西安电子科技大学

操作系统课程设计

(2018 年度)

实验报告

实验名称：Priority-Inversion

班　　级：1603019

姓　　名：张俊华

学　　号：16030199025

一、实验内容

操作系统中存在优先级反转问题——当一个高优先级线程通过信号量机制访问共享资源时，该 信号量已被一低优先级任务占有，而这个低优先级任务在访问共享资源时，可能又被其他一些中等 优先级任务抢先，因此造成高优先级任务被许多低优先级任务阻塞，实时性难以保证。 **我们的任务是：解决由锁（Lock）造成的优先级反转问题，解决策略是优先级捐赠。**

二、分析与设计

优先级捐赠

所谓的优先级反转，出现在高低优先级线程对锁 （Lock）的竞争之中。为了避免高优先级任务被许多低优先级任务阻塞，就需要提高占有锁的进程的优先级，将高优先级线程的优先级赋予低优先级线程，就是题目要求的优先级捐赠。

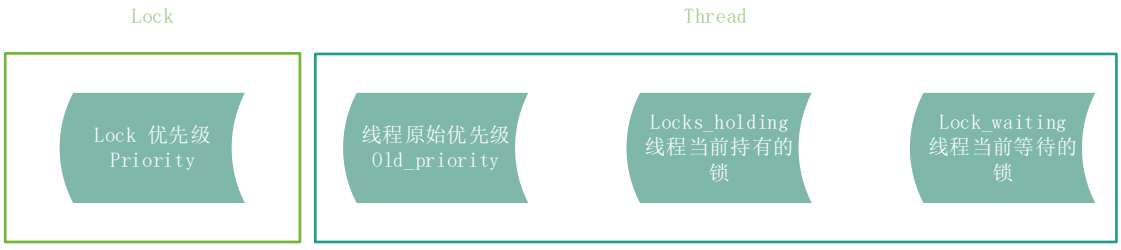
根据老师的提示，和检查 Pintos 的测试样例，可以看到优先级捐赠的典型情况有下面三种

仔细观察发生优先级捐赠的情况，可以意识到，若想完成优先级的捐赠，在两个线程之间，需要有一个用于交换优先级的媒介。自然地，锁（lock）作为两个进程之间都需要获取的目标，就可以承担起传递优先级的重任。

实现

要让锁来传递优先级，就需要对锁现有的数据结构进行改造。需要为锁引入优先级的概念。

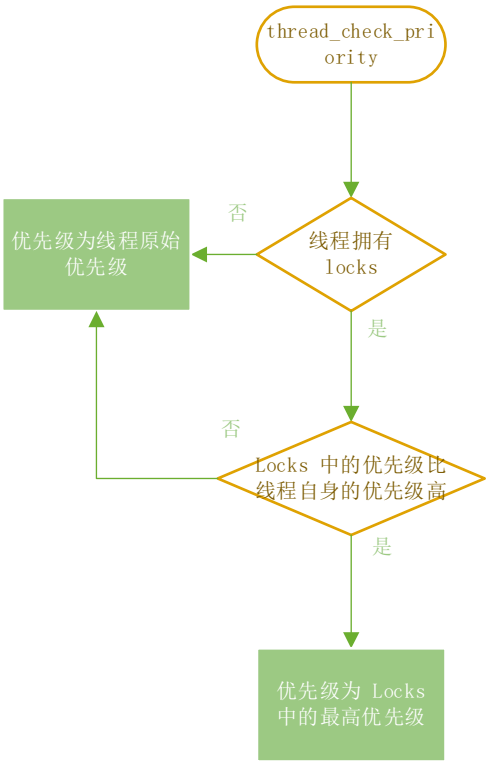
锁引入了优先级之后，要表示线程和锁之间的关系，还需要对线程的结构进行改造。同时，由于线程在优先级捐赠过程结束之后需要恢复原始优先级，因此还需要增加一个字段，记录线程原始的优先级。



在数据结构准备完毕之后，仔细观察三种捐赠类型，寻找优先级捐赠过程中不变的特征。可以发现，不管是多么复杂的捐赠过程，其核心之处在于，在线程的并发流程中，始终保持 **线**

程的优先级，和其所拥有的锁的最高优先级相同。 *（锁的优先级是指，占有该锁的线程中的最高优先级）。*

意识到这点之后，我们就可以绘制出优先级捐赠的核心实现流程：



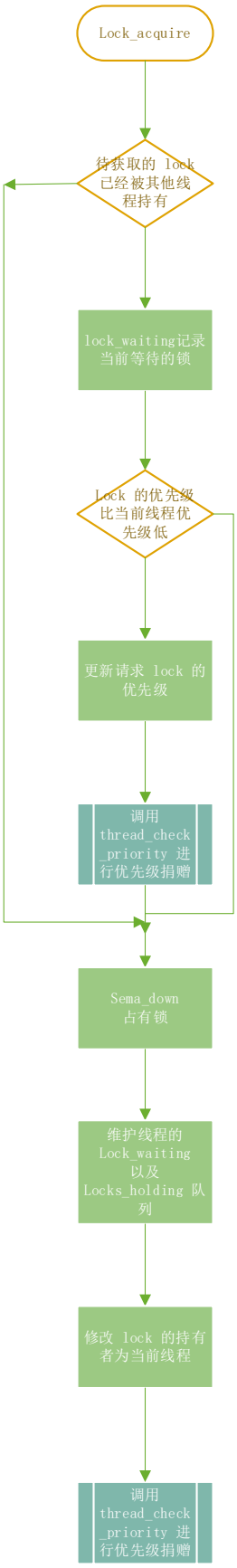
优先级捐赠的核心，就是比较线程的优先级和其占有锁的优先级。并根据比较结果，对线程自身的优先级进行实时调整。可以把这个比较过程封装成函数 `thread_check_priority`，于是，整个优先级捐赠的关键，也是最复杂的部分，就是在于调用 `thread_check_priority`，完成捐赠的时机。

调用 `thread_check_priority` 的时机

由于之前分析的，线程的优先级和其所持有锁的优先级时时相关。因此，自然地，**调用 `thread_check_priority` 的时机就在线程的优先级或其持有锁的优先级改变之时**。

什么时候锁的优先级会被改变？——当其他优先级的线程也对同一个锁进行了请求的时候。

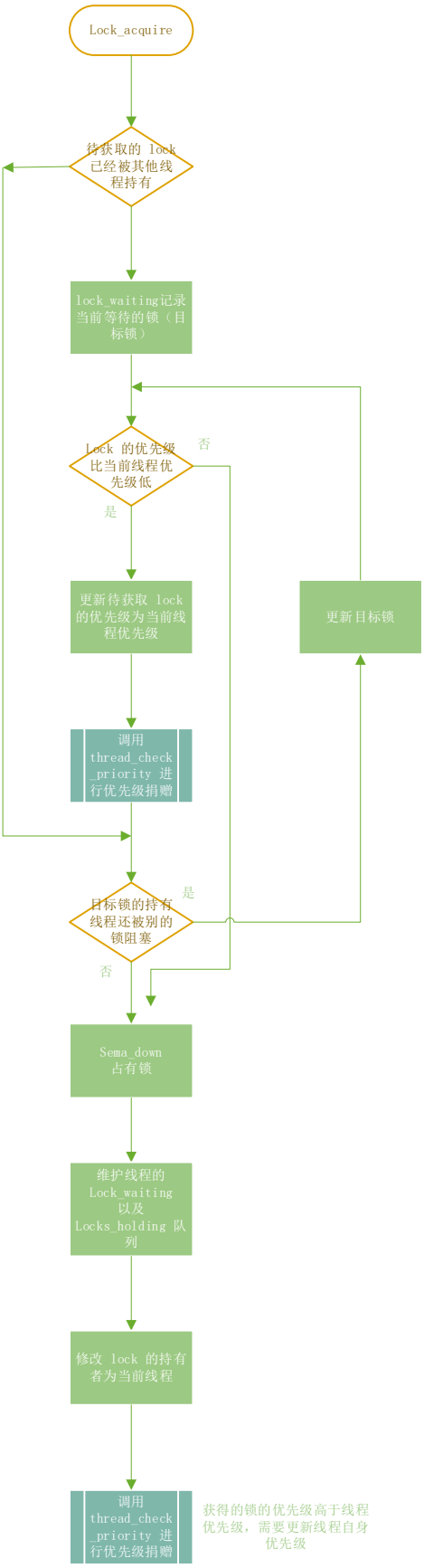
因此，我们需要对 `lock_acquire` 函数，进行改造。需要调用 `thread_check_priority` 的时机就在这里。



要注意的是，在这个函数中，**thread_check_priority** 函数需要调用两次：

- 在锁被获取前需要进行优先级捐赠 *提升占有这个锁的线程的优先级，使其尽快结束*
- 在锁被获取之后需要检查拿到的锁的优先级 *判断拿到的锁是否具有更高的优先级，提升自身的优先级，使自己尽快结束（递归捐赠）*

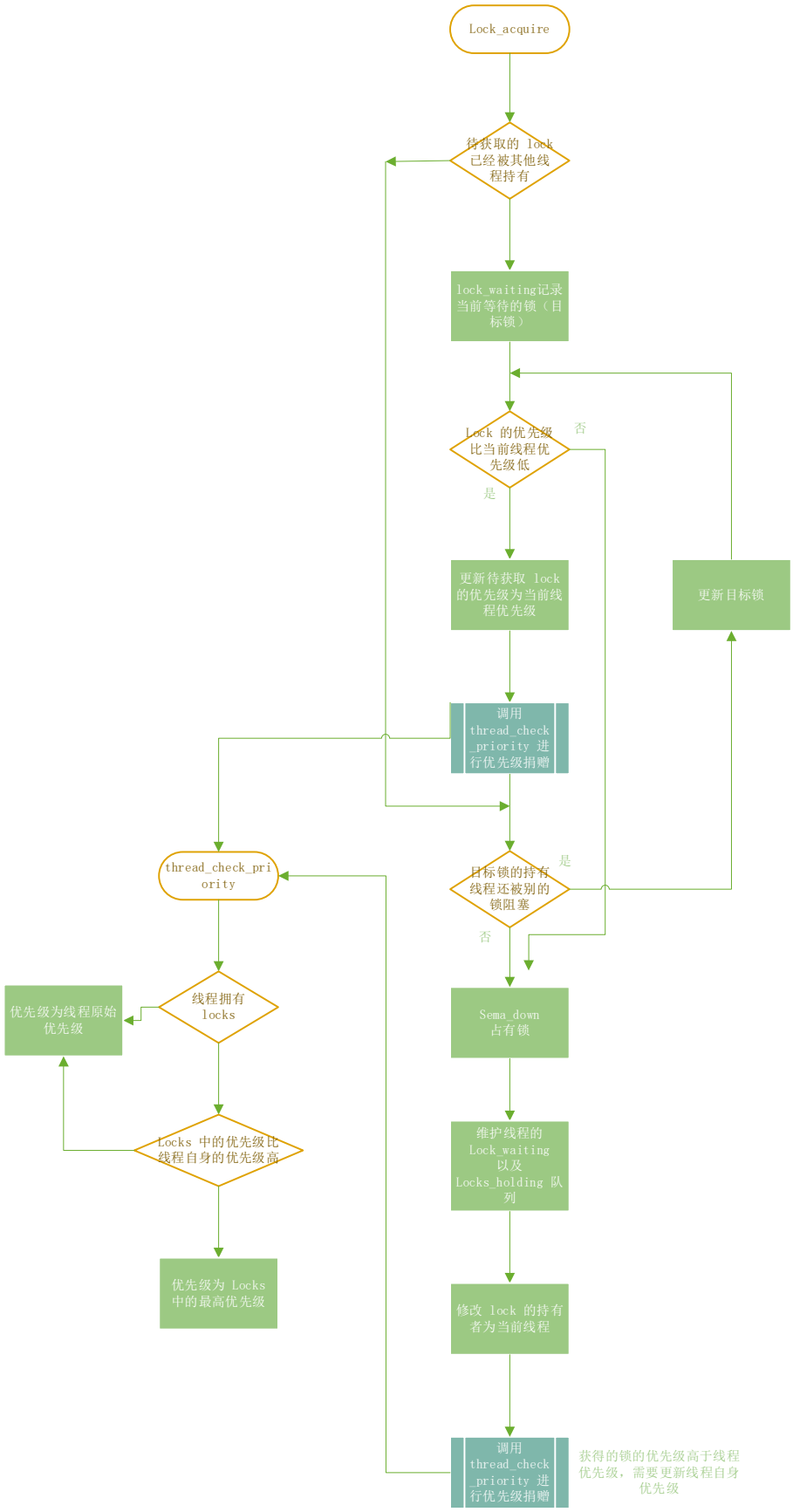
实现之前的流程之后 `make check` 运行测试样例发现，无法通过多重捐赠的测试，因此需要对流程进行修改，在循环中对锁进行逐层迭代遍历。修改后的 `lock_acquire` 函数流程如下图所示：



如此，我们就完成了通过锁的优先级，对线程的优先级进行调整。同时，由于有了实验二线程优先级实现的保证，可以确定，只要线程的优先级发生了改变，线

程的执行顺序就会按照我们的设计意图进行。至此，优先级捐赠过程就大致设计完成。

除此之外，还应该注意的是，当线程自身的优先级发生改变的时候（线程的创建、唤醒、优先级通过函数修改等）同样需要调用 `thread_check_priority` ，并维护其所持有锁的优先级，在这里就不再赘述。


Lock_acquire
待获取的 lock 已经被其他线程持有
lock_waiting记录当前等待的锁（目标锁）
Lock 的优先级比当前线程优先级低
否
是
更新待获取 lock 的优先级为当前线程优先级
更新目标锁
调用 thread_check_priority 进行优先级捐赠
thread_check_priority
目标锁的持有线程还被别的锁阻塞
是
否
优先级为线程原始优先级
线程拥有 locks
Sema_down 占有锁
Locks 中的优先级比线程自身的优先级高
维护线程的 Lock_waiting 以及 Locks_holding 队列
优先级为 Locks 中的最高优先级
修改 lock 的持有者为当前线程
调用 thread_check_priority 进行优先级捐赠
获得的锁的优先级高于线程优先级，需要更新线程自身优先级

三、具体实现

新增加的数据结构

- **在 struct lock 中新增**

```
int priority;               /* Priority of the lock. */
```

  - priority 为锁的优先级，该优先级在锁没有被获取时候为 PRI_MIN，被获取后等于等待获取锁的所有线程优先级的最大值。

- **在 struct thread 中新增**

```
int old_priority;                   /* Old priority. */
struct list locks_holding;          /* Locks that the thread is ho
lding. */
struct lock *lock_waiting;          /* The lock that the thread is
 waiting for. */
```

  - old_priority 记录线程自己的优先级，不受捐献的优先级影响。
  - locks_holding 记录拥有锁的列表。
  - lock_waiting 记录需要等待获取的锁。

编写 thread_check_priority 函数

在 thread.c 中新增，并在 thread.h 中定义

```
void
thread_check_priority (struct thread *t)
{
  int max_priority = PRI_MIN;

  if (!list_empty (&t->locks_holding))
  {
    list_sort (&t->locks_holding, lock_cmp_by_priority, NULL);
    if (list_entry (list_front (&t->locks_holding), struct lock, elem)
->priority > max_priority)
      max_priority = list_entry (list_front (&t->locks_holding), stru
ct lock, elem)->priority;
  }

  if (max_priority > t->old_priority)
    t->priority = max_priority;
  else
```

```
    t->priority = t->old_priority;

  list_sort (&ready_list, thread_cmp_by_priority, NULL);
}
```

lock_acquire 修改

按照设计流程图对 lock_acquire 进行修改，其中，迭代实现多重捐赠通过 while 循环 + 迭代器指针实现。

```
void
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  if (lock->holder != NULL)
  {
    thread_current ()->lock_waiting = lock;
    struct lock *iterator_lock = lock;
    while (iterator_lock != NULL &&
           thread_current ()->priority > iterator_lock->priority)
    {
      iterator_lock->priority = thread_current ()->priority;
      thread_check_priority (iterator_lock->holder);
      iterator_lock = iterator_lock->holder->lock_waiting;
    }
  }

  sema_down (&lock->semaphore);

  thread_current ()->lock_waiting = NULL;
  list_insert_ordered (&thread_current ()->locks_holding, &lock->elem,
 lock_cmp_by_priority, NULL);
  lock->holder = thread_current ();
  thread_check_priority (thread_current ());
}
```

其他

在 lock_release 函数新增下列语句

```
list_remove (&lock->elem);
  thread_check_priority (thread_current ());
```

```
  lock->priority = PRI_MIN;
  lock->holder = NULL;
```

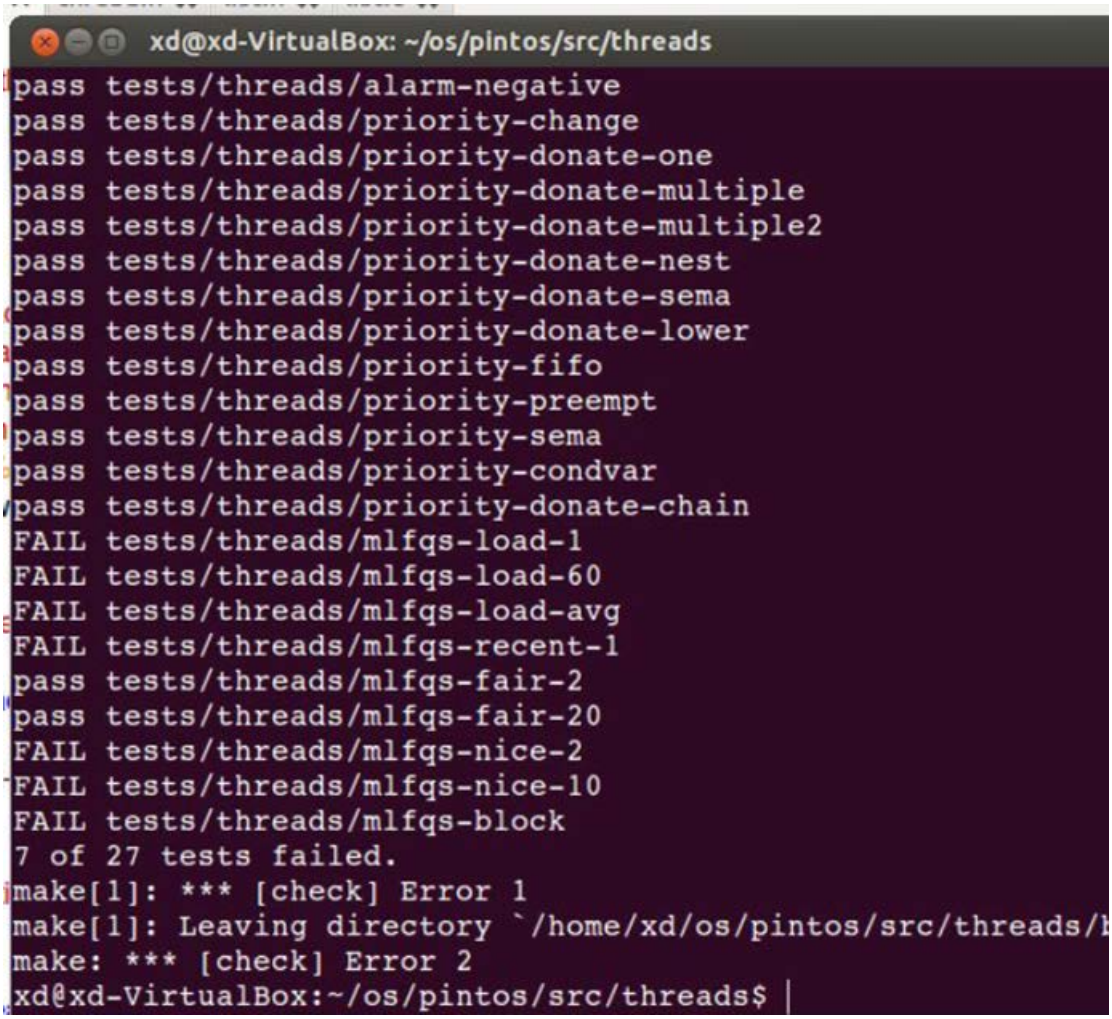
修改 thread*set*priority 函数

```
void
thread_set_priority (int new_priority)
{
  thread_current ()->old_priority = new_priority;
  thread_check_priority (thread_current ());
  thread_yield ();
}
```

直到这里，优先级捐赠已经实现。

四、实验结果

完成上述修改后，对 Pintos 源码进行重新 make check, 可以看到，通过了和线程优先级相关的以下测试，达到了实验要求



五、实验心得

- 我想，通过本次实验，我最大的收获，并不只是完成了复杂的优先级捐献，除了获得通过测试的成就感外，我还学会了解决复杂问题的一种方法。有的时候，问题看似复杂，在问题纷乱的变化中，我认为最重要的，还是找出其中真正不变的核心。在问题复杂的表象下，寻找其核心变化的规律。只有保证了核心逻辑的正确，问题才可能完全解决。

- 古语云：三思而后行。把过程弄清楚，解决方案想清楚，有了清晰的思路，再开始写代码也不迟。这次实验刚开始的时候，急于上手写代码，走了很多弯路。后来想，不如先写报告，把思路梳理一下，后续的实验过程才顺利许多。