们学到准确的知识，也可以提升我们的学习效率。

内容四：参考文献

1.Linux 内核中的同步机制：

<https://blog.csdn.net/yiqiaoxihui/article/details/82026389>

2.Linux Documentation：

<https://github.com/torvalds/linux/tree/b51594df17d0ce80b9f9f35394a1f42d7ac94472/Documentation/locking>

3.Linux 源码：

<https://github.com/torvalds/linux/blob/master/include/linux/mutex.h>

4.顺序锁：<https://blog.csdn.net/longwang155069/article/details/52231519>

5.C++的 barrier 注解：<https://en.cppreference.com/w/cpp/experimental/barrier>

6.barrier 的 Wiki 描述：[https://en.wikipedia.org/wiki/Barrier_\(computer_science\)](https://en.wikipedia.org/wiki/Barrier_(computer_science))

7.关于 barrier 的一些讨论：<https://stackoverflow.com/questions/38999911/what->

[is-the-best-way-to-realize-a-synchronization-barrier-between-threads](#)

8. Synchronization, Part 7: The Reader Writer Problem:

<https://github.com/angrave/SystemProgramming/wiki/Synchronization%2C-Part-7%3A-The-Reader-Writer-Problem>

9. Reader-Writer lock 的 Wiki:

https://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock