

# 中山大学移动信息工程学院本科生实验报告

(2016 学年秋季学期)

课程名称: Operating System

任课教师: 饶洋辉

批改人(此处为 TA 填写):

年级+班级	1403	专业 (方向)	移动互联网
学号	14353435	姓名	朱恩东
电话	15919154005	Email	<a href="mailto:562870140@qq.com">562870140@qq.com</a>
开始日期	2016. 4. 25	完成日期	2016. 5. 4

## 1. 实验目的

通过修改 pintos, 修复优先级反转现象。实现优先级捐赠与恢复机制, 并可以适用于较为复杂的多重捐赠和嵌套捐赠的情况。通过 priority-donate 系列测试。

## 2. 实验过程

(1) test 解释:

priority-donate-one:

首先我们来看源代码:

```
void
test_priority_donate_one (void)
{
    struct lock lock;

    /* This test does not work with the MLFQS. */
    ASSERT (!thread_mlfqs);

    /* Make sure our priority is the default. */
    ASSERT (thread_get_priority () == PRI_DEFAULT);

    lock_init (&lock);
    lock_acquire (&lock);
    thread_create ("acquire1", PRI_DEFAULT + 1, acquire1_thread_func, &lock);
    msg ("This thread should have priority %d. Actual priority: %d.",
        PRI_DEFAULT + 1, thread_get_priority ());
    thread_create ("acquire2", PRI_DEFAULT + 2, acquire2_thread_func, &lock);
    msg ("This thread should have priority %d. Actual priority: %d.",
        PRI_DEFAULT + 2, thread_get_priority ());
    lock_release (&lock);
    msg ("acquire2, acquire1 must already have finished, in that order.");
    msg ("This should be the last line before finishing this test.");
}
```

以上是测试的主体部分, 主线程首先初始化一个锁 lock, 并占有该锁。此后主线程创建了一个新的线程 acquire1, 该线程的优先级为 PRI\_DEFAULT+1, 高于主线程的优先级, 因此会发生优先级抢占, 并开始执行 acquire1\_thread\_func, 其实现如下:

```

45  static void
46  acquire1_thread_func (void *lock_)
47  {
48      struct lock *lock = lock_;
49
50      lock_acquire (lock);
51      msg ("acquire1: got the lock");
52      lock_release (lock);
53      msg ("acquire1: done");
54  }

```

可以看到，acquire1 所做的其实就是申请锁 lock，但是由于锁 lock 已经被主线程所占有，因此 acquire1 会在 P 操作时被 block 掉，而又因为 acquire1 的优先级高于主线程 main，因此在 lock\_acquire 操作中还会发生一次优先级捐赠，捐赠后主线程的优先级变为 PRI\_DEFAULT+1，与 acquire1 相同。

由于 acquire1 被 block 掉了，主线程重新开始执行，并输出一段文字展示当前优先级，因此如果优先级捐赠操作正确的话，应该可以看到输出的两个数字是相同的。接下来主线程又创建了一个新的线程 acquire2，其优先级为 PRI\_DEFAULT+2，高于主线程当前的优先级，因此会发生优先级抢占，并开始执行 acquire1\_thread\_func，其实现如下：

```

56  static void
57  acquire2_thread_func (void *lock_)
58  {
59      struct lock *lock = lock_;
60
61      lock_acquire (lock);
62      msg ("acquire2: got the lock");
63      lock_release (lock);
64      msg ("acquire2: done");
65  }

```

可以看到，acquire2 所做的仍然是申请锁 lock，但是由于锁 lock 仍在被主线程所占有，因此 acquire2 会在 P 操作时也会被 block 掉，而又因为 acquire2 的优先级高于主线程 main，因此在 lock\_acquire 操作中又会发生第二次优先级捐赠，捐赠后主线程的优先级变为 PRI\_DEFAULT+2，与 acquire2 相同。

主线程重新开始执行，并再次输出一段文字展示当前优先级，因此如果优先级捐赠操作正确的话，应该可以看到输出的两个数字还是相同的。之后主线程进行 lock\_release 操作，归还了锁。由于主线程只占有 lock 这一个锁，因此主线程在 lock\_release 操作中会发生优先级恢复，恢复至其最初的优先级 PRI\_DEFAULT。同时唤醒等待队列中优先级最高的线程 acquire2 被唤醒，由于主线程优先级较低，因此将会让出 cpu。

接下来 acquire2 重新开始执行，得到锁后输出提示信息，然后释放锁 lock，此时 acquire1 从等待队列中被唤醒，但由于 acquire2 的优先级更高，因此会继续执行 acquire2 直到线程结束被杀死。然后 acquire1 抢占到 cpu，得到了锁 lock，并输出提示信息，然后释放锁 lock，这次没有线程被唤醒，acquire1 继续执行到结束被杀死。接着主线程才再次抢占到 cpu，输出最后的提示信息之后也执行结束，整个 test 完成。因此我们预期的输出结果应该为：

```

6 (priority-donate-one) begin
7 (priority-donate-one) This thread should have priority 32. Actual priority: 32.
8 (priority-donate-one) This thread should have priority 33. Actual priority: 33.
9 (priority-donate-one) acquire2: got the lock
10 (priority-donate-one) acquire2: done
11 (priority-donate-one) acquire1: got the lock
12 (priority-donate-one) acquire1: done
13 (priority-donate-one) acquire2, acquire1 must already have finished, in that order.
14 (priority-donate-one) This should be the last line before finishing this test.
15 (priority-donate-one) end

```

priority-donate-multiple:

首先我们来看源代码:

```

21 void
22 test_priority_donate_multiple (void)
23 {
24     struct lock a, b;
25
26     /* This test does not work with the MLFQS. */
27     ASSERT (!thread_mlfqs);
28
29     /* Make sure our priority is the default. */
30     ASSERT (thread_get_priority () == PRI_DEFAULT);
31
32     lock_init (&a);
33     lock_init (&b);
34
35     lock_acquire (&a);
36     lock_acquire (&b);
37
38     thread_create ("a", PRI_DEFAULT + 1, a_thread_func, &a);
39     msg ("Main thread should have priority %d. Actual priority: %d.",
40         PRI_DEFAULT + 1, thread_get_priority ());
41
42     thread_create ("b", PRI_DEFAULT + 2, b_thread_func, &b);
43     msg ("Main thread should have priority %d. Actual priority: %d.",
44         PRI_DEFAULT + 2, thread_get_priority ());
45
46     lock_release (&b);
47     msg ("Thread b should have just finished.");
48     msg ("Main thread should have priority %d. Actual priority: %d.",
49         PRI_DEFAULT + 1, thread_get_priority ());
50
51     lock_release (&a);
52     msg ("Thread a should have just finished.");
53     msg ("Main thread should have priority %d. Actual priority: %d.",
54         PRI_DEFAULT, thread_get_priority ());
55 }

```

以上是测试的主体部分，主线程首先初始化两个锁 a 和 b，并占有这两个锁。此后主线程先创建了一个新的线程 a，该线程的优先级为 PRI\_DEFAULT+1，高于主线程的优先级，因此会发生优先级抢占，并开始执行 a\_thread\_func，其实现如下：



```

57 static void
58 a_thread_func (void *lock_)
59 {
60     struct lock *lock = lock_;
61
62     lock_acquire (lock);
63     msg ("Thread a acquired lock a.");
64     lock_release (lock);
65     msg ("Thread a finished.");
66 }

```

可以看到，线程 a 所做的其实就是申请锁 a，但是由于锁 a 已经被主线程所占有，因此线程 a 会在 P 操作时被 block 掉，而又因为线程 a 的优先级高于主线程 main，因此在 lock\_acquire 操作中还会发生一次优先级捐赠，捐赠后主线程的优先级变为 PRI\_DEFAULT+1，与线程 a 相同。

由于线程 a 被 block 掉了，主线程重新开始执行，并输出一段文字展示当前优先级，因此如果优先级捐赠操作正确的话，应该可以看到输出的两个数字是相同的。接下来主线程又创建了一个新的线程线程 b，其优先级为 PRI\_DEFAULT+2，高于主线程当前的优先级，因此会发生优先级抢占，并开始执行 b\_thread\_func，其实现如下：

```

68 static void
69 b_thread_func (void *lock_)
70 {
71     struct lock *lock = lock_;
72
73     lock_acquire (lock);
74     msg ("Thread b acquired lock b.");
75     lock_release (lock);
76     msg ("Thread b finished.");
77 }

```

可以看到，进程 b 所做的是申请锁 b，但是由于锁 b 已经在被主线程所占有，因此进程 b 会在 P 操作时也会被 block 掉，而又因为进程 b 的优先级高于主线程 main，因此在 lock\_acquire 操作中又会发生第二次优先级捐赠，捐赠时会进行判断，由于主线程当前优先级为 PRI\_DEFAULT+1，是小于进程 b 的优先级的，因此该优先级捐赠成功，捐赠后主线程的优先级变为 PRI\_DEFAULT+2，与 acquire2 相同。

主线程重新开始执行，并再次输出一段文字展示当前优先级，因此如果优先级捐赠操作正确的话，应该可以看到输出的两个数字还是相同的。之后主线程进行 lock\_release 操作，归还了锁 b。由于主线程同时占有着 a 和 b 两个锁，因此主线程在 lock\_release 操作中的优先级恢复需要进行判断，由于锁 a 的优先级为 PRI\_DEFAULT+1，是大于主线程原本的优先级的，因此主线程会恢复至锁 a 的优先级 PRI\_DEFAULT+1。同时唤醒锁 b 的等待队列中优先级最高的线程 b 被唤醒，由于主线程优先级较低，因此将会让出 cpu。

接下来进程 b 得到 cpu，开始执行。得到锁后输出提示信息，然后释放锁 b，此时锁 b 的等待队列为空，没有线程被唤醒。然后主线程再次抢占到 cpu，接着输出优先级提示信息。然后释放了锁 a，这时锁 a 的等待队列中优先级最高的进程 a 被唤醒，而由于此时主线程已不再拥有任何锁，所以他的优先级被恢复到最初的 PRI\_DEFAULT。接着优先级更高的线程 a 再次抢占到 cpu，输出最后的提示信息之后执行结束，接下来主线程再次得到 cpu 输出最后的提示信息后也执行完毕，整个 test 完成。因此我们预期的输出结果应该为：

```

6 (priority-donate-multiple) begin
7 (priority-donate-multiple) Main thread should have priority 32. Actual priority: 32.
8 (priority-donate-multiple) Main thread should have priority 33. Actual priority: 33.
9 (priority-donate-multiple) Thread b acquired lock b.
10 (priority-donate-multiple) Thread b finished.
11 (priority-donate-multiple) Thread b should have just finished.
12 (priority-donate-multiple) Main thread should have priority 32. Actual priority: 32.
13 (priority-donate-multiple) Thread a acquired lock a.
14 (priority-donate-multiple) Thread a finished.
15 (priority-donate-multiple) Thread a should have just finished.
16 (priority-donate-multiple) Main thread should have priority 31. Actual priority: 31.
17 (priority-donate-multiple) end

```

priority-donate-multiple2:

首先我们来看源代码:

```

27 void
28 test_priority_donate_multiple2 (void)
29 {
30     struct lock a, b;
31
32     /* This test does not work with the MLFQS. */
33     ASSERT (!thread_mlfqs);
34
35     /* Make sure our priority is the default. */
36     ASSERT (thread_get_priority () == PRI_DEFAULT);
37
38     lock_init (&a);
39     lock_init (&b);
40
41     lock_acquire (&a);
42     lock_acquire (&b);
43
44     thread_create ("a", PRI_DEFAULT + 3, a_thread_func, &a);
45     msg ("Main thread should have priority %d. Actual priority: %d.",
46         PRI_DEFAULT + 3, thread_get_priority ());
47
48     thread_create ("c", PRI_DEFAULT + 1, c_thread_func, NULL);
49
50     thread_create ("b", PRI_DEFAULT + 5, b_thread_func, &b);
51     msg ("Main thread should have priority %d. Actual priority: %d.",
52         PRI_DEFAULT + 5, thread_get_priority ());
53
54     lock_release (&a);
55     msg ("Main thread should have priority %d. Actual priority: %d.",
56         PRI_DEFAULT + 5, thread_get_priority ());
57
58     lock_release (&b);
59     msg ("Threads b, a, c should have just finished, in that order.");
60     msg ("Main thread should have priority %d. Actual priority: %d.",
61         PRI_DEFAULT, thread_get_priority ());
62 }

```

以上是测试的主体部分，主线程首先初始化两个锁 a 和 b，并占有这两个锁。此后主线程先创建了一个新的线程 a，该线程的优先级为 PRI\_DEFAULT+3，高于主线程的优先级，因此会发生优先级抢占，并开始执行 a\_thread\_func，其实现如下：

```

64  static void
65  a_thread_func (void *lock_)
66  {
67      struct lock *lock = lock_;
68
69      lock_acquire (lock);
70      msg ("Thread a acquired lock a.");
71      lock_release (lock);
72      msg ("Thread a finished.");
73  }

```

可以看到，线程 a 所做的是申请锁 a，但是由于锁 a 已经被主线程所占有，因此线程 a 会在 P 操作时被 block 掉，而又因为线程 a 的优先级高于主线程 main，因此在 lock\_acquire 操作中还会发生一次优先级捐赠，捐赠后主线程的优先级变为 PRI\_DEFAULT+3，与线程 a 相同。

由于线程 a 被 block 掉了，主线程重新开始执行，并输出一段文字展示当前优先级，因此如果优先级捐赠操作正确的话，应该可以看到输出的两个数字是相同的。接下来主线程又创建了一个新的线程线程 c，其优先级为 PRI\_DEFAULT+1，低于主线程当前的优先级，因此被放入就绪队列中，不会被执行。接下来主线程又创建了一个新的线程 b，其优先级为 PRI\_DEFAULT+5，高于主线程优先级，因此发生优先级抢占，b 进程获得 cpu，并开始执行 b\_thread\_func，其实现如下：

```

75  static void
76  b_thread_func (void *lock_)
77  {
78      struct lock *lock = lock_;
79
80      lock_acquire (lock);
81      msg ("Thread b acquired lock b.");
82      lock_release (lock);
83      msg ("Thread b finished.");
84  }

```

可以看到，线程 b 所做的是申请锁 b，但是由于锁 b 已经被主线程所占有，因此线程 b 会在 P 操作时被 block 掉，而又因为线程 b 的优先级高于主线程 main，因此在 lock\_acquire 操作中还会发生一次优先级捐赠，因为主线程已经接受过一次捐赠，所以在捐赠时需要进行一次判断。因为线程 b 的优先级高于主线程当前接受捐赠的优先级，因此此次捐赠可以成功进行。捐赠后主线程的优先级变为 PRI\_DEFAULT+5，与线程 b 相同。

主线程继续执行，并释放了锁 a，因此锁 a 的等待队列中优先级最高的线程 a 被唤醒，并且由于主线程还拥有一个锁 b，而锁 b 的优先级高于锁 a 的优先级，因此主线程的优先级在此时还不会被更改。由于 a 的优先级低于当前主线程的优先级，因此也不会发生优先级抢占，a 被放入就绪队列中等待，而主线程继续执行。主线程接下来释放了锁 b，因此锁 b 的等待队列中优先级最高的线程 b 被唤醒，由于此时主线程已经不拥有任何锁，因此优先级被还原为 PRI\_DEFAULT，同时发生优先级抢占。而此时主线程以及线程 a，b，c 都已经在就绪队列中，因此优先级最高的线程 b 先抢占到 cpu，执行完函数中剩下的语句，输出相应信息；接下来线程 a 抢占到 cpu，执行完它对应函数的剩余语句，并输出相应信息；然后线程 c 抢占到 cpu，执行完它对应函数的剩余语句，并输出相应信息；最后由主线程获得 cpu，完成整个 test。因此预期的输出结果应该为：



```

6 (priority-donate-multiple2) begin
7 (priority-donate-multiple2) Main thread should have priority 34. Actual priority: 34.
8 (priority-donate-multiple2) Main thread should have priority 36. Actual priority: 36.
9 (priority-donate-multiple2) Main thread should have priority 36. Actual priority: 36.
10 (priority-donate-multiple2) Thread b acquired lock b.
11 (priority-donate-multiple2) Thread b finished.
12 (priority-donate-multiple2) Thread a acquired lock a.
13 (priority-donate-multiple2) Thread a finished.
14 (priority-donate-multiple2) Thread c finished.
15 (priority-donate-multiple2) Threads b, a, c should have just finished, in that order.
16 (priority-donate-multiple2) Main thread should have priority 31. Actual priority: 31.
17 (priority-donate-multiple2) end

```

priority-donate-lower:

首先我们来看源代码:

```

15 void
16 test_priority_donate_lower (void)
17 {
18     struct lock lock;
19
20     /* This test does not work with the MLFQS. */
21     ASSERT (!thread_mlfqs);
22
23     /* Make sure our priority is the default. */
24     ASSERT (thread_get_priority () == PRI_DEFAULT);
25
26     lock_init (&lock);
27     lock_acquire (&lock);
28     thread_create ("acquire", PRI_DEFAULT + 10, acquire_thread_func, &lock);
29     msg ("Main thread should have priority %d. Actual priority: %d.",
30         PRI_DEFAULT + 10, thread_get_priority ());
31
32     msg ("Lowering base priority...");
33     thread_set_priority (PRI_DEFAULT - 10);
34     msg ("Main thread should have priority %d. Actual priority: %d.",
35         PRI_DEFAULT + 10, thread_get_priority ());
36     lock_release (&lock);
37     msg ("acquire must already have finished.");
38     msg ("Main thread should have priority %d. Actual priority: %d.",
39         PRI_DEFAULT - 10, thread_get_priority ());
40 }

```

以上是测试的主体部分, 主线程首先初始化初始化一个锁 lock, 然后占有该锁, 接着主线程创建了一个优先级为 PRI\_DEFAULT+10 的线程 acquire, 由于子线程优先级较高, 因此会抢占到 cpu, 并开始执行 acquire\_thread\_func 函数, 其实现如下:

```

42 static void
43 acquire_thread_func (void *lock_)
44 {
45     struct lock *lock = lock_;
46
47     lock_acquire (lock);
48     msg ("acquire: got the lock");
49     lock_release (lock);
50     msg ("acquire: done");
51 }

```

可以看到子线程在函数中会申请锁 lock，但由于该锁已经被主线程所占有，因此 acquire 线程会在 P 操作时被 block 掉，同时由于子线程的优先级较高，因此 acquire 线程会把自己的优先级捐赠给主线程，捐赠后主线程的优先级为 PRI\_DEFAULT+10，和子线程相同。

之后，主线程重新获得 cpu，并继续执行测试代码，输出提示信息。如果优先级捐赠操作是正确的，我们应该能看到输出的两个优先级是相同的。此后，主线程尝试调低它的优先级为 PRI\_DEFAULT-10，但由于当前主线程仍处于被捐赠的状态，且被捐赠的优先级更高，因此在此时还不会发生优先级的改变，优先级仍为 PRI\_DEFAULT+10，继续执行。

随后主线程释放了锁 lock，并唤醒了 lock 的等待队列中优先级最高的 acquire 线程。同时，由于主线程此时已不再拥有任何锁，所以其优先级被还原为之前（已被更改）的 PRI\_DEFAULT-10，同时让出 cpu 给优先级更高的 acquire 线程。Acquire 线程随后继续执行完对应函数的内容，主线程重获 cpu，并输出最后的提示信息后完成测试。预期的输出结果应该为：

```

6 (priority-donate-lower) begin
7 (priority-donate-lower) Main thread should have priority 41. Actual priority: 41.
8 (priority-donate-lower) Lowering base priority...
9 (priority-donate-lower) Main thread should have priority 41. Actual priority: 41.
10 (priority-donate-lower) acquire: got the lock
11 (priority-donate-lower) acquire: done
12 (priority-donate-lower) acquire must already have finished.
13 (priority-donate-lower) Main thread should have priority 21. Actual priority: 21.
14 (priority-donate-lower) end

```

priority-donate-sema:

首先我们来看源代码：

```

29 void
30 test_priority_donate_sema (void)
31 {
32     struct lock_and_sema ls;
33
34     /* This test does not work with the MLFQS. */
35     ASSERT (!thread_mlfqs);
36
37     /* Make sure our priority is the default. */
38     ASSERT (thread_get_priority () == PRI_DEFAULT);
39
40     lock_init (&ls.lock);
41     sema_init (&ls.sema, 0);
42     thread_create ("low", PRI_DEFAULT + 1, l_thread_func, &ls);
43     thread_create ("med", PRI_DEFAULT + 3, m_thread_func, &ls);
44     thread_create ("high", PRI_DEFAULT + 5, h_thread_func, &ls);
45     sema_up (&ls.sema);
46     msg ("Main thread finished.");
47 }

```



以上是测试的主体部分，主线程首先初始化一个锁 lock，和一个初始值为 0 的信号量 sema。然后主线程先创建了一个优先级为 PRI\_DEFAULT+1 的线程 low，由于 low 线程优先级较高，因此会抢占到 cpu，并开始执行 l\_thread\_func 函数，其实现如下：

```
49 static void
50 l_thread_func (void *ls_)
51 {
52     struct lock_and_sema *ls = ls_;
53
54     lock_acquire (&ls->lock);
55     msg ("Thread L acquired lock.");
56     sema_down (&ls->sema);
57     msg ("Thread L downed semaphore.");
58     lock_release (&ls->lock);
59     msg ("Thread L finished.");
60 }
```

可以看到子线程 low 在函数中先会申请锁 lock，并且成功地占有了 lock，且输出对应的提示信息。接下来子线程开始尝试降低信号量 sema，但是由于 sema 的值已经为零，low 线程在 P 操作中被 block 掉了。因此主线程重新得到 cpu，继续执行。

之后，主线程又创建了一个新的线程 med，其优先级为 PRI\_DEFAULT+3，由于该线程优先级高于主线程，因此成功抢占到 cpu，开始执行函数 m\_thread\_func，其实现如下：

```
62 static void
63 m_thread_func (void *ls_)
64 {
65     struct lock_and_sema *ls = ls_;
66
67     sema_down (&ls->sema);
68     msg ("Thread M finished.");
69 }
70
```

可以看到，med 线程在函数中所做的是调低信号量 sema，而由于此时的信号量值仍然为 0，med 线程在 P 操作时被 block 掉，主线程再次获得 cpu。

随后主线程又创建了一个新的线程 high，其优先级为 PRI\_DEFAULT+5，，由于该线程优先级高于主线程，因此成功抢占到 cpu，开始执行函数 h\_thread\_func，其实现如下：

```
71 static void
72 h_thread_func (void *ls_)
73 {
74     struct lock_and_sema *ls = ls_;
75
76     lock_acquire (&ls->lock);
77     msg ("Thread H acquired lock.");
78
79     sema_up (&ls->sema);
80     lock_release (&ls->lock);
81     msg ("Thread H finished.");
82 }
```

可以看到，high 线程在函数中所做的是首先是申请锁 lock，而由于此时的 lock 已经被 low 线程所占有，所以 high 线程在 P 操作时会被 block 掉。且由于 high 线程的优先级高于 low 线程的优先级，因此发生了优先级的卷赠，low 线程的优先级在捐赠后变为 PRI\_DEFAULT+5。由于此时三个子线程都仍然处于 block 状态，所以主线程重新获得 cpu。

主线程随后提升信号量 sema 的值，因此 sema 的等待队列中优先级最高的线程 low 被唤醒，并且成功抢占了主线程的 cpu。线程 low 接着执行 l\_thread\_func 函数中剩余的内容，成功地降低了

sema 的信号量值，并释放了锁 lock。此时 lock 的等待队列中优先级最高的线程 high 被唤醒，而由于线程 low 此时已不再有任何锁，所以它的优先级被还原为原本的 PRI\_DEFAULT+1，并被线程 high 抢占到 cpu。线程 high 接着执行 h\_thread\_func 函数中的剩余内容，调高了信号量的值，因此 sema 的等待队列中的另一个线程 med 也被唤醒，加入到了就绪队列中。因此 high 执行完毕后，cpu 由优先级更高的 med 抢占到，执行完函数中的剩余内容。再由 low 抢占到 cpu，输出剩余内容，最后又主线程得到 cpu，完成测试。预期的输出结果应该为：

```
6 (priority-donate-sema) begin
7 (priority-donate-sema) Thread L acquired lock.
8 (priority-donate-sema) Thread L downed semaphore.
9 (priority-donate-sema) Thread H acquired lock.
10 (priority-donate-sema) Thread H finished.
11 (priority-donate-sema) Thread M finished.
12 (priority-donate-sema) Thread L finished.
13 (priority-donate-sema) Main thread finished.
14 (priority-donate-sema) end
```

priority-donate-nest:

首先我们来看源代码：

```
27 void
28 test_priority_donate_nest (void)
29 {
30     struct lock a, b;
31     struct locks locks;
32
33     /* This test does not work with the MLFQS. */
34     ASSERT (!thread_mlfqs);
35
36     /* Make sure our priority is the default. */
37     ASSERT (thread_get_priority () == PRI_DEFAULT);
38
39     lock_init (&a);
40     lock_init (&b);
41
42     lock_acquire (&a);
43
44     locks.a = &a;
45     locks.b = &b;
46     thread_create ("medium", PRI_DEFAULT + 1, medium_thread_func, &locks);
47     thread_yield ();
48     msg ("Low thread should have priority %d. Actual priority: %d.",
49         PRI_DEFAULT + 1, thread_get_priority ());
50
51     thread_create ("high", PRI_DEFAULT + 2, high_thread_func, &b);
52     thread_yield ();
53     msg ("Low thread should have priority %d. Actual priority: %d.",
54         PRI_DEFAULT + 2, thread_get_priority ());
55
56     lock_release (&a);
57     thread_yield ();
58     msg ("Medium thread should just have finished.");
59     msg ("Low thread should have priority %d. Actual priority: %d.",
60         PRI_DEFAULT, thread_get_priority ());
61 }
```

以上是测试的主体部分，主线程首先初始化两个锁 a 和 b，并且先占有了锁 a。随后主线程又创建了一个新的线程 medium，优先级为 PRI\_DEFAULT+1，由于 medium 线程优先级较高，因此会抢占到 cpu，并开始执行 medium\_thread\_func 函数，其实现如下：

```
63 static void
64 medium_thread_func (void *locks_)
65 {
66     struct locks *locks = locks_;
67
68     lock_acquire (locks->b);
69     lock_acquire (locks->a);
70
71     msg ("Medium thread should have priority %d. Actual priority: %d.",
72         PRI_DEFAULT + 2, thread_get_priority ());
73     msg ("Medium thread got the lock.");
74
75     lock_release (locks->a);
76     thread_yield ();
77
78     lock_release (locks->b);
79     thread_yield ();
80
81     msg ("High thread should have just finished.");
82     msg ("Middle thread finished.");
83 }
```

可以看到子线程 medium 在函数中先会申请锁 b，并且成功地占有了锁 b。接下来子线程 medium 开始申请锁 a，但是由于锁 a 已经被主线程所占用，因此 medium 线程在 P 操作中被 block 掉了。而又由于 medium 线程的优先级要高于主线程，因此在这里还会发生一次优先级的捐赠。主线程接受捐赠后优先级变为 PRI\_DEFAULT+1，重新得到 cpu，继续执行。

之后，主线程内做了一次 thread\_yield 操作，但是由于当前就绪队列中没有其他线程，因此主线程是不会让出 cpu 的。接下来，主线程又创建了一个新的线程 high，其优先级为 PRI\_DEFAULT+2，由于该线程优先级高于主线程，因此成功抢占到 cpu，开始执行函数 h\_thread\_func，并将锁 b 传入函数作为使用的参数，其实现如下：

```
85 static void
86 high_thread_func (void *lock_)
87 {
88     struct lock *lock = lock_;
89
90     lock_acquire (lock);
91     msg ("High thread got the lock.");
92     lock_release (lock);
93     msg ("High thread finished.");
94 }
```

可以看到，high 线程在函数中所做的是锁 b，而由于此时的锁 b 已被线程 medium 所占用，high 线程在 P 操作时会被 block 掉。同时，由于 high 线程的优先级较高，因此这里首先会发生一次对 medium 线程的优先级捐赠，medium 线程在接受捐赠后，优先级变为 PRI\_DEFAULT+2。同时，由于线程 medium 同时也在锁 a 的等待队列中，因此这里会发生一次嵌套捐赠，第二次捐赠由线程 a 捐赠给主线程，接受捐赠后，主线程的优先级也变为 PRI\_DEFAULT+2。

随后主线程继续执行，并释放了锁 a，这时锁 a 的等待队列中优先级最高的 medium 线程被唤醒，且由于主线程现在不再拥有任何一个锁，故其优先级被恢复为 PRI\_DEFAULT，并让出 cpu 给优先级更高的线程 medium 执行，开始执行函数 medium\_thread\_func 函数中剩余的内容。Medium 线程随后成功地得到 a b 两个锁，并输出了对应的提示信息。随后 medium 线程释放了锁 a，但这次锁 a 的等



待队列中已没有线程，所以不会有线程被唤醒，也不会出现优先级恢复的现象，medium 线程继续占用 cpu。接下来 medium 线程释放了锁 b，所以锁 b 的等待队列中的线程 high 被唤醒。同时，由于 medium 已不再占用任何锁，所以优先级被恢复为原本的 PRI\_DEFAULT+1，并让出 cpu 给优先级更高的 high 线程执行。

High 线程随后顺利地执行完函数中的剩余内容，并由 medium 线程再次抢占到 cpu 执行完毕，最后由主线程得到 cpu，输出剩余内容后执行完测试。预期的输出结果应该为：

```
6 (priority-donate-nest) begin
7 (priority-donate-nest) Low thread should have priority 32. Actual priority: 32.
8 (priority-donate-nest) Low thread should have priority 33. Actual priority: 33.
9 (priority-donate-nest) Medium thread should have priority 33. Actual priority: 33.
10 (priority-donate-nest) Medium thread got the lock.
11 (priority-donate-nest) High thread got the lock.
12 (priority-donate-nest) High thread finished.
13 (priority-donate-nest) High thread should have just finished.
14 (priority-donate-nest) Middle thread finished.
15 (priority-donate-nest) Medium thread should just have finished.
16 (priority-donate-nest) Low thread should have priority 31. Actual priority: 31.
17 (priority-donate-nest) end
```

priority-donate-chain:

首先我们来看源代码：

```
35 struct lock_pair
36 {
37     struct lock *second;
38     struct lock *first;
39 };
40
```

这是这个测试定义的一个结构体 lock\_pair，其本质其实其实就是由两个锁的指针 first 和 second 组成的。

```
44 void
45 test_priority_donate_chain (void)
46 {
47     int i;
48     struct lock locks[NESTING_DEPTH - 1];
49     struct lock_pair lock_pairs[NESTING_DEPTH];
50
51     /* This test does not work with the MLFQS. */
52     ASSERT (!thread_mlfqs);
53
54     thread_set_priority (PRI_MIN);
55
56     for (i = 0; i < NESTING_DEPTH - 1; i++)
57         lock_init (&locks[i]);
58
59     lock_acquire (&locks[0]);
60     msg ("%s got lock.", thread_name ());
61 }
```

```

62     for (i = 1; i < NESTING_DEPTH; i++)
63     {
64         char name[16];
65         int thread_priority;
66
67         snprintf (name, sizeof name, "thread %d", i);
68         thread_priority = PRI_MIN + i * 3;
69         lock_pairs[i].first = i < NESTING_DEPTH - 1 ? locks + i: NULL;
70         lock_pairs[i].second = locks + i - 1;
71
72         thread_create (name, thread_priority, donor_thread_func, lock_pairs + i);
73         msg ("%s should have priority %d. Actual priority: %d.",
74             thread_name (), thread_priority, thread_get_priority ());
75
76         snprintf (name, sizeof name, "interloper %d", i);
77         thread_create (name, thread_priority - 1, interloper_thread_func, NULL);
78     }
79
80     lock_release (&locks[0]);
81     msg ("%s finishing with priority %d.", thread_name (),
82         thread_get_priority ());
83 }

```

以上是测试的主体部分，主线程首先初始化两个数组，一个是 lock 类型的数组 locks，另一个是 lock\_pair 类型的 lock\_pairs。随后主线程把优先级调为 PRI\_MIN，也就是 0。然后先占有了锁 locks[0]。随后主线程进入了循环中，在每次循环中，首先是设置 lock\_pairs[i] 的 first 成员为 locks[i]，设置 lock\_pairs[i] 的 second 成员为 locks[i-1]，接着又创建了一个新的线程“thread i”，优先级为 PRI\_MIN+i\*3，由于新创建的线程优先级较高，因此会抢占到 cpu，并开始执行 donor\_thread\_func 函数，其实现如下：

```

85 static void
86 donor_thread_func (void *locks_)
87 {
88     struct lock_pair *locks = locks_;
89
90     if (locks->first)
91         lock_acquire (locks->first);
92
93     lock_acquire (locks->second);
94     msg ("%s got lock", thread_name ());
95
96     lock_release (locks->second);
97     msg ("%s should have priority %d. Actual priority: %d",
98         thread_name (), (NESTING_DEPTH - 1) * 3,
99         thread_get_priority ());
100
101     if (locks->first)
102         lock_release (locks->first);
103
104     msg ("%s finishing with priority %d.", thread_name (),
105         thread_get_priority ());
106 }
107

```

可以看到该函数中先会申请 locks 中的 first 成员，再申请 second 成员。由于每次循环时，locks 的 first 成员是可以被成功占有的，而 second 成员在之前一次的循环中已经被占用，所以每个执行该函数的子线程会在申请 locks->second 成员时被 block 掉。并且由于每次循环时创建的优先级是呈升序的，因此每次还会发生一次嵌套捐赠，捐赠后主线程以及之前创建的所有线程的优先级都为最新创建的线程的最高的优先级。而在每次 block 掉一个新创建的线程后，就绪队列中只剩下主线程，因此主线程再次得到 cpu，开始继续执行循环体中的剩余内容。每次循环中主线程还会创建一



个子线程“interloper i”，其优先级为  $PRI\_MIN+i*3-1$ ，但是由于主线程在接受优先级捐赠之后优先级更高一些，所以不会被抢占，可以继续执行。interloper\_thread\_func 函数中的内容如下：

```
108 static void
109 interloper_thread_func (void *arg_ UNUSED)
110 {
111     msg ("%s finished.", thread_name ());
112 }
```

其实就是输出一段提示信息，以表示是哪一个线程执行了当前函数。

7次循环之后，主线程跳出了循环，并释放了 locks[0]，此时“thread 1”被唤醒，并且使得主线程的优先级恢复为原来的  $PRI\_MIN$ 。因此“thread 1”成功地抢占了主线程的 cpu，并输出对应的提示信息。随后“thread 1”释放了 second 的成员，这时“thread 2”也会被唤醒，并使得“thread 1”的优先级也发生恢复，以此类推，直到“thread 7”被唤醒，才可以顺利地执行完成，最后又由就绪队列中优先级最高的“interloper 7”抢占到 cpu 并输出提示信息，接下来是“thread 6”.....最后由主线程获得 cpu，并执行完整个测试，预期的输出结果应该为：

```
6 (priority-donate-chain) begin
7 (priority-donate-chain) main got lock.
8 (priority-donate-chain) main should have priority 3. Actual priority: 3.
9 (priority-donate-chain) main should have priority 6. Actual priority: 6.
10 (priority-donate-chain) main should have priority 9. Actual priority: 9.
11 (priority-donate-chain) main should have priority 12. Actual priority: 12.
12 (priority-donate-chain) main should have priority 15. Actual priority: 15.
13 (priority-donate-chain) main should have priority 18. Actual priority: 18.
14 (priority-donate-chain) main should have priority 21. Actual priority: 21.
15 (priority-donate-chain) thread 1 got lock
16 (priority-donate-chain) thread 1 should have priority 21. Actual priority: 21
17 (priority-donate-chain) thread 2 got lock
18 (priority-donate-chain) thread 2 should have priority 21. Actual priority: 21
19 (priority-donate-chain) thread 3 got lock
20 (priority-donate-chain) thread 3 should have priority 21. Actual priority: 21
21 (priority-donate-chain) thread 4 got lock
22 (priority-donate-chain) thread 4 should have priority 21. Actual priority: 21
23 (priority-donate-chain) thread 5 got lock
24 (priority-donate-chain) thread 5 should have priority 21. Actual priority: 21
25 (priority-donate-chain) thread 6 got lock
26 (priority-donate-chain) thread 6 should have priority 21. Actual priority: 21
27 (priority-donate-chain) thread 7 got lock
28 (priority-donate-chain) thread 7 should have priority 21. Actual priority: 21
29 (priority-donate-chain) thread 7 finishing with priority 21.
30 (priority-donate-chain) interloper 7 finished.
31 (priority-donate-chain) thread 6 finishing with priority 18.
32 (priority-donate-chain) interloper 6 finished.
33 (priority-donate-chain) thread 5 finishing with priority 15.
34 (priority-donate-chain) interloper 5 finished.
35 (priority-donate-chain) thread 4 finishing with priority 12.
36 (priority-donate-chain) interloper 4 finished.
37 (priority-donate-chain) thread 3 finishing with priority 9.
38 (priority-donate-chain) interloper 3 finished.
39 (priority-donate-chain) thread 2 finishing with priority 6.
40 (priority-donate-chain) interloper 2 finished.
41 (priority-donate-chain) thread 1 finishing with priority 3.
42 (priority-donate-chain) interloper 1 finished.
43 (priority-donate-chain) main finishing with priority 0.
44 (priority-donate-chain) end
```



(2) 实验思路（伪代码）：

申请锁的伪代码：

```
1 Pseudocode for acquiring a lock
2   curr <- current thread
3   thrd <- lock's holder
4   another <- current lock
5
6   if thrd != NULL
7       while: 'another' satisfy the condition for priority denote
8           do priority denote operation to 'another' and its holder thread
9           update another to the locks curr waiting
10
11   acquire the lock
12   update the condition flag of curr and lock
```

优先级捐赠的伪代码：

```
1 Pseudocode for priority denoting
2   t <- the thread to be denoted
3   if t.locks not empty
4       sort locks
5       priority_ch <- obtain the maximal priority of the list from the head element
6       if priority_ch > t.priority
7           t.priority <- priority_ch
8   update the order of every thread element in list
```

释放锁的伪代码：

```
1 Pseudocode for releasing a lock
2   t <- current thread
3   lock <- the lock to be released
4   if t.locks not empty
5       sort locks
6       priority_ch <- obtain the maximal priority of the list from the head element
7       t.priority <- max(priority_ch, t.old_priority)
8   else
9       t.priority <- t.old_priority
10  update t.denoted, lock.holder
```

(3) 代码分析：

首先我们需要对 thread 结构体和 lock 结构体做一些改动，增加一些成员变量：

```
101
102   /*added for priority-donate */
103   int old_priority ;
104   struct list locks ;
105   bool donated ;
106   struct lock *blocked ;
107   /*added for priority-donate */
108
```

这是 thread 结构体中的修改，old\_priority 是指该线程原本的优先级，也即在优先级恢复时应最终恢复到的优先级；locks 是一链表，表示的是该线程占有的锁；donated 是一个 bool 变量，表示该线程当前的优先级是否是经过捐赠的；blocked 是一个 lock 的指针，表示当前线程是在等待哪一个锁。

```

25
26     /* added for priority-donate */
27     struct list_elem holder_elem ;
28     int lock_priority ;
29     /* added for priority-donate */

```

这是 lock 结构体中的修改, holder\_elem 表示该线程被那个线程在 locks 链表中所使用的链表元素; lock\_priority 表示该锁的优先级, 也就是在申请该锁的线程中最大的优先级。

```

199 void
200 lock_acquire (struct lock *lock)
201 {
202     ASSERT (lock != NULL);
203     ASSERT (!intr_context ());
204     ASSERT (!lock_held_by_current_thread (lock));
205
206     enum intr_level old_level = intr_disable() ;
207
208     struct thread * curr = thread_current() ;
209     struct thread * thrd = lock->holder ;
210     struct lock *another = lock ;
211
212     if(thrd != NULL)
213     {
214         thrd->donated = true ;
215         curr->blocked = lock ;
216         while(another != NULL && curr->priority > another->lock_priority)
217         {
218             another->lock_priority = curr->priority ;
219             thread_donate_priority (another->holder) ;
220             another = another->holder->blocked ;
221         }
222     }
223
224     sema_down (&lock->semaphore);
225
226     curr->blocked = NULL ;
227     lock->lock_priority = curr->priority ;
228     lock->holder = curr ;
229     list_insert_ordered(&thread_current()->locks, &lock->holder_elem, lock_cmp_priority, NULL) ;
230
231     intr_set_level(old_level) ;
232 }

```

这里是修改后的 lock\_acquire() 函数, 当当前线程申请一个锁的时候, 我们先看该锁是否被其他线程所占有, 如果没有被占有的话, 就可以直接把该锁分配给线程, 否则的话就需要执行第一个 if 语句内的内容, 进行优先级捐赠。在 if 语句内, 我们首先是修改了锁的持有者线程 thrd 的 donated 标志位为 true, 然后将当前线程 curr 的 blocked 指针指向当前正在申请的锁 lock, 而 while 循环一定能够先执行一次, 完成对 thrd 线程的优先级捐赠。接下来我们需要检查是否发生嵌套捐赠, 如果 thrd 线程也在等待另外一个锁, 且当前线程的优先级要高于那个锁的优先级, 则我们需要移动 another 指针到那个锁的持有者线程, 并做同样的优先级捐赠操作。重复 while 循环里的操作直到不满足嵌套捐赠的条件为止。

在 if 语句外是 sema\_down 函数, 表示对锁的 P 操作, 如果能够成功执行, 这说明当前线程已经获得了该锁, 所以我们接下来应该对当前线程以及锁的相关变量进行更新。首先当前线程能够顺利执行, 即说明没有在等待任何锁, 因此 curr 的 blocked 指针应该改为 NULL, 锁 lock 的 holder 指针应该指向当前线程 curr, 优先级应该与 curr 相同。最后, 我们要把该锁有序地插入到 curr 的 locks 链表中, 插入顺序由 lock\_cmp\_priority 函数定义。

```

629 void
630 thread_donate_priority(struct thread *t)
631 {
632     enum intr_level old_level = intr_disable();
633     if(!list_empty(&t->locks))
634     {
635         list_sort(&t->locks, lock_cmp_priority, NULL);
636         int lock_priority = list_entry(list_front(&t->locks), struct lock, holder_elem)->lock_priority;
637         if(lock_priority > t->old_priority)
638         {
639             t->priority = lock_priority;
640         }
641     }
642     list_remove(&t->elem);
643     list_insert_ordered(&ready_list, &t->elem, cmp_elem, NULL);
644
645     intr_set_level(old_level);
646 }
647

```

以上是 thread\_donate\_priority 函数的实现,该函数的作用其实就是再进行优先级捐赠时改变目标线程的优先级。首先我们要保证目标线程 t 的 locks 不为空,也即线程 t 必须有占有锁,在 if 语句内,我们要取出 t 的 locks 链表中优先级最大的锁,并判断是否进行优先级捐赠。假如说 locks 内部已经是一个按照优先级排好序的链表的话,则我们就可以直接取出它的首元素,但是要记得我们之前在 lock\_acquire 函数中是直接改变了 locks 里元素的优先级,所以这个链表原本的有序性会被破坏,那么如何实现更改链表中元素的值之后可以保持有序排列呢?这里有两种实现方法,一种是在修改元素时,直接创建一个新的元素有序插入,然后再把原来的元素删除,另一种就是我这里实现的在每次捐赠优先级时进行一次排序。考虑到代码的易读性,我使用的是排序的方式。这样我们可以保证变量 lock\_priority 中储存的是链表中锁的最大优先级,我们再把该最大优先级与线程原本的优先级进行比较,如果该优先级更大的话,则进行一次优先级捐赠。在完成优先级捐赠后,原本 t 所在线程队列的有序性也被打乱了,所以我们要先把原本的 t 线程删除,再有序插入完成优先级捐赠后的线程 t。

```

60 void
61 sema_down (struct semaphore *sema)
62 {
63     enum intr_level old_level;
64
65     ASSERT (sema != NULL);
66     ASSERT (!intr_context ());
67
68     old_level = intr_disable ();
69     while (sema->value == 0)
70     {
71         list_insert_ordered (&sema->waiters, &thread_current ()->elem, cmp_elem, NULL);
72         thread_block ();
73     }
74     sema->value--;
75     intr_set_level (old_level);
76 }
77

```

以上是修改后的 sema\_down() 函数的一部分,while 循环内是在判断信号是否等于 0,一旦等于 0,则该线程会被有序地插入到信号量 sema 的等待队列中,然后该线程被 block 掉。当跳出循环后,sema 信号量的值可以减一,并成功地完成 sema\_down 操作。



```

260 void
261 lock_release (struct lock *lock)
262 {
263     ASSERT (lock != NULL);
264     ASSERT (lock_held_by_current_thread (lock));
265     enum intr_level old_level = intr_disable() ;
266
267     list_remove(&lock->holder_elem) ;
268     struct thread *curr = thread_current() ;
269     int recover_priority = curr->old_priority ;
270     if(!list_empty(&curr->locks))
271     {
272         list_sort(&curr->locks, lock_cmp_priority , NULL) ;
273         int lock_priority = list_entry(list_front(&curr->locks), struct lock, holder_elem)->lock_priority ;
274         if(lock_priority > recover_priority)
275         {
276             recover_priority = lock_priority ;
277         }
278     }
279     curr->priority = recover_priority ;
280     lock->holder = NULL;
281     if(list_empty(&curr->locks))
282     {
283         curr->donated = false ;
284     }
285     sema_up (&lock->semaphore);
286
287     intr_set_level(old_level) ;
288 }

```

以上是 lock\_release() 函数，首先我们把 lock 从它所处的 locks 链表中删除掉，如果当前线程 curr 的链表 locks 为空，则说明 curr 线程已不占有任何锁，可以直接恢复其优先级到最初设置的 old\_priority 的值。否则的话，我们将其优先级设置为其链表中锁的最大优先级值，这也就是 if 语句中的内容。那么如何得到 locks 链表中优先级最大的锁呢，可能可以想到的一个办法是直接保证插入有序即可，但是如之前所说，我们中途是有直接更改锁的优先级的情况的，所以有序性会被打破，这也就是我们在取出优先级之前所做的 list\_sort 的原因。在更改完优先级之后，我们需要对 curr 以及 lock 的变量进行一次更新，首先是把 lock 的 holder 指针置为空，同时如果 curr 的 locks 链表为空的话，说明其当前已没有优先级捐赠情况，应该把 donated 重新置为 false。

```

108 void
109 sema_up (struct semaphore *sema)
110 {
111     ASSERT (sema != NULL);
112
113     enum intr_level old_level = intr_disable ();
114
115     int max_priority = PRI_MIN ;
116     if (!list_empty (&sema->waiters)){
117         list_sort(&sema->waiters, cmp_elem, NULL) ;
118         struct list_elem * awake = list_pop_front (&sema->waiters) ;
119         max_priority = list_entry(awake, struct thread, elem)->priority ;
120         thread_unblock (list_entry (awake, struct thread, elem));
121     }
122     sema->value++ ;
123     if(thread_current()->priority < max_priority) thread_yield();
124
125     intr_set_level (old_level);
126 }

```

这是修改后的 sema\_up() 函数部分，首先我们初始化一个 max\_priority 变量的值为 PRI\_MIN，如果当前信号量 sema 的等待队列为空，我们可以直接不唤醒任何线程，而直接调高信号量；但是如果 sema 的等待队列不为空，则需要执行 if 语句中的内容，首先欧文们对等待队列进行一次排

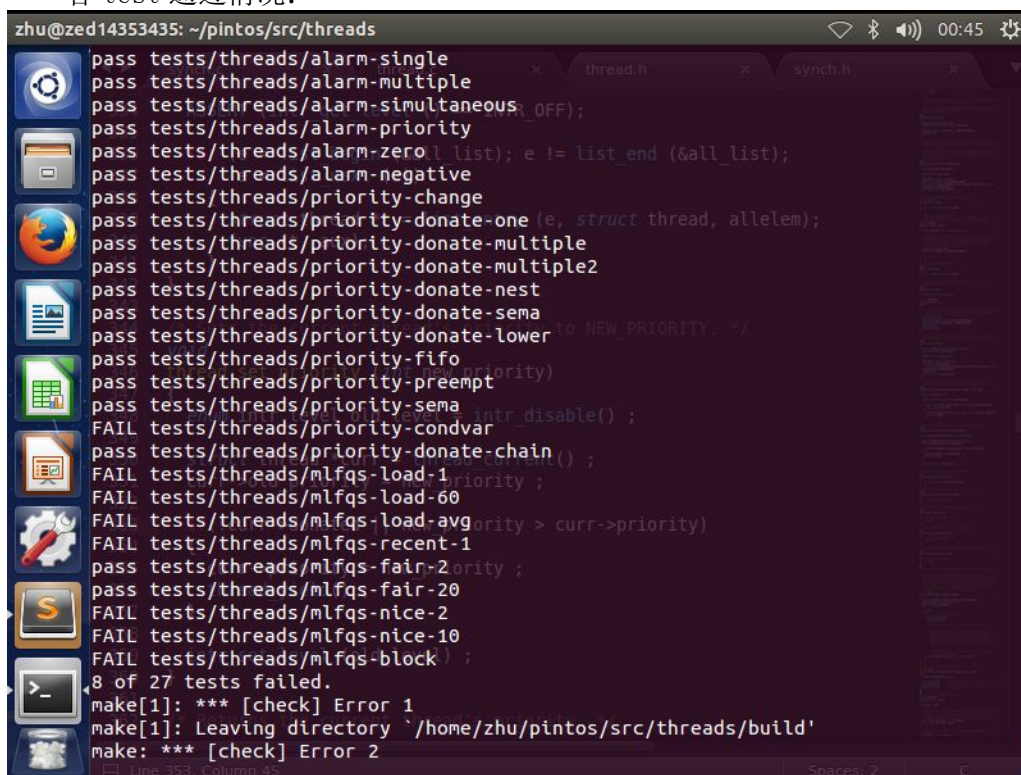
序，方便后面取出队列中优先级最大的线程，然后我们用 `max_priority` 来储存当前取出的优先级最高线程的优先级，并把该线程 block 掉。这里需要注意的一点是，我们让一个线程让出 cpu 的原则是就绪队列中必须要有优先级更高的变量，因此在最后我们需要进行一次判断，只有当当前线程的优先级低于就绪队列中的最大的优先级时才执行 `thread_yield()` 函数，让当前线程让出 cpu。

```
345 void
346 thread_set_priority (int new_priority)
347 {
348     enum intr_level old_level = intr_disable() ;
349
350     struct thread *curr = thread_current() ;
351     curr->old_priority = new_priority ;
352
353     if(!curr->donated || new_priority > curr->priority)
354     {
355         curr->priority = new_priority ;
356         thread_yield() ;
357     }
358
359     intr_set_level (old_level) ;
360 }
```

这是修改后的 `thread_set_priority` 函数，相比于之前其实我们只是多判断了一个情况，即当满足当前线程 `curr` 没有被捐赠优先级或者新设置的优先级要高于当前优先级时，才会更改当前优先级，并执行 `thread_yield()` 函数，这样做的目的是为了防止在捐赠优先级后，`thread_set_priority` 又把被捐赠的线程优先级再次调低了。但是无论如何 `old_priority` 的值都是会保存为当前设置的优先级的。

### 3. 实验结果

各 test 通过情况:



```
zhu@zed14353435: ~/pintos/src/threads
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
FAIL tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
8 of 27 tests failed.
make[1]: *** [check] Error 1
make[1]: Leaving directory `/home/zhu/pintos/src/threads/build'
make: *** [check] Error 2
```



priority-donate-one 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
Bochs x86 Emulator 2.6.2
Built from SVN snapshot on May 26, 2013
Compiled on Mar 25, 2016 at 20:40:44
=====
00000000000i[e = li] reading configuration from bochsrc.txt;
00000000000e[e = li] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keybo
ard' option.
00000000000i[struct] installing nogui module as the Bochs GUI;
00000000000i[inc (t) using log file bochsout.txt
Pilo hda1 }
Loading.....
Kernel command line: run priority-donate-one
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-donate-one':
(priority-donate-one) begin
(priority-donate-one) This thread should have priority 32. Actual priority: 32.
(priority-donate-one) This thread should have priority 33. Actual priority: 33.
(priority-donate-one) acquire2: got the lock
(priority-donate-one) acquire2: done
(priority-donate-one) acquire1: got the lock
(priority-donate-one) acquire1: done
(priority-donate-one) acquire2, acquire1 must already have finished, in that orde
r.
(priority-donate-one) This should be the last line before finishing this test.
(priority-donate-one) end
Execution of 'priority-donate-one' complete.
```

priority-donate-multiple 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
00000000000e[ ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keybo
ard' option.
00000000000i[ (int] installing nogui module as the Bochs GUI
00000000000i[ ] using log file bochsout.txt
Pilo hda1
Loading.....
Kernel command line: run priority-donate-multiple
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-donate-multiple':
(priority-donate-multiple) begin
(priority-donate-multiple) Main thread should have priority 32. Actual priority:
32.
(priority-donate-multiple) Main thread should have priority 33. Actual priority:
33.
(priority-donate-multiple) Thread b acquired lock b.
(priority-donate-multiple) Thread b finished.
(priority-donate-multiple) Thread b should have just finished.
(priority-donate-multiple) Main thread should have priority 32. Actual priority:
32.
(priority-donate-multiple) Thread a acquired lock a.
(priority-donate-multiple) Thread a finished.
(priority-donate-multiple) Thread a should have just finished.
(priority-donate-multiple) Main thread should have priority 31. Actual priority:
31.
(priority-donate-multiple) end
Execution of 'priority-donate-multiple' complete.
```

priority-donate-multiple2 运行情况:



```
zhu@zed14353435: ~/pintos/src/threads
ard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000e[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: run priority-donate-multiple2
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-donate-multiple2':
(priority-donate-multiple2) begin
(priority-donate-multiple2) Main thread should have priority 34. Actual priority
: 34.
(priority-donate-multiple2) Main thread should have priority 36. Actual priority
: 36.
(priority-donate-multiple2) Main thread should have priority 36. Actual priority
: 36.
(priority-donate-multiple2) Thread b acquired lock b.
(priority-donate-multiple2) Thread b finished.
(priority-donate-multiple2) Thread a acquired lock a.
(priority-donate-multiple2) Thread a finished.
(priority-donate-multiple2) Thread c finished.
(priority-donate-multiple2) Threads b, a, c should have just finished, in that or
der.
(priority-donate-multiple2) Main thread should have priority 31. Actual priority
: 31.
(priority-donate-multiple2) end
Execution of 'priority-donate-multiple2' complete.
```

priority-donate-nest 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
Compiled on Mar 25 2016 at 20:40:44
=====
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keybo
ard' option.
00000000000i[e = li] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: run priority-donate-nest
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-donate-nest':
(priority-donate-nest) begin
(priority-donate-nest) Low thread should have priority 32. Actual priority: 32.
(priority-donate-nest) Low thread should have priority 33. Actual priority: 33.
(priority-donate-nest) Medium thread should have priority 33. Actual priority: 3
3.
(priority-donate-nest) Medium thread got the lock.
(priority-donate-nest) High thread got the lock.
(priority-donate-nest) High thread finished.
(priority-donate-nest) High thread should have just finished.
(priority-donate-nest) Middle thread finished.
(priority-donate-nest) Medium thread should just have finished.
(priority-donate-nest) Low thread should have priority 31. Actual priority: 31.
(priority-donate-nest) end
Execution of 'priority-donate-nest' complete.
```

priority-donate-sema 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
squish-pty bochs -q

=====
334   ASSERT (intr_get_level() == INTR_LEVEL_NONE);
335   Built from SVN snapshot on May 26, 2013
336   Compiled on Mar 25 2016 at 20:40:44
=====
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keybo
ard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: run priority-donate-sema
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-donate-sema':
(priority-donate-sema) begin
(priority-donate-sema) Thread L acquired lock.
(priority-donate-sema) Thread L downed semaphore.
(priority-donate-sema) Thread H acquired lock.
(priority-donate-sema) Thread H finished.
(priority-donate-sema) Thread M finished.
(priority-donate-sema) Thread L finished.
(priority-donate-sema) Main thread finished.
(priority-donate-sema) end
Execution of 'priority-donate-sema' complete.
```

priority-donate-lower 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
Built from SVN snapshot on May 26, 2013
Compiled on Mar 25 2016 at 20:40:44

=====
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keybo
ard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: run priority-donate-lower
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-donate-lower':
(priority-donate-lower) begin
(priority-donate-lower) Main thread should have priority 41. Actual priority: 41
(priority-donate-lower) Lowering base priority.
(priority-donate-lower) Main thread should have priority 41. Actual priority: 41
(priority-donate-lower) acquire: got the lock
(priority-donate-lower) acquire: done
(priority-donate-lower) acquire must already have finished.
(priority-donate-lower) Main thread should have priority 21. Actual priority: 21
(priority-donate-lower) end
Execution of 'priority-donate-lower' complete.
```

priority-donate-chain 运行情况:

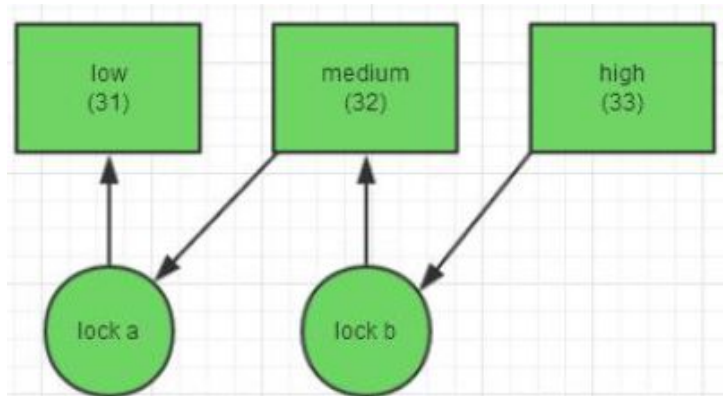


```
zhu@zed14353435: ~/pintos/src/threads
(priority-donate-chain) begin
(priority-donate-chain) main got lock.
(priority-donate-chain) main should have priority 3. Actual priority: 3.
(priority-donate-chain) main should have priority 6. Actual priority: 6.
(priority-donate-chain) main should have priority 9. Actual priority: 9.
(priority-donate-chain) main should have priority 12. Actual priority: 12.
(priority-donate-chain) main should have priority 15. Actual priority: 15.
(priority-donate-chain) main should have priority 18. Actual priority: 18.
(priority-donate-chain) main should have priority 21. Actual priority: 21.
(priority-donate-chain) thread 1 got lock
(priority-donate-chain) thread 1 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 2 got lock
(priority-donate-chain) thread 2 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 3 got lock
(priority-donate-chain) thread 3 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 4 got lock
(priority-donate-chain) thread 4 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 5 got lock
(priority-donate-chain) thread 5 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 6 got lock
(priority-donate-chain) thread 6 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 7 got lock
(priority-donate-chain) thread 7 should have priority 21. Actual priority: 21
(priority-donate-chain) thread 7 finishing with priority 21.
(priority-donate-chain) interloper 7 finished.
(priority-donate-chain) thread 6 finishing with priority 18.
(priority-donate-chain) interloper 6 finished.
(priority-donate-chain) thread 5 finishing with priority 15.
(priority-donate-chain) interloper 5 finished.
(priority-donate-chain) thread 4 finishing with priority 12.
(priority-donate-chain) interloper 4 finished.
(priority-donate-chain) thread 3 finishing with priority 9.
(priority-donate-chain) interloper 3 finished.
(priority-donate-chain) thread 2 finishing with priority 6.
(priority-donate-chain) interloper 2 finished.
(priority-donate-chain) thread 1 finishing with priority 3.
(priority-donate-chain) interloper 1 finished.
(priority-donate-chain) main finishing with priority 0.
(priority-donate-chain) end
Execution of 'priority-donate-chain' complete.
```

## 4. 回答问题

(1)

首先我们要知道嵌套捐赠发生的原因：



如上图，是最简单的一种嵌套捐赠的情况。High 线程在等待一个比自己优先级的线程占有的锁 lock b，但是由于 lock b 的拥有者 medium 线程同样在等待一个锁 lock a，所以如果我们只对 medium 线程进行优先级捐赠的话，将会使得 medium 线程的优先级高于 low 线程，但是又在等待 low 线程的优先级反转现象。

要解决这种现象，就要采取嵌套捐赠的方法，即我们在捐赠优先级的时候，不能仅仅考虑当前正在等待的锁的优先级，还要递归地去检查锁的拥有者在等待的锁的优先级，假如满足优先级捐赠的条件的话，就要继续进行捐赠，否则才停止当前优先级捐赠操作。

(2)

要解决一个线程占有多个锁的情况其实并不难，只需要牢牢记住一个线程的优先级一定是等于它拥有的锁当中最大的优先级（接受了优先级捐赠），或者等于它自身原本的优先级（未接受优先级捐赠）。之前我们的定义中已经规定，锁的优先级定义为其当前等待队列中最大的优先级。

所以，当获取一个锁的时候，我们首先检查一下当前锁的等待队列是否为空，如果为空的话，则我们可以直接把该锁的优先级设置为当前线程的优先级，如果等待队列不为空的话，则我们再对



当前线程的优先级和锁的等待队列中最大的优先级进行比较。如果锁的等待队列中的优先级更大的话，则进行一次优先级捐赠；反之，则我们可以直接获取该锁，并修改该锁的优先级。

当释放一个锁的时候，我们可以先把其从当前线程的 locks 链表中去除，并将该锁的 holder 指针置为空。同时我们需要判断当前线程 locks 链表在删除一个元素后是否为空，如果为空的话则该线程的优先级直接恢复为其原本的优先级，否则我们将其优先级恢复为 locks 队列中的最大优先级与当前线程原本优先级中的较大值。

## 5. 实验感想

本次实验是关于线程的优先级捐赠与恢复。本次实验较为复杂，其基本思想已经在之前讨论的比较详细了。所以在这部分我主要说一下在本次实验中遇到的一些细节问题。

首先是本次实验中一开始在函数的存放位置上出了一点差错。我原本就近地在 synch.c 实现了本次实验新增的一些优先级捐赠以及比较的函数，但是在编译时发现一直无法通过，报错信息是没有定义某些变量。起初我以为是 synch.c 没有包含必需的头文件，但是在手动加上 thread.h 的头文件后，发现问题依然没有解决。后来我又到 thread.c 中查看了每一个涉及到的变量，才发现有相当一部分的变量和函数是声明为了 static。之前我们接触到的 static 修饰符只要是用于修饰变量为静态变量，也即储存在内存的静态储存区。但是 static 修饰符还有一个作用是限制变量的作用域，它可以使得修饰的变量或者函数只在当前文件可见，因此可以避免出现多个文件中重名的问题。因此我写在 synch.c 中的函数中使用了静态修饰的变量，比如 ready\_list，所以会出现没有定义的报错信息。

同时，在本次实验中我学会使用了 pintos 中的 debug 工具。由于本次实验刚写完的时候，运行总是出现断言错误，但是又不知道是在什么地方调用了断言所处的函数，因此很难去发现问题出现的原因。后来我在网上阅读 pintos 的说明文档时发现 pintos 是有内置的 debug 函数的，其实定义和实现分别在 debug.h 和 debug.c 中。比如我这次 debug\_backtrace\_all() 和 debug\_backtrace() 这两个函数，分别是用来追踪函数调用栈的，前者是输出当前线程和就绪队列中的所有线程的调用栈，后者是输出当前运行线程的调用栈。这两个函数可以放在代码中的任何地方，一旦执行到这两个函数，就输出之前所有的调用栈。比如我这次本来总是在 list\_insert 中的一个断言出错，所以我在这个断言之前插入一个 debug\_backtrace() 函数，输出之前的调用栈。但是输出的调用栈是 16 进制的机器码，很难读懂，因此可以使用 pintos 指令中的 backtrace xxx 指令，将机器码转换成对应的代码行。这样就可逐层地查看每一级函数调用，从而发现自己在哪个函数出错。比如本次我就是通过 debug\_backtrace 发现了在链表初始化时的一个错误。

本次实验还有一点是关于中断的开始于关闭，本次实验中涉及到的函数比较多，随时注意开启与关闭中断是一个好习惯。虽然在原始的版本中，sema\_up 和 sema\_down 都是没有关中断的。而且本次函数中还有类似于优先级捐赠这一类函数，应用到了 list\_sort 这一类函数，更需要保证中断关闭来维持原子操作。之前我可能是在某个函数中忘记关中断，所以造成了异常中断，使得测试无法进行。后来是在所有操作中都保证了原子操作，才修复了这一问题。