Lab3 同步机制实习报告

姓名 姜慧强 学号 1801210840
日期 2019.03.17

目录

内容一：总体概述

本次实验室操作系统高级课程的第三次实验。在 Lab2 阅读线程调度相关代码并完成实现基于优先级的调度算法，完成了基于时间片的调度算法的基础上，主要完成了线程间同步机制相关代码阅读。线程间同步主要是通过信号量，锁，条件变量等方式实现的。并在代码阅读的基础上，完成了基于信号量、条件变量解决了“生产者-消费者问题”。之后还通过条件变量及 Lock，实现了 barrier 强化版-两阶段锁协议，完成了读者、写者问题的解决。通过 lab3 的实践，强化了对同步类，List 类，Thread 类，调度类等代码的理解。

内容二：任务完成情况

任务完成列表（Y/N）

EXER1	EXER2	EXER3	EXER4	CHANELLAGE1	CHANELLAGE2	CHANELLAGE3
Y	Y	Y	Y	Y	Y	N

具体 Exercise 的完成情况

Exercise1

这里的同步机制一般指的是线程同步机制，进程同步机制会更复杂一些。

Linux 的线程同步机制和 Nachos 中使用的机制(信号量，锁，条件变量)基本一致。采用了互斥量 mutex，条件变量，信号量，读写锁。

- Mutex

Linux kerenal 中关于 Mutex struct 的代码在<include/linux/mutex.h>中

```
struct mutex {
        atomic_long_t           owner;      // mutex 获得的 owner ID
                                            // 若==0，则 mutex 未被占用;
                                            // 若> 0，则 mutex 被此 ownerId 占用,
                                            // 只能由当前 owner 解 mutex
        spinlock_t              wait_lock; // 自旋锁类型
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
        struct list_head        wait_list; // 等待队列
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
```

```
        struct lockdep_map      dep_map;
#endif
};
```

上面的 struct 定义了一个原子变量用于维护 mutex 的互斥效果。另外定义了一个 wait_list 用于存储 sleep 的 thread。这部分代码和 nachos 中 Semaphore 的设计基本一致。

而具体实现 mutex 的代码位于<kernel/locking/mutex.c>中。__mutex_init 函数主要做一些变量声明和初始化的工作。

```
void
__mutex_init(struct mutex *lock, const char *name, struct lock_class_key *key)
{
        atomic_long_set(&lock->owner, 0);     // init atomic 变量 owner
        spin_lock_init(&lock->wait_lock);       // init 自旋锁类型变量
        INIT_LIST_HEAD(&lock->wait_list);     // init 等待队列变量
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        osq_lock_init(&lock->osq);
#endif
        debug_mutex_init(lock, name, key);
}
```

以加锁为例，调用的的是 mutex_lock 函数。

```
void __sched mutex_lock(struct mutex *lock)
{
        might_sleep();                          // 打印堆栈 debug sleep

        if (!__mutex_trylock_fast(lock))        // atomic 获得 owner，如果能
                __mutex_lock_slowpath(lock); //
}
EXPORT_SYMBOL(mutex_lock);
#endif
```

其中，might_sleep()是一个全局 Linux API，主要用于在中断时候，debug 打印 context 堆栈。__mutex_trylock_fast(lock) 是一个去获取 lock 的 owner 的函数，如果能获取则返回 true

```
static __always_inline bool __mutex_trylock_fast(struct mutex *lock)
{ww_acquire_ctx
        unsigned long curr = (unsigned long)current;
        unsigned long zero = 0UL;
        if (atomic_long_try_cmpxchg_acquire(&lock->owner, &zero, curr))   // 获取 owner
                return true;
        return false;
}
```

如果有权限获取 owner 则

```
static noinline void __sched
__mutex_lock_slowpath(struct mutex *lock)
{
```

```
        __mutex_lock(lock, TASK_UNINTERRUPTIBLE, 0, NULL, _RET_IP_);   // 调用__mutex_lock
}
```

然后再嵌套调用，不知道是为了什么，写了那么多层

```
static int __sched
__mutex_lock(struct mutex *lock, long state, unsigned int subclass,
                struct lockdep_map *nest_lock, unsigned long ip)
{
        // 调用__mutex_lock_common
        return __mutex_lock_common(lock, state, subclass, nest_lock, ip, NULL, false);
}
```

然后就到了 Linux 真正处理 mock_lock 的地方

```
static __always_inline int __schedw
__mutex_lock_common(struct mutex *lock, long state, unsigned int subclass,
                        // lock TASK_UNINTERRUPTIBLE 0
                    struct lockdep_map *nest_lock, unsigned long ip,        // NULL _RET_IP_
                    struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx) // NULL false
{
        struct mutex_waiter waiter;
        bool first = false;
        struct ww_mutex *ww;       // ww = wound/wait mutex 用于死锁检验
        int ret;

        might_sleep(); // 一样的去打印 context 的堆栈

        ww = container_of(lock, struct ww_mutex, base); // 获得 ww_mutex
        if (use_ww_ctx && ww_ctx) {                          // mutet_lock 进不到这
个,ww_mutex_lock 有可能进
                if (unlikely(ww_ctx == READ_ONCE(ww->ctx))) // ww_mutex 获得的 ctx 和需要的
ctx 对比
                        return -EALREADY;

                /*
                 * Reset the wounded flag after a kill. No other process can
                 * race and wound us here since they can't have a valid owner
                 * pointer if we don't have any locks held.
                 */
                if (ww_ctx->acquired == 0)     // 如果 ww_ctx 没有被获得 则重设 wounded 位
                        ww_ctx->wounded = 0;
        }

        preempt_disable();   // 设置不可抢占
        mutex_acquire_nest(&lock->dep_map, subclass, 0, nest_lock, ip); // 检查 mutex 需要的
条件
```

```
        if (__mutex_trylock(lock) ||                                    // 尝试上 lock
            mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, NULL)) {    // 尝试上乐观锁
                /* got the lock, yay! */
                lock_acquired(&lock->dep_map, ip);                      // 上 lock
                if (use_ww_ctx && ww_ctx)                               // ww_mutex_lock 时
                        ww_mutex_set_context_fastpath(ww, ww_ctx);      // 设置上下文 path
                preempt_enable();                                       // 解除不可抢占
                return 0;
        }
        spin_lock(&lock->wait_lock); // 对等待队列上自旋锁
        /*
         * After waiting to acquire the wait_lock, try again.
         */
        if (__mutex_trylock(lock)) {                                    // 那再试试呗 hhh
                if (use_ww_ctx && ww_ctx)
                        __ww_mutex_check_waiters(lock, ww_ctx);

                goto skip_wait;
        }

        debug_mutex_lock_common(lock, &waiter); // 掉一下 debug 模式下
mutet_lock_common

        lock_contended(&lock->dep_map, ip);     // 去等锁

        if (!use_ww_ctx) {                      // mutex_lock 时候
                /* add waiting tasks to the end of the waitqueue (FIFO): */
                __mutex_add_waiter(lock, &waiter, &lock->wait_list); // 加到 wait_queue


#ifdef CONFIG_DEBUG_MUTEXES
                waiter.ww_ctx = MUTEX_POISON_WW_CTX;
#endif
        } else {
                /*
                 * Add in stamp order, waking up waiters that must kill
                 * themselves.
                 */
                ret = __ww_mutex_add_waiter(&waiter, lock, ww_ctx); // 加到 ww_mutex 的
wait_queue
                if (ret)
                        goto err_early_kill;

                waiter.ww_ctx = ww_ctx;
```

```
        }

        waiter.task = current;

        set_current_state(state);   // 设置 state
        for (;;) {                  // 做了一个自旋操作  retry lock
                /*
                 * Once we hold wait_lock, we're serialized against
                 * mutex_unlock() handing the lock off to us, do a trylock
                 * before testing the error conditions to make sure we pick up
                 * the handoff.
                 */
                if (__mutex_trylock(lock))   // 等到了
                        goto acquired;

                /*
                 * Check for signals and kill conditions while holding
                 * wait_lock. This ensures the lock cancellation is ordered
                 * against mutex_unlock() and wake-ups do not go missing.
                 */
                if (unlikely(signal_pending_state(state, current))) { // if state 不对
                        ret = -EINTR;
                        goto err;
                }

                if (use_ww_ctx && ww_ctx) {   // 如果是 ww_mutex 且 wait_queue 有需要被 kill
掉的
                        ret = __ww_mutex_check_kill(lock, &waiter, ww_ctx);
                        if (ret)
                                goto err;
                }

                spin_unlock(&lock->wait_lock); // 解自旋锁
                schedule_preempt_disabled();     // 解除不可抢占

                /*
                 * ww_mutex needs to always recheck its position since its waiter
                 * list is not FIFO ordered.
                 */
                if ((use_ww_ctx && ww_ctx) || !first) {
                        first = __mutex_waiter_is_first(lock, &waiter);
                        if (first)
                                __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
                }
```

```
                set_current_state(state); // update state
                /*
                 * Here we order against unlock; we must either see it change
                 * state back to RUNNING and fall through the next schedule(),
                 * or we must see its unlock and acquire.
                 */
                if (__mutex_trylock(lock) || // 再试一次
                    (first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, &waiter)))
                        break;

                spin_lock(&lock->wait_lock);
        }
        spin_lock(&lock->wait_lock);
acquired:
        __set_current_state(TASK_RUNNING);

        if (use_ww_ctx && ww_ctx) {
                /*
                 * Wound-Wait; we stole the lock (!first_waiter), check the
                 * waiters as anyone might want to wound us.
                 */
                if (!ww_ctx->is_wait_die &&
                    !__mutex_waiter_is_first(lock, &waiter))
                        __ww_mutex_check_waiters(lock, ww_ctx);
        }

        mutex_remove_waiter(lock, &waiter, current); // 从等待队列中 remove
        if (likely(list_empty(&lock->wait_list)))
                __mutex_clear_flag(lock, MUTEX_FLAGS); // 清除 flag

        debug_mutex_free_waiter(&waiter);

skip_wait:
        /* got the lock - cleanup and rejoice! */
        lock_acquired(&lock->dep_map, ip);

        if (use_ww_ctx && ww_ctx)
                ww_mutex_lock_acquired(ww, ww_ctx);

        spin_unlock(&lock->wait_lock); // cleanup
        preempt_enable();
        return 0;
```

```
err:
        __set_current_state(TASK_RUNNING);
        mutex_remove_waiter(lock, &waiter, current);
err_early_kill:
        spin_unlock(&lock->wait_lock);
        debug_mutex_free_waiter(&waiter);
        mutex_release(&lock->dep_map, 1, ip);
        preempt_enable();
        return ret;
}
```

上面的__mutex_common 被 mutex_lock，ww_mutex_lock 两个函数复用，其中加了多次自旋锁尝试加锁。

- 读写锁 rwlock
- 条件变量 Condition
- 自旋锁 spinlock

Spinlock 相关代码在<include/linux/spinlock_api_smp.h>中

```
static inline int __raw_spin_trylock(raw_spinlock_t *lock)
{
        preempt_disable(); // 设置不可抢占
        if (do_raw_spin_trylock(lock)) {   // 尝试获得自旋锁
                spin_acquire(&lock->dep_map, 0, 1, _RET_IP_); // 获得自旋锁
                return 1;
        }
        preempt_enable(); // 接触不可抢占
        return 0;
}
```

其中 spin_acquire 定义在<include/linux/lockdep.h>

```
#define spin_acquire(l, s, t, i)                lock_acquire_exclusive(l, s, t, NULL, i)
#define lock_acquire_exclusive(l, s, t, n, i)           lock_acquire(l, s, t, 0, 1, n, i)
```

而 lock_acquire()实现的代码在<kernel/locking/lockdep.c>

```
void lock_acquire(struct lockdep_map *lock, unsigned int subclass,
                        int trylock, int read, int check,
                        struct lockdep_map *nest_lock, unsigned long ip)
{
        unsigned long flags;

        if (unlikely(current->lockdep_recursion)) // 如果锁的递归深度标志位!=0
                return;

        raw_local_irq_save(flags); // 刷一下 flags 到 disk
        check_flags(flags); // 检查 flag

        current->lockdep_recursion = 1; // 互斥
```

```
        trace_lock_acquire(lock, subclass, trylock, read, check, nest_lock, ip); // 追踪锁获得 打印
日志
        __lock_acquire(lock, subclass, trylock, read, check,
                       irqs_disabled_flags(flags), nest_lock, ip, 0, 0);    // lock acquire
        current->lockdep_recursion = 0; // 解除互斥
        raw_local_irq_restore(flags); // 再刷一下 flags
}
EXPORT_SYMBOL_GPL(lock_acquire);
static int __lock_acquire(struct lockdep_map *lock, unsigned int subclass,
                          int trylock, int read, int check, int hardirqs_off,
                          struct lockdep_map *nest_lock, unsigned long ip,
                          int references, int pin_count)
{
        struct task_struct *curr = current;
        struct lock_class *class = NULL;
        struct held_lock *hlock;
        unsigned int depth;
        int chain_head = 0;
        int class_idx;
        u64 chain_key;

        if (subclass < NR_LOCKDEP_CACHING_CLASSES)
                class = lock->class_cache[subclass]; // 找到 cache
        /*
         * Not cached?
         */
        if (unlikely(!class)) {
                class = register_lock_class(lock, subclass, 0); // 注册 lock
                if (!class)
                        return 0;
        }
        atomic_inc((atomic_t *)&class->ops); // 原子操作获得 class 操作符
        if (very_verbose(class)) {
                printk("\nacquire class [%px] %s", class->key, class->name);
                if (class->name_version > 1)
                        printk(KERN_CONT "#%d", class->name_version);
                printk(KERN_CONT "\n");
                dump_stack();
        }
        depth = curr->lockdep_depth;    // init depth

        if (DEBUG_LOCKS_WARN_ON(depth >= MAX_LOCK_DEPTH)) // stack 深度溢出
                return 0;
```

```
	class_idx = class - lock_classes + 1;

	if (depth) {
		hlock = curr->held_locks + depth - 1;
		if (hlock->class_idx == class_idx && nest_lock) {
			if (hlock->references) {
				/*
				 * Check: unsigned int references:12, overflow.
				 */
				if (DEBUG_LOCKS_WARN_ON(hlock->references == (1 << 12)-
1)) // 2^12 - 1
					return 0;

				hlock->references++;
			} else {
				hlock->references = 2;
			}

			return 1;
		}
	}

	hlock = curr->held_locks + depth;
	if (DEBUG_LOCKS_WARN_ON(!class))
		return 0;
	hlock->class_idx = class_idx; // 记录 hlock 信息
	hlock->acquire_ip = ip;
	hlock->instance = lock;
	hlock->nest_lock = nest_lock;
	hlock->irq_context = task_irq_context(curr);
	hlock->trylock = trylock;
	hlock->read = read;
	hlock->check = check;
	hlock->hardirqs_off = !!hardirqs_off;
	hlock->references = references;
#ifdef CONFIG_LOCK_STAT
	hlock->waittime_stamp = 0;
	hlock->holdtime_stamp = lockstat_clock();
#endif
	hlock->pin_count = pin_count;

	if (check && !mark_irqflags(curr, hlock))
		return 0;
```

```
        /* mark it as used: */
        if (!mark_lock(curr, hlock, LOCK_USED))
                return 0;

        if (DEBUG_LOCKS_WARN_ON(class_idx > MAX_LOCKDEP_KEYS)) // 又溢出了
                return 0;

        chain_key = curr->curr_chain_key;
        if (!depth) {
                /*
                 * How can we have a chain hash when we ain't got no keys?!
                 */
                if (DEBUG_LOCKS_WARN_ON(chain_key != 0))
                        return 0;
                chain_head = 1;
        }

        hlock->prev_chain_key = chain_key;
        if (separate_irq_context(curr, hlock)) {
                chain_key = 0;
                chain_head = 1;
       }
        chain_key = iterate_chain_key(chain_key, class_idx);
        curr->curr_chain_key = chain_key;
        curr->lockdep_depth++;
        check_chain_key(curr);
        return 1;
}
```

__lock_acquire()函数完成对锁类的加锁，在 linux 里这个函数运用在 mutex 和 spinlock 的加锁中，主要的逻辑用来打印错误日志。

- 屏障 barrier

Exercise2

threads/synch.h

在 synch.h 中声明了 nachos 关于线程间调度的三种方式 Semaphore, Lock, Condition。定义了主要 private 变量和 public 方法。

threads/synch.cc

synch.cc 主要实现 synch.h 中一些较为复杂的 public 方法。

其中 class Semaphore 中的函数已经被实现了，Lock，Condition 的实现是 Experience3 的任务。

Semaphore 的实现相当于一个例子，

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

    while (value == 0) {                      // 等 Value 值
      queue->Append((void*)currentThread);  // 如果有新的线程进来了
      currentThread->Sleep();               // 做 sleep 操作
    }
    value--;                                // 如果可用了减去一个 value,

    (void)interrupt->SetLevel(oldLevel);  // 恢复中断
                                                                    }
```

从上述代码中，可以看出关于线程间控制的代码通过控制中断实现，依次类似的，信号量的释放，Lock 的获取、释放，条件变量的 wait，wakeup 都是通过控制中断来实现的。

具体而言，需要实现以下方法：

```
void Semaphore::P() {} // 信号量的获取
void Semaphore::V() {} // 信号量的释放
void Lock::Acquire() {}  // 锁的获取
void Lock:: Release () {} // 锁的释放
void Condition::Wait(Lock* conditionLock) {}     // 条件变量的获取
void Condition::Signal(Lock* conditionLock) {}    // 条件变量单线程唤醒
void Condition::Broadcast(Lock* conditionLock) {} // 条件变量广播唤醒线程
```

threads/synchlist.h

在 **synchlist**.h 中构造了一个线程安全的 List 叫做 sysnchList。其中通过 synch.h 声明的 Lock，Condition 实现线程安全。

threads/synchlist.cc

实现 synchlist.cc 声明的复杂方法。具体实现与常见的同步互斥问题相同。

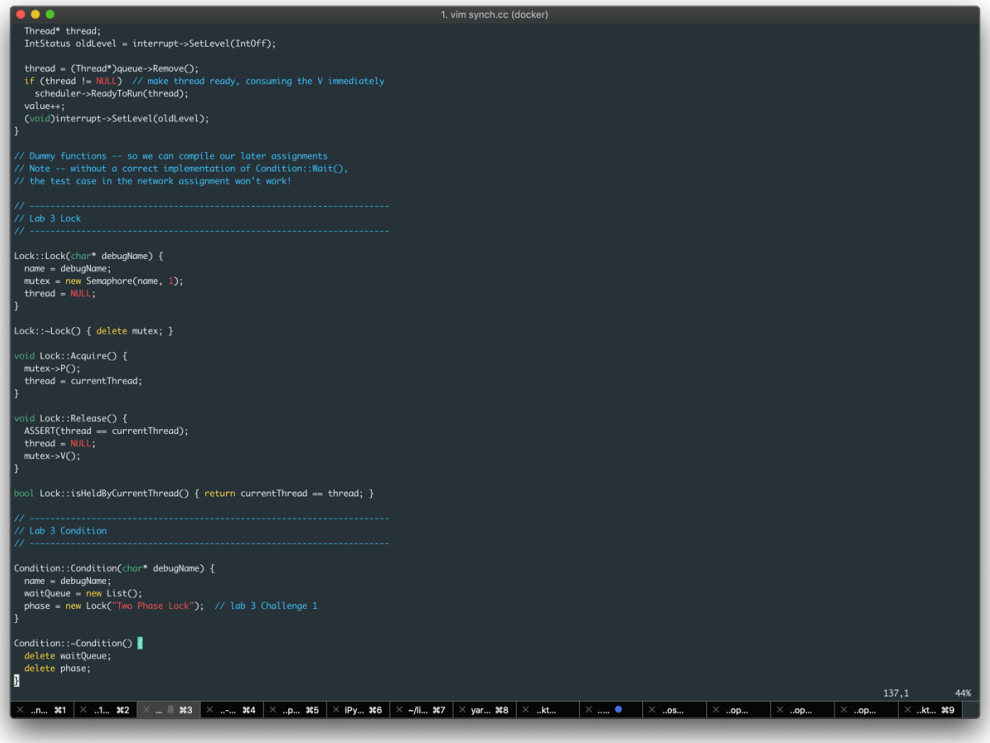
具体而言需要实现：

```
void SynchList::Append(void *item) {} // 线程安全的 append
void *SynchList::Remove() {}
void SynchList::Mapcar(VoidFunctionPtr func) {}
```

Exercise3

Exercise3 要求实现锁和条件变量，仿照 Semaphore 中的操作。Lock 的实现相对而言简单一点，只需要完成 Semaphore 的调用就行了，通过设置一个互斥信号量来完成。

Condition 的实现稍微复杂一点，设置一个互斥资源锁 Lock，通过把当前 currentThread 塞到一个 Queue 中，实现 wait。如果要唤醒就从 Queue 中拿出来，就行遍历。



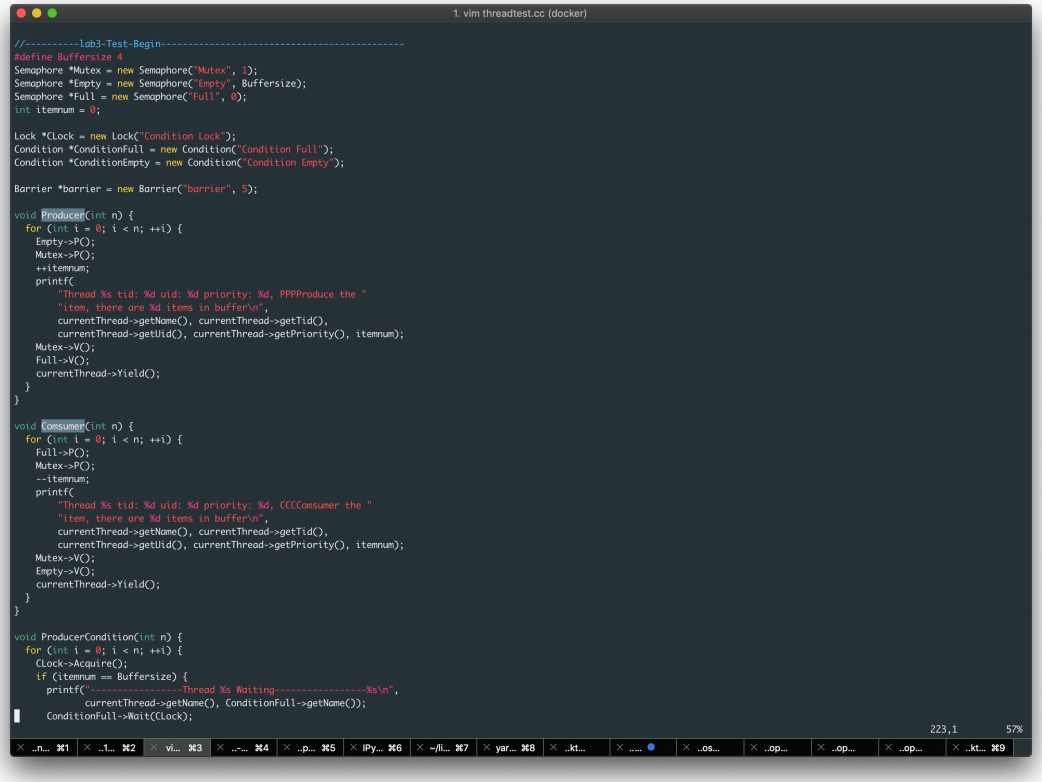
Exercise 4

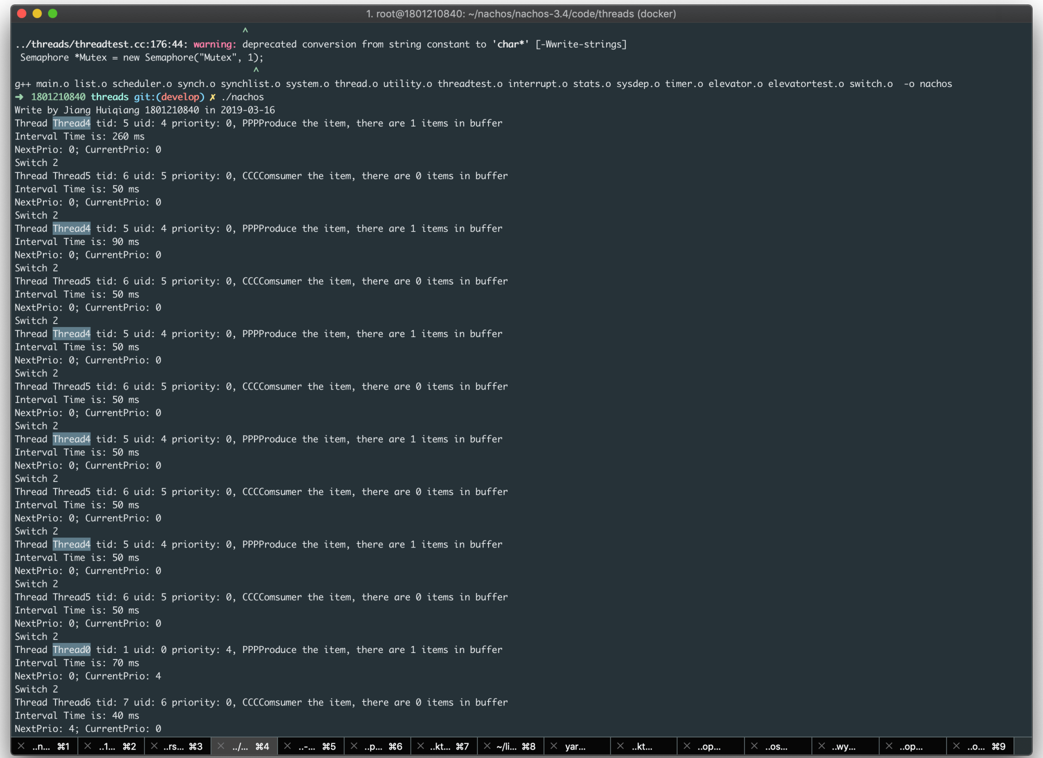
通过上面的 Exercise 3 中实现的信号量，Lock，Condition 来完成同步互斥机制。同步互斥机制问题选择“生产者-消费者”来进行分析。

① 信号量

通过定义三个信号量，来实现生产者和消费者。每次消费之前先去 Full，Mutex 中看看有没有生产好的商品。生产之前去 Empty 有没有空闲的位置。

```
#define Buffersize 4
Semaphore *Mutex = new Semaphore("Mutex", 1);
Semaphore *Empty = new Semaphore("Empty", Buffersize);
Semaphore *Full = new Semaphore("Full", 0);
int itemnum = 0;
```

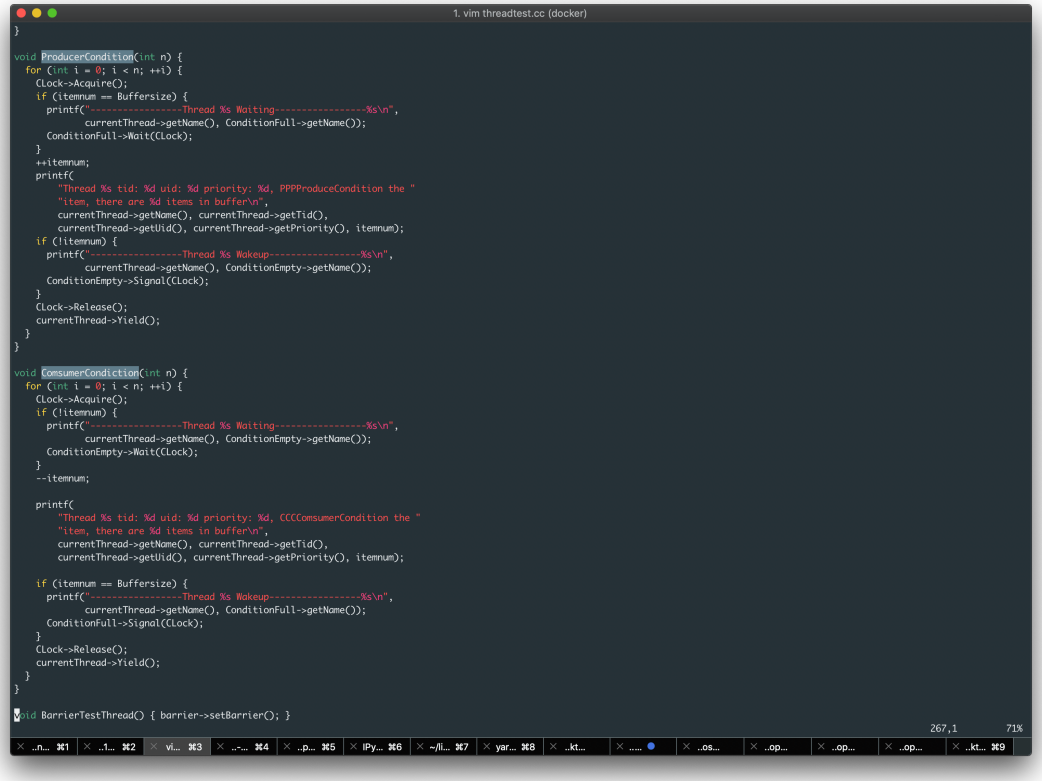


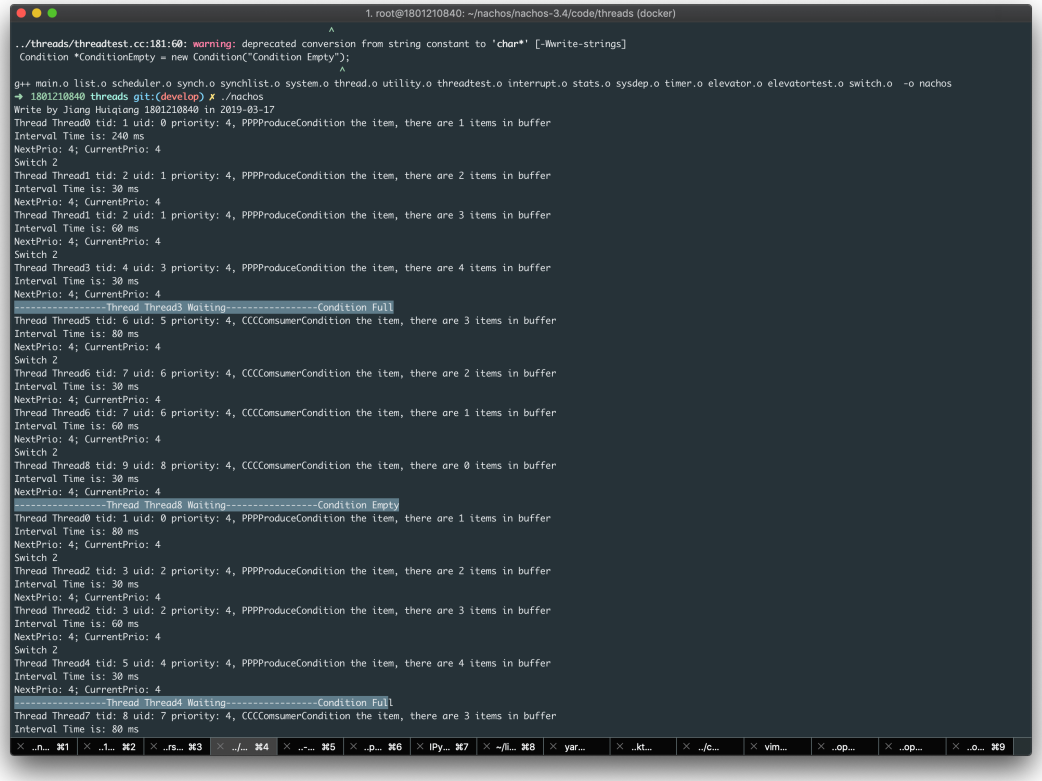


② 条件变量

实现思路基本上和信号量的方案一致。当生产的商品达到最大限额的时候，通过 Condition 进行 Wait 操作，直到被有资源线程被唤醒。

```
Lock *CLock = new Lock("Condition Lock");
Condition *ConditionFull = new Condition("Condition Full");
Condition *ConditionEmpty = new Condition ("Condition Empty");
```


1. vim threadtest.cc (docker)
}
void ProducerCondition(int n) {
for (int i = 0; i < n; ++i) {
CLock->Acquire();
if (itemnum == Buffersize) {
printf("----------------Thread %s Waiting----------------%s\n",
currentThread->getName(), ConditionFull->getName());
ConditionFull->Wait(CLock);
}
++itemnum;
printf(
"Thread %s tid: %d uid: %d priority: %d, PPPProduceCondition the "
"item, there are %d items in buffer\n",
currentThread->getName(), currentThread->getTid(),
currentThread->getUid(), currentThread->getPriority(), itemnum);
if (!itemnum) {
printf("----------------Thread %s Wakeup----------------%s\n",
currentThread->getName(), ConditionEmpty->getName());
ConditionEmpty->Signal(CLock);
}
CLock->Release();
currentThread->Yield();
}
}
void ComsumerCondiction(int n) {
for (int i = 0; i < n; ++i) {
CLock->Acquire();
if (!itemnum) {
printf("----------------Thread %s Waiting----------------%s\n",
currentThread->getName(), ConditionEmpty->getName());
ConditionEmpty->Wait(CLock);
}
--itemnum;
printf(
"Thread %s tid: %d uid: %d priority: %d, CCCComsumerCondition the "
"item, there are %d items in buffer\n",
currentThread->getName(), currentThread->getTid(),
currentThread->getUid(), currentThread->getPriority(), itemnum);
if (itemnum == Buffersize) {
printf("----------------Thread %s Wakeup----------------%s\n",
currentThread->getName(), ConditionFull->getName());
ConditionFull->Signal(CLock);
}
CLock->Release();
currentThread->Yield();
}
}
void BarrierTestThread() { barrier->setBarrier(); }
267,1 71%


1. root@1801210840: ~/nachos/nachos-3.4/code/threads (docker)
../threads/threadtest.cc:181:60: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
Condition *ConditionEmpty = new Condition("Condition Empty");
g++ main.o list.o scheduler.o synch.o synchlist.o system.o thread.o utility.o threadtest.o interrupt.o stats.o sysdep.o timer.o elevator.o elevatortest.o switch.o -o nachos
➜ 1801210840 threads git:(develop) ✗ ./nachos
Write by Jiang Huiqiang 1801210840 in 2019-03-17
Thread Thread0 tid: 1 uid: 0 priority: 4, PPPProduceCondition the item, there are 1 items in buffer
Interval Time is: 240 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
Thread Thread1 tid: 2 uid: 1 priority: 4, PPPProduceCondition the item, there are 2 items in buffer
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
Thread Thread1 tid: 2 uid: 1 priority: 4, PPPProduceCondition the item, there are 3 items in buffer
Interval Time is: 60 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
Thread Thread3 tid: 4 uid: 3 priority: 4, PPPProduceCondition the item, there are 4 items in buffer
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
----------------Thread Thread3 Waiting----------------Condition Full
Thread Thread5 tid: 6 uid: 5 priority: 4, CCCComsumerCondition the item, there are 3 items in buffer
Interval Time is: 80 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
Thread Thread6 tid: 7 uid: 6 priority: 4, CCCComsumerCondition the item, there are 2 items in buffer
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
Thread Thread6 tid: 7 uid: 6 priority: 4, CCCComsumerCondition the item, there are 1 items in buffer
Interval Time is: 60 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
Thread Thread8 tid: 9 uid: 8 priority: 4, CCCComsumerCondition the item, there are 0 items in buffer
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
----------------Thread Thread8 Waiting----------------Condition Empty
Thread Thread0 tid: 1 uid: 0 priority: 4, PPPProduceCondition the item, there are 1 items in buffer
Interval Time is: 80 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
Thread Thread2 tid: 3 uid: 2 priority: 4, PPPProduceCondition the item, there are 2 items in buffer
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
Thread Thread2 tid: 3 uid: 2 priority: 4, PPPProduceCondition the item, there are 3 items in buffer
Interval Time is: 60 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
Thread Thread4 tid: 5 uid: 4 priority: 4, PPPProduceCondition the item, there are 4 items in buffer
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
----------------Thread Thread4 Waiting----------------Condition Full
Thread Thread7 tid: 8 uid: 7 priority: 4, CCCComsumerCondition the item, there are 3 items in buffer
Interval Time is: 80 ms

Challenge 1

Challenge1 要求通过信号量等线程同步机制，完成一个 barrier 的功能。即线程在一定数量前，等待，直到达到某一特定数量，再一起执行。

如果只是实现一个 barrier 还是比较简单的。设置一个 resourceNum，一个 Lock， 一个条件变量 Condition。每次有 Thread 进来的时候，先去抢锁，如果还有资源，则进到条件变量 Condition 的 WaitQueue 中。如果资源被消耗完了，说明已经达到一定数量了。这时候原本积累在条件变量 Condition 的 WaitQueue 中的 Threads 就需要一起被唤醒。

但上面的代码有很大的 bug，就线程数必须和 Barrier 要求的线程数完全一致。这个在实际使用上是不能接受的。故想对代码进行优化，使其能实现两阶段锁协议的功能。

实现两阶段锁协议，代码就比刚才要复杂多了。通过维护一个 Lock 锁来实现阶段的拦截。如果进入到 consumer 阶段，则进程进入的时候需要先去获取这个锁。当 Barrier 拦截的线程 Queue 被访问完了的时候，这个锁就被释放，那么新的 Thread 就可以进入，从而到下一个 Queue 阶段。

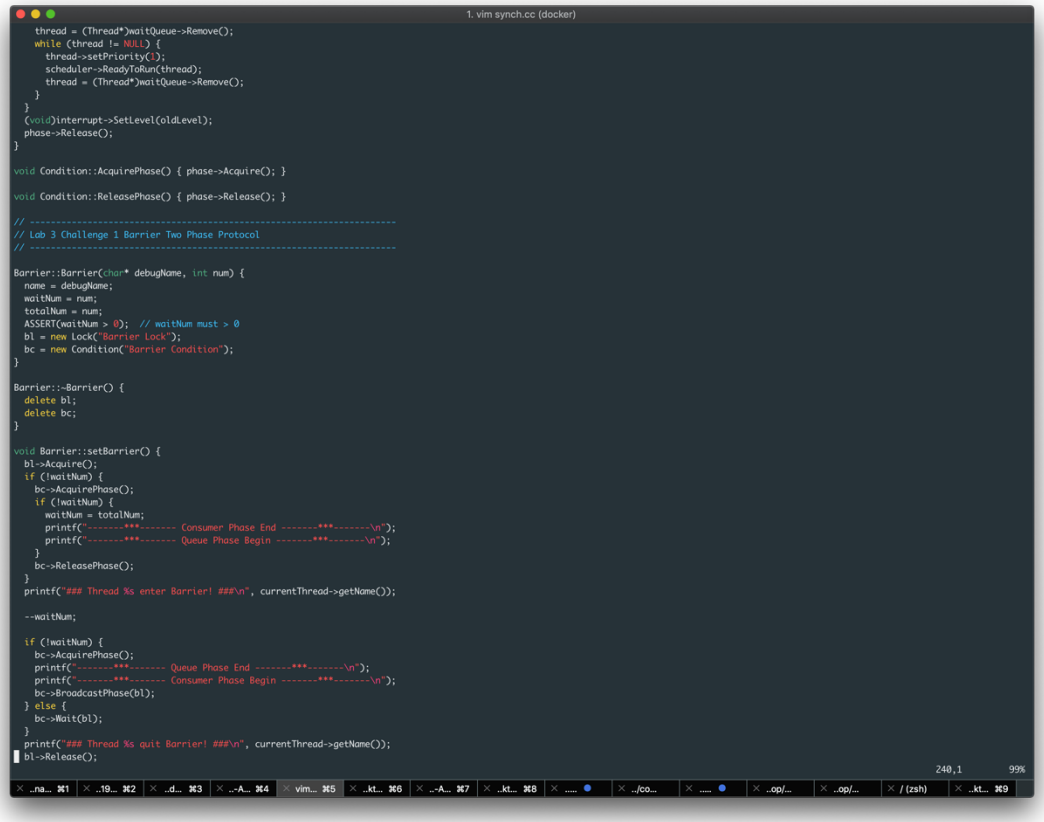
那么这个时候就出现一个问题，我 Barrier 之前实现的 Queue 释放，实际上调用的是 Condition 的 Broadcast 方法，至于什么时候广播结束，Barrier 实际上是不知道的。两种思路，一个是加一个回调函数，另外一个就是改 Condition Class。这里通过修改 Condition Class 来实现这个功能。通过在 Condition 中加一个 Phase Lock，然后暴露出去 AcquirePahse(), ReleasePhase()两个 public 方法，来实现两阶段锁协议。当 Broadcast 释放完 Queue 之后，下面的进程才能进入。

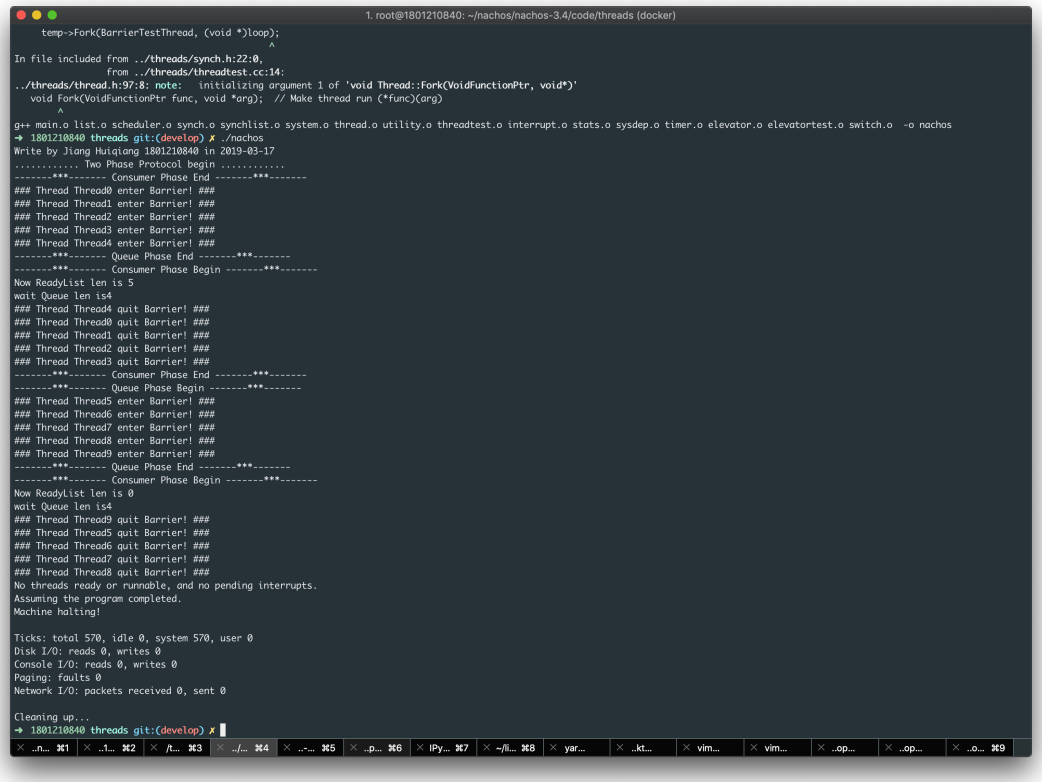
这样设计完之后，又会出现一个问题，Condition 的广播唤醒实际上做的是把 Queue 中的 Thread 依次放入 Schedule 的 ReadyToRunlList 中，至于什么时候被调用，Condition 也不知道。

于是想能不能去查一下 Schedule 的 ReadyToRun List 里面还有多少变量，如果变量数字==之前的那个数字，说明 Queue 里面的 Thread 已经被处理完了。但想法是好的，仿照这 Semaphore 的 PV 进行设计，弄一个 While 循环来监听 ReadyToRun，但这样做就会把 currentThread 堵死。尝试了几种策略包括 sleep()，等等。

实践出来 bug 不断，包括虽然加 Phase 锁，但第一轮的 Thread 比第二轮的线程还晚结束。

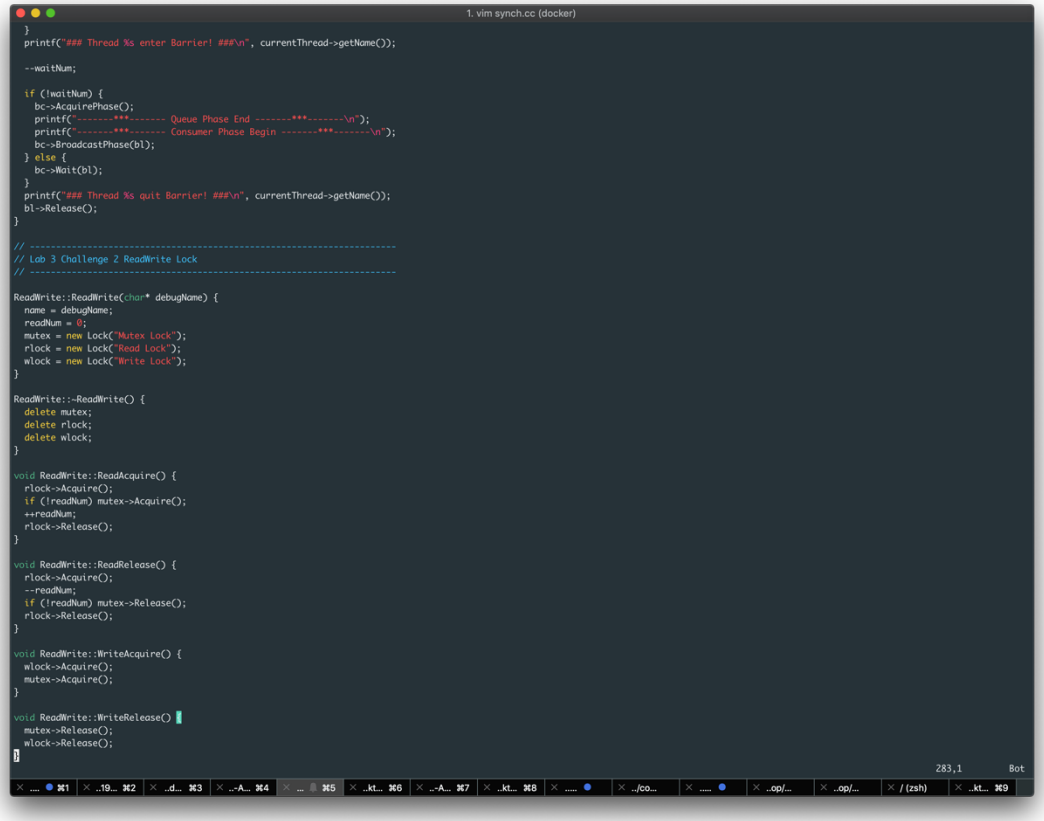
最后，给出了设置高权限的策略。在 Queue 中的 Thread 应该是高优先级的，应该是优先完成的。高优先级别的好处，还有放在 ReadyToRun 的前面，当 Broadcast 线程结束之后，下一个轮到的应该是原来 WaitQueue 中的进程，而不是还没进入 Queue 阶段的进程，这样就避免了第一轮 Thread 比第二轮进程还晚结束的情况。也算是完成了原来设计的两阶段锁结构的设计。

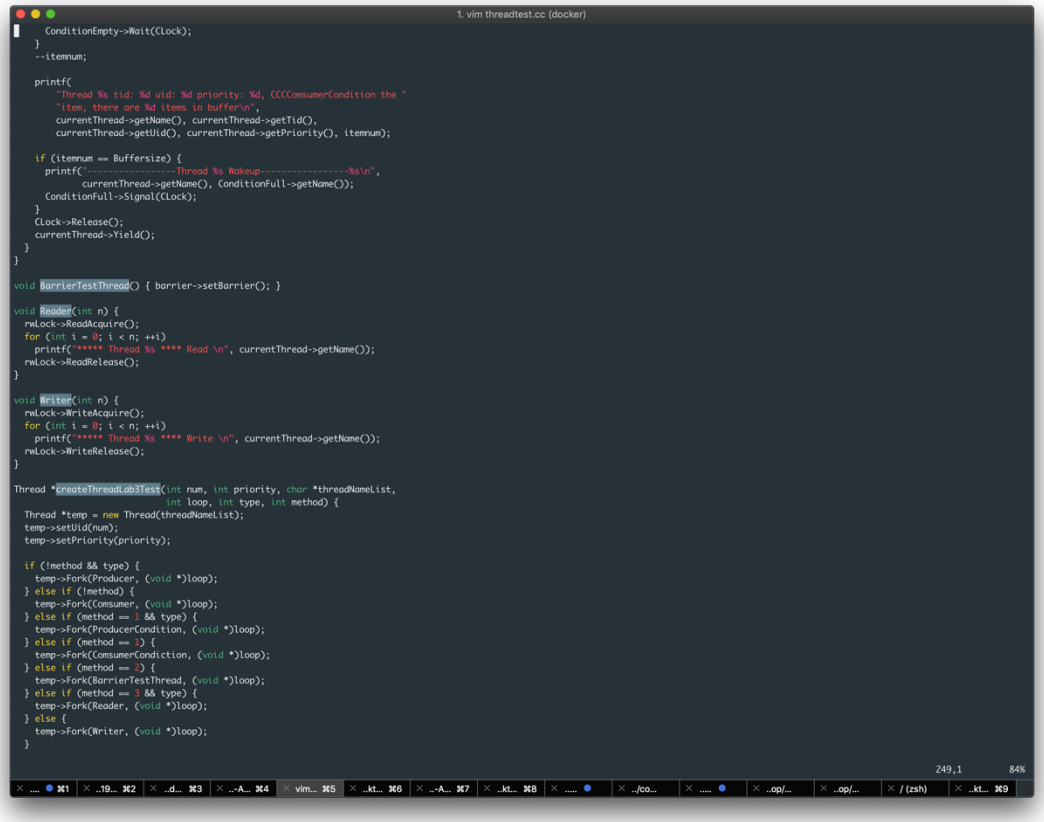

1. vim synch.cc (docker)
thread = (Thread*)waitQueue->Remove();
while (thread != NULL) {
thread->setPriority(1);
scheduler->ReadyToRun(thread);
thread = (Thread*)waitQueue->Remove();
}
}
(void)interrupt->SetLevel(oldLevel);
phase->Release();
}
void Condition::AcquirePhase() { phase->Acquire(); }
void Condition::ReleasePhase() { phase->Release(); }
// --
// Lab 3 Challenge 1 Barrier Two Phase Protocol
// --
Barrier::Barrier(char* debugName, int num) {
name = debugName;
waitNum = num;
totalNum = num;
ASSERT(waitNum > 0); // waitNum must > 0
bl = new Lock("Barrier Lock");
bc = new Condition("Barrier Condition");
}
Barrier::~Barrier() {
delete bl;
delete bc;
}
void Barrier::setBarrier() {
bl->Acquire();
if (!waitNum) {
bc->AcquirePhase();
if (!waitNum) {
waitNum = totalNum;
printf("-------***------- Consumer Phase End -------***-------\n");
printf("-------***------- Queue Phase Begin -------***-------\n");
}
bc->ReleasePhase();
}
printf("### Thread %s enter Barrier! ###\n", currentThread->getName());
--waitNum;
if (!waitNum) {
bc->AcquirePhase();
printf("-------***------- Queue Phase End -------***-------\n");
printf("-------***------- Consumer Phase Begin -------***-------\n");
bc->BroadcastPhase(bl);
} else {
bc->Wait(bl);
}
printf("### Thread %s quit Barrier! ###\n", currentThread->getName());
bl->Release();
240,1 99%

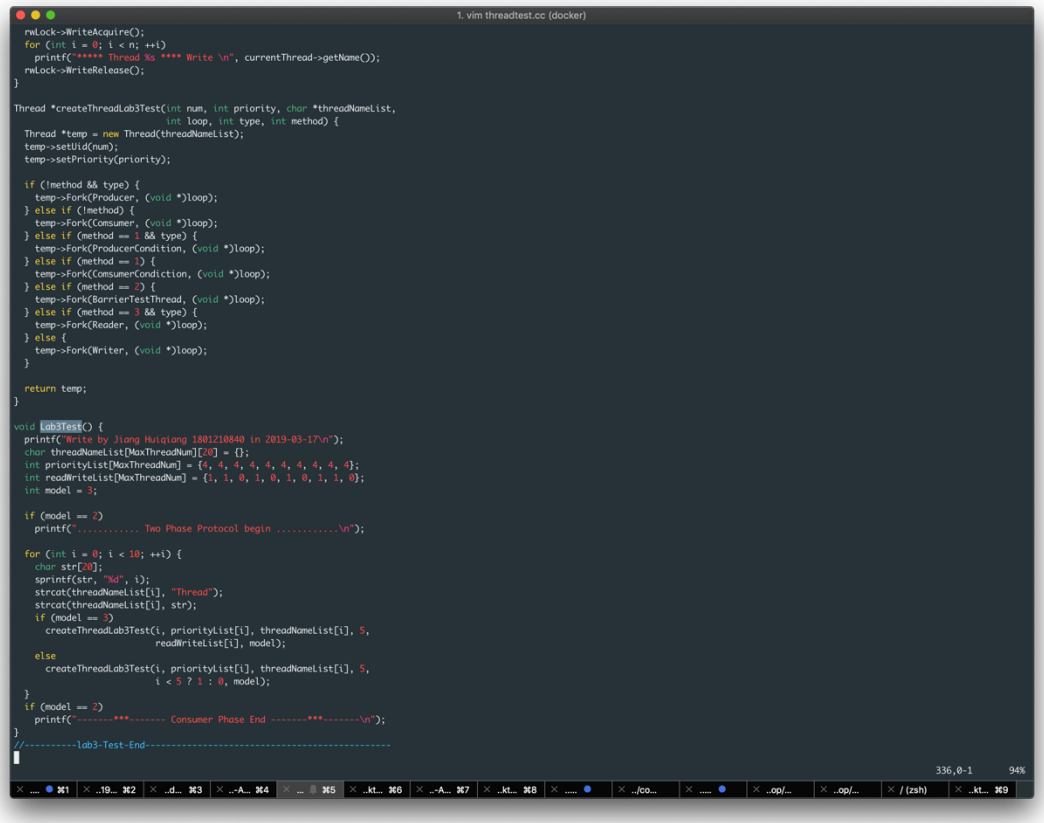

1. root@1801210840: ~/nachos/nachos-3.4/code/threads (docker)
temp->Fork(BarrierTestThread, (void *)loop);
^
In file included from ../threads/synch.h:22:0,
from ../threads/threadtest.cc:14:
../threads/thread.h:97:8: note: initializing argument 1 of 'void Thread::Fork(VoidFunctionPtr, void*)'
void Fork(VoidFunctionPtr func, void *arg); // Make thread run (*func)(arg)
^
g++ main.o list.o scheduler.o synch.o synchlist.o system.o thread.o utility.o threadtest.o interrupt.o stats.o sysdep.o timer.o elevator.o elevatortest.o switch.o -o nachos
➜ 1801210840 threads git:(develop) ✗ ./nachos
Write by Jiang Huiqiang 1801210840 in 2019-03-17
---------- Two Phase Protocol begin ----------
-------***------- Consumer Phase End -------***-------
Thread Thread0 enter Barrier!
Thread Thread1 enter Barrier!
Thread Thread2 enter Barrier!
Thread Thread3 enter Barrier!
Thread Thread4 enter Barrier!
-------***------- Queue Phase End -------***-------
-------***------- Consumer Phase Begin -------***-------
Now ReadyList len is 5
wait Queue len is4
Thread Thread4 quit Barrier!
Thread Thread0 quit Barrier!
Thread Thread1 quit Barrier!
Thread Thread2 quit Barrier!
Thread Thread3 quit Barrier!
-------***------- Consumer Phase End -------***-------
-------***------- Queue Phase Begin -------***-------
Thread Thread5 enter Barrier!
Thread Thread6 enter Barrier!
Thread Thread7 enter Barrier!
Thread Thread8 enter Barrier!
Thread Thread9 enter Barrier!
-------***------- Queue Phase End -------***-------
-------***------- Consumer Phase Begin -------***-------
Now ReadyList len is 0
wait Queue len is4
Thread Thread9 quit Barrier!
Thread Thread5 quit Barrier!
Thread Thread6 quit Barrier!
Thread Thread7 quit Barrier!
Thread Thread8 quit Barrier!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
Ticks: total 570, idle 0, system 570, 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...
➜ 1801210840 threads git:(develop) ✗

Challenge 2

利用前面的 Lock 实现读者-写者锁。我选用了读者优先的策略。维护了一个全局锁 mutex，一个读锁 rlock，一个写锁 wlock。支持读者并发读，只有没有读者的时候，写者才能进行写操作。




1. vim threadtest.cc (docker)
ConditionEmpty->Wait(CLock);
}
--itemnum;
printf(
"Thread %s tid: %d uid: %d priority: %d, CCCComsumerCondition the "
"item, there are %d items in buffer\n",
currentThread->getName(), currentThread->getTid(),
currentThread->getUid(), currentThread->getPriority(), itemnum);
if (itemnum == Buffersize) {
printf("----------------Thread %s Wakeup----------------%s\n",
currentThread->getName(), ConditionFull->getName());
ConditionFull->Signal(CLock);
}
CLock->Release();
currentThread->Yield();
}
}
void BarrierTestThread() { barrier->setBarrier(); }
void Reader(int n) {
rwLock->ReadAcquire();
for (int i = 0; i < n; ++i)
printf("***** Thread %s **** Read \n", currentThread->getName());
rwLock->ReadRelease();
}
void Writer(int n) {
rwLock->WriteAcquire();
for (int i = 0; i < n; ++i)
printf("***** Thread %s **** Write \n", currentThread->getName());
rwLock->WriteRelease();
}
Thread *createThreadLab3Test(int num, int priority, char *threadNameList,
int loop, int type, int method) {
Thread *temp = new Thread(threadNameList);
temp->setUid(num);
temp->setPriority(priority);
if (!method && type) {
temp->Fork(Producer, (void *)loop);
} else if (!method) {
temp->Fork(Comsumer, (void *)loop);
} else if (method == 1 && type) {
temp->Fork(ProducerCondition, (void *)loop);
} else if (method == 1) {
temp->Fork(ComsumerCondiction, (void *)loop);
} else if (method == 2) {
temp->Fork(BarrierTestThread, (void *)loop);
} else if (method == 3 && type) {
temp->Fork(Reader, (void *)loop);
} else {
temp->Fork(CWriter, (void *)loop);
}
249,1 84%
⌘1 ⌘2 ⌘3 ⌘4 vim... ⌘5 ⌘6 ⌘7 ⌘8 ⌘9


1. vim threadtest.cc (docker)
rwLock->WriteAcquire();
for (int i = 0; i < n; ++i)
printf("***** Thread %s **** Write \n", currentThread->getName());
rwLock->WriteRelease();
}
Thread *createThreadLab3Test(int num, int priority, char *threadNameList,
int loop, int type, int method) {
Thread *temp = new Thread(threadNameList);
temp->setUid(num);
temp->setPriority(priority);
if (!method && type) {
temp->Fork(Producer, (void *)loop);
} else if (!method) {
temp->Fork(Comsumer, (void *)loop);
} else if (method == 1 && type) {
temp->Fork(ProducerCondition, (void *)loop);
} else if (method == 1) {
temp->Fork(ComsumerCondiction, (void *)loop);
} else if (method == 2) {
temp->Fork(BarrierTestThread, (void *)loop);
} else if (method == 3 && type) {
temp->Fork(Reader, (void *)loop);
} else {
temp->Fork(CWriter, (void *)loop);
}
return temp;
}
void Lab3Test() {
printf("Write by Jiang Huiqiang 1801210840 in 2019-03-17\n");
char threadNameList[MaxThreadNum][20] = {};
int priorityList[MaxThreadNum] = {4, 4, 4, 4, 4, 4, 4, 4, 4, 4};
int readWriteList[MaxThreadNum] = {1, 1, 0, 1, 0, 1, 0, 1, 1, 0};
int model = 3;
if (model == 2)
printf("............ Two Phase Protocol begin ....\n");
for (int i = 0; i < 10; ++i) {
char str[20];
sprintf(str, "%d", i);
strcat(threadNameList[i], "Thread");
strcat(threadNameList[i], str);
if (model == 3)
createThreadLab3Test(i, priorityList[i], threadNameList[i], 5,
readWriteList[i], model);
else
createThreadLab3Test(i, priorityList[i], threadNameList[i], 5,
i < 5 ? 1 : 0, model);
}
if (model == 2)
printf("-------***------- Consumer Phase End -------***-------\n");
}
//----------lab3-Test-End----------
336,0-1 94%
⌘1 ⌘2 ⌘3 ⌘4 ⌘5 ⌘6 ⌘7 ⌘8 ⌘9

内容三：遇到的困难以及解决方法

遇到的困难主要是 Challenge 1 中实践时遇到的问题。一开始的方案只适合 BarrierNum == ThreadNum。一旦有线程数比较多的时候就不适用。于是改成两阶段锁协议，但遇到的问题更多。Phase Lock 应该放在哪个类。加了 Phase Lock 之后为啥第一轮的 Thread 比第二轮的还迟退出。最后给出了基于优先级的方案，完成了两阶段锁结构的实现。

内容四：收获及感想

这次的作业比 lab2 难度有所增加，做起来还是比较吃力的，尤其是 Challenge 1 的实现，还是很有意思的。通过这次实践，强化了自己对两阶段锁协议的理解，强化了自己对线程同步机制的理解。

内容五：对课程的意见和建议

感谢老师给我们这个机会动手体会 os 中线程间同步机制的具体逻辑。

内容六：参考文献

[1] Stevens, W. R. (2002). UNIX 环境高级编程：英文版. 机械工业出版社.
[2] RobertLove, et al. Linux 内核设计与实现. 机械工业出版社, 2006.
[3] Linux 内核调度分析（线程同步）