

中山大学本科生实验报告

(2018 学年春季学期)

课程名称: Operationg System

任课教师: 饶洋辉

年级+班级	2016 级	专业 (方向)	信息与计算科学
学号	16339038	姓名	舒展

1. 实验目的

- 学习并掌握什么是优先级反转
- 明确为什么要避免优先级反转
- 了解线程加锁的五种情况并实现正确的优先级捐赠
- 了解捐赠状态下修改优先级的三种情况并实现正确的优先级修改

2. 实验过程

(一) Test 源文件分析

- priority-donate-one

测试目的:

一个初始优先级为 31 的主线程拥有一把锁，让两个优先级分别为 32 和 33 的子线程来申请这把锁。测试通过输出中间信息，来查看主线程优先级的变化以及主线程与子线程的调度顺序是否正确。这个测试主要考察一把锁时的优先级捐赠情况，以及锁的 waiters 队列中线程是否按照优先级排好序。

过程分析:

主线程 test_priority_donate_one:

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

首先是一句断言，保证该主线程的初始优先级为 31（这样才会被接下来创建的两个子线程给抢占）。

```
lock_init (&lock);
lock_acquire (&lock);
```

- 初始化开头创建的锁
- 并调用 lock_acquire 函数让主线程去申请这把锁。因为这把锁的 holder 为空，所以主线程可以马上顺利地拥有这把锁。（lock_acquire 函数会在代码分析部分详细讲述）

```
thread_create ("acquire1", PRI_DEFAULT + 1, acquire1_thread_func, &lock);
```

接着调用 `thread_create` 函数创建了一个优先级为 32 的子线程。因为该子线程优先级比主线程要高，所以马上会发生优先级抢占，该子线程拿到 cpu 资源，开始执行 `acquire1_thread_func` 函数。

下面进入 `acquire1_thread_func` 函数进行分析：

```
lock_acquire (lock);
```

调用 `lock_acquire` 方法，为子线程申请该把锁。由于锁此时正被主线程所拥有，所以子线程 1 会被锁给阻塞掉，同时将自己的优先级 31 捐赠给主线程。

回到主线程：

```
msg ("This thread should have priority %d. Actual priority: %d.",  
     PRI_DEFAULT + 1, thread_get_priority ());
```

输出该信息来判断是否发生优先级捐赠，此时主线程的实际优先级应该是 31

```
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 ());
```

- 接着又创建一个优先级为 33 的子线程，马上发生优先级抢占；子线程 2 调用 `acquire2_thread_func` 函数来申请锁，被阻塞，同时发生优先级捐赠。
- 主线程输出的实际优先级应该变为 33。

```
lock_release (&lock);
```

主线程释放该锁（`lock_release` 函数会在代码分析部分详细讲述），优先级回到原先的 31。这时锁的 `waiters` 队列中优先级最高的子线程 2 会被马上唤醒，并发生优先级抢占。抢占到 cpu 资源的子线程 2 会执行晚 `lock_acquire` 函数，成功申请到锁，并继续执行 `acquire2_thread_func` 函数。

进入 `acquire2_thread_func` 函数：

```
lock_acquire (lock);  
msg ("acquire2: got the lock");  
lock_release (lock);  
msg ("acquire2: done");
```

- 子线程 2 被唤醒后，输出语句 “`acquire2: got the lock`”
- 然后调用 `lock_release` 函数释放锁，子线程 1 会被唤醒，放入 `ready_list`；但由于子线程 1 优先级低于子线程 2，所以不会发生优先级抢占。
- 子线程 2 继续输出语句 “`acquire2: done`”，然后结束线程。子线程 1 得以拿到 cpu 资源，并获得锁。

进入 `acquire1_thread_func` 函数：

```
lock_acquire (lock);
msg ("acquire1: got the lock");
lock_release (lock);
msg ("acquire1: done");
```

同样，子线程 1 输出一条语句，然后释放锁，接着输出第二条语句，结束线程。

子线程 1、2 都运行结束后，主线程才能有机会重新拿到 cpu 资源：

```
msg ("acquire2, acquire1 must already have finished, in that order.");
msg ("This should be the last line before finishing this test.");
```

主线程输出这两条语句，然后结束测试。

结果分析：

```
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 order.
(priority-donate-one) This should be the last line before finishing this test.
(priority-donate-one) end
Execution of 'priority-donate-one' complete.

shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ =====
```

如图，输出语句的顺序与上述分析的一致，测试通过。

• priority-donate-multiple

测试目的：

一个初始优先级为 31 的主线程拥有两把锁，然后创建两个优先级分别为 32 和 33 的子线程，让它们分别申请这两把锁。通过输出中间信息来查看主线程的优先级变化以及各个线程之间的调度顺序是否正确。这个测试主要考察课件（一）上面所提到的线程加锁的五种情况的情况五。

过程分析：

主线程 test_priority_donate_multiple：

```
lock_init (&a);
lock_init (&b);

lock_acquire (&a);
lock_acquire (&b);
```

- 主线程首先初始化了两个锁 a, b,
- 然后调用 lock_acquire 函数来依次申请这两把锁。因为这两把锁的 holder 此时都为空，所以主线程可以立即成功申请到这两把锁。

```
thread_create ("a", PRI_DEFAULT + 1, a_thread_func, &a);
```

接着调用 `thread_create` 函数创建一个优先级为 32 的子线程 1。因为该子线程的优先级比主线程的优先级高，所有创建后会马上发生优先级抢占；该子线程拿到 cpu 资源，然后执行 `a_thread_func` 函数。

下面进入 `a_thread_func` 函数：

```
lock_acquire (lock);
```

子线程先调用 `lock_acquire` 函数，申请锁 a。由于此时锁 a 已经被主线程所拥有，所以子线程会被锁 a 阻塞，进入锁 a 的 waiters 队列，并将自己的优先级 32 捐赠给主线程。

回到主线程：

```
msg ("Main thread should have priority %d. Actual priority: %d.",  
     | | PRI_DEFAULT + 1, thread_get_priority ());
```

主线程输出该语句来检查是否发生了正确的优先级捐赠。

```
thread_create ("b", PRI_DEFAULT + 2, b_thread_func, &b);  
msg ("Main thread should have priority %d. Actual priority: %d.",  
     | | PRI_DEFAULT + 2, thread_get_priority ());
```

- 同样，主线程创建一个优先级为 33 的子线程 2；创建后子线程 2 马上抢夺主线程的 cpu 资源，执行 `b_thread_func` 函数；在 `b_thread_func` 函数中，子线程 2 申请锁 b，被锁 b 阻塞并将自己的优先级 33 捐赠给主线程。
- 回到主线程，输出第二条语句来检查是否发生正确的优先级捐赠。

```
lock_release (&b);
```

接着主线程释放锁 b，优先级降为 32；锁 b 的 waiters 队列中唯一的线程子线程 2 被唤醒，马上发生优先级抢占。子线程 b 拿到 cpu 资源后，会执行完 `lock_acquire` 函数，拿到锁 b。

子线程 2 继续执行 `b_thread_func` 函数：

```
lock_acquire (lock);  
msg ("Thread b acquired lock b.");  
lock_release (lock);  
msg ("Thread b finished.");
```

- 子线程 2 输出 “Thread b acquire lock b.”，
- 调用 `lock_release` 释放锁 b。此时就绪队列中并没有优先级更高的线程
- 子线程 2 继续输出第二条信息，结束运行。

回到主线程：

```
msg ("Thread b should have just finished.");  
msg ("Main thread should have priority %d. Actual priority: %d.",  
     | | PRI_DEFAULT + 1, thread_get_priority ());
```

输出这两条语句来确定之间的调度关系以及优先级捐赠是否正确。这个时候主线程的优先级应该为 32.

```
lock_release (&a);
msg ("Thread a should have just finished.");
msg ("Main thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT, thread_get_priority());
```

- 接着主线程释放锁 a。与前面分析的一样，主线程的优先级降为 31，子线程 1 被唤醒，抢占主线程的 cpu，获得锁 a，继续执行完 a_thread_func 函数
- 然后主线程再继续执行，输出两条信息，结束运行。

结果分析：

```
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.

shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ =====
```

如图，输出语句与上述分析的一致，测试通过。

失败原因：

下面分析一下在未修改源码前，该测试的输出结果已经失败的原因。

在未修改代码前，输出结果如下：

```
(priority-donate-multiple) begin
(priority-donate-multiple) Main thread should have priority 32. Actual priority
: 31.
(priority-donate-multiple) Main thread should have priority 33. Actual priority
: 31.
(priority-donate-multiple) Thread b should have just finished.
(priority-donate-multiple) Main thread should have priority 32. Actual priority
: 31.
(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
```

- 在未修改代码前，各线程的调度顺序与修改后的调度顺序是一样的。这是之前实现了优先级抢占的缘故。
- 创建子线程 a 后发生优先级抢占，线程 a 申请锁 a，被阻塞；回到主线程，此时主线程的优先级应该为 32，因为线程 a 会将自己的优先级捐赠给主线程；但实际由于没有实现优先级捐赠功能，所以主线程优先级仍然为 31。下面创建子线程 b 同理。

- 当主线程释放 lock b 后，线程 b 抢占主线程的 cpu，完成并结束运行；主线程重新拿到 cpu，此时它的优先级应该为 32，但实际上仍然为 31；这是应该主线程在释放锁后，没有去检查它的锁队列，所以主线程的优先级自始自终一直为 31 没有变过。释放锁 a 时同理。

• priority-donate-multiple2

测试目的：

让一个初始优先级为 31 的主线程拥有两把锁，然后创建两个优先级为 34 和 36 的子线程来分别申请这两把锁，同时再创建了一个无需申请锁的优先级为 32 的子线程；通过输出中间信息来判断线程间调度顺序以及优先级捐赠是否正确。这个测试是 priority-donate-multiple2 测试的改进版，调度顺序要更加复杂。

过程分析：

主线程 test_priority_donate_multiple2：

```
lock_init (&a);
lock_init (&b);

lock_acquire (&a);
lock_acquire (&b);
```

初始化两把锁，然后调用 lock_acquire 函数让主线程分别申请这两把锁。因为锁 a, b 的 holder 都是空，所以主线程可以马上顺利地拥有这两把锁。

```
thread_create ("a", PRI_DEFAULT + 3, a_thread_func, &a);
```

接着调用 thread_create 函数，创建了一个优先级为 34 的子线程 a，发生优先级抢占，线程 a 拿到 cpu 资源，开始执行 a_thread_func 函数。

进入 a_thread_func 函数：

```
lock_acquire (lock);
```

线程 a 首先申请锁 a，被锁 a 阻塞并将自己的优先级 34 捐赠给主线程。

回到主线程：

```
msg ("Main thread should have priority %d. Actual priority: %d.",
     | | PRI_DEFAULT + 3, thread_get_priority ());
```

输出该信息以确定主线程此时的优先级，应该为 34

```
thread_create ("c", PRI_DEFAULT + 1, c_thread_func, NULL);
```

然后主线程创建了一个优先级为 32 的子线程 c。由于此时主线程的优先级已经为 34 了，所以线程 c 进入就绪队列，并不会发生优先级抢占。

```
thread_create ("b", PRI_DEFAULT + 5, b_thread_func, &b);
msg ("Main thread should have priority %d. Actual priority: %d.",
     | | PRI_DEFAULT + 5, thread_get_priority ());
```

主线程继续创建了一个优先级为 36 的子线程 b，发生优先级抢占，线程 b 申请锁 b，被阻塞，将自己的优先级 36 捐赠给主线程；

- 然后回到主线程，输出信息，确定此时主线程的优先级为 36.

```
lock_release (&a);
```

主线程释放锁 a，优先级仍然保持 36 不变；锁 a 的申请者线程 a 被唤醒，但不会发生优先级抢占，进入就绪队列等待。

```
msg ("Main thread should have priority %d. Actual priority: %d.",
     | | PRI_DEFAULT + 5, thread_get_priority ());
```

主线程继续输出该条信息，确定此时主线程的优先级仍为 36.

```
lock_release (&b);
```

主线程释放锁 b，优先级降为最初的优先级 31；锁 b 的申请者线程 b 被唤醒，马上发生优先级抢占，线程 b 拿到 cpu 资源，继续执行完 lock_acquire 函数，拿到锁 b。

线程 b 继续执行 b_thread_func:

```
lock_acquire (lock);
msg ("Thread b acquired lock b.");
lock_release (lock);
msg ("Thread b finished.");
```

- 输出信息“Thread b acquired lock b.”，
- 释放锁 b
- 最后输出“Thread b finished.”结束运行。

然后按照 ready_list 中优先级的大小，线程 a 会拿到 cpu 资源，执行完 lock_acquire 函数，成功拿到锁 a，然后继续执行 a_thread_func:

```
lock_acquire (lock);
msg ("Thread a acquired lock a.");
lock_release (lock);
msg ("Thread a finished.");
```

- 输出信息“Thread a acpuired lock a.”
- 释放锁 a
- 最后输出“Thread a finished.”结束运行。

然后按照 ready_list 中优先级的大小，线程 c 拿到 cpu 资源，开始执行 c_thread_func 函数：

```
c_thread_func (void *a_ UNUSED)
{
    msg ("Thread c finished.");
}
```

输出信息“Thread c finished.”，结束运行。

主线程此时终于可以再次拿到cpu资源：

```
msg ("Threads b, a, c should have just finished, in that order.");
msg ("Main thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT, thread_get_priority ());
```

输出两条信息，结束测试。注意此时主线程的优先级应该为31.

结果分析：

```
Executing 'priority-donate-multiple2':
(priority-donate-multiple2) begin
(priority-donate-multiple2) Main thread should have priority 34. Actual priorit
y: 34.
(priority-donate-multiple2) Main thread should have priority 36. Actual priorit
y: 36.
(priority-donate-multiple2) Main thread should have priority 36. Actual priorit
y: 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 o
rder.
(priority-donate-multiple2) Main thread should have priority 31. Actual priorit
y: 31.
(priority-donate-multiple2) end
Execution of 'priority-donate-multiple2' complete.

shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ =====
```

如上图，信息输出与分析的一致，测试通过。

• priority-donate-nest

测试目的：

让一个低优先级的线程拥有锁A；让一个中等优先级的线程拥有锁B，并让它在申请锁A的过程中被阻塞；再让一个高优先级的申请锁B，实现优先级的嵌套捐赠。测试通过输出中间信息来检查线程调度顺序及嵌套捐赠行为是否正确。

过程分析：

主线程 test_priority_donate_nest (初始优先级为31)：

```
lock_acquire (&a);
```

首先让主线程（“Low thread”）拥有锁a。

```
thread_create ("medium", PRI_DEFAULT + 1, medium_thread_func, &locks);
```

然后创建一个优先级为 32 的“medium thread”；发生优先级抢占，medium thread 拿到 cpu 资源。

medium thread 执行 medium_thread_func 函数：

```
lock_acquire (locks->b);
lock_acquire (locks->a);
```

- medium thread 首先顺利申请到锁 b
- 然后再申请锁 a；由于锁 a 已经被 low thread 拥有，所以 medium thread 会被锁 a 阻塞，进入锁 a 的 waiters 队列，同时讲自己的优先级 32 捐赠给 low thread。

回到主线程（low thread）：

```
msg ("Low thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT + 1, thread_get_priority ());
```

输出该条信息以确认优先级捐赠正确，此时 low thread 的优先级应为 32.

```
thread_create ("high", PRI_DEFAULT + 2, high_thread_func, &b);
```

然后在主线程中接着创建一个优先级为 33 的线程 high thread，同样发生优先级抢占，high thread 拿到 cpu 资源。

high thread 执行 high_thread_func 函数：

```
lock_acquire (lock);
```

high thread 申请锁 b，因为此时锁 b 已经被 medium thread 所拥有，所以 high thread 会被锁 b 阻塞，进入锁 b 的 waiters 队列，同时发生嵌套捐赠，将自己的优先级 33 捐赠给 medium thread 及 low thread。

回到 low thread：

```
msg ("Low thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT + 2, thread_get_priority ());
```

输出该信息以确定嵌套捐赠行为是正确，此时 low thread 的优先级应该为 33。

```
lock_release (&a);
```

low thread 释放锁 a，优先级降为 31. medium thread 于是被唤醒，发生优先级抢占，拿到 cpu 资源。

medium thread 执行完 lock_acquire 函数，成功拿到锁 a（此时该线程同时拥有锁 a 和锁 b），继续执行 medium_thread_fun 函数：

```
msg ("Medium thread should have priority %d. Actual priority: %d.",  
     | | PRI_DEFAULT + 2, thread_get_priority ());  
msg ("Medium thread got the lock.");
```

medium thread 输出信息以确定发生了正确的优先级捐赠行为。因为 high thread 被锁 b 阻塞，所以 medium thread 此时的优先级应该为 33.

```
lock_release (locks->a);
```

medium thread 释放锁 a；由于锁 a 的 waiters 队列为空，所以不会唤醒任何队列。

```
lock_release (locks->b);
```

medium thread 接着释放锁 b，优先级降为 32；锁 b 的 waiters 队列中的 high thread 线程被唤醒，立即发生优先级抢占。high thread 拿到 cpu 资源。

high thread 执行完 lock_acquire 函数，顺利拿到锁 b，然后继续执行 high_thread_func 函数：

```
lock_acquire (lock);  
msg ("High thread got the lock.");  
lock_release (lock);  
msg ("High thread finished.");
```

- High thread 输出 “High thread got the lock.”
- 然后释放锁 b，锁 b 的 waiters 队列为空，并不会唤醒任何线程，high thread 继续运行；
- 输出信息 “High thread finished.”，然后结束运行。

medium_thread_func 有机会拿到 cpu，继续执行：

```
msg ("High thread should have just finished.");  
msg ("Middle thread finished.");
```

medium thread 输出信息，表示 high thread 刚刚执行完，然后结束运行。

主线程 (low thread) 终于有机会再次拿到 cpu 资源：

```
msg ("Medium thread should just have finished.");  
msg ("Low thread should have priority %d. Actual priority: %d.",  
     | | PRI_DEFAULT, thread_get_priority ());
```

low thread 输出信息，表示 medium thread 刚刚执行完，并确认当前优先级正确。此时 low thread 的优先级应该为初始的 31. 输出完信息后，主线程结束运行，测试完成。

结果分析:

```
Executing 'priority-donate-nest':  
ASSERTION (num_threads) failed.  
(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:  
33.  
(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.  
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ ======  
=====
```

如上图，输出信息与分析的一致，测试通过。

• priority-donate-sema

测试目的:

创建一个 low thread，使其拥有一把锁，让它在对一个信号量进行 P 操作时被阻塞；创建一个 medium thread，使其在对同一个信号量进行 P 操作时被阻塞；创建一个 high thread，让它请求锁，被锁阻塞并发生优先级捐赠。

接下来让主线程对信号量进行 V 操作，唤醒 low thread;low thread 释放锁，唤醒 high thread;high thread 对信号量进行 V 操作，唤醒 medium thread;然后 high thread, medium thread, low thread 与主线程依次介绍运行。

通过输出中间信息以检查线程之间的调度顺序与优先级捐赠行为是否正确。

过程分析:

先来简单看下结构体 lock_and_sema:

```
struct lock_and_sema  
{  
    struct lock lock;  
    struct semaphore sema;  
};
```

这个结构体中有一个锁对象，还有一个信号量对象（注意锁中本身也有一个信号量对象，要与之区别开）。

主线程 teset_priority_donate_sema:

```
lock_init (&ls.lock);  
sema_init (&ls.sema, 0);
```

首先初始化 ls 中的锁，然后将 ls 中的信号量的值初始为 0.

```
thread_create ("low", PRI_DEFAULT + 1, l_thread_func, &ls);
```

创建一个 low thread, 优先级为 32, 发生优先级抢占, low thread 拿到 cpu 资源。

Low thread 开始执行 l_thread_func 函数:

```
lock_acquire (&ls->lock);
msg ("Thread L acquired lock.");
sema_down (&ls->sema);
```

- Low thread 顺利拿到 ls 中的锁
- 输出信息 “Thread L acquired lock.” ;
- 然后对 ls 中的信号量进行 P 操作。由于信号量的初始值为 0, 所以 low thread 会被该信号量阻塞掉。

回到主线程:

```
thread_create ("med", PRI_DEFAULT + 3, m_thread_func, &ls);
```

创建了一个 med thread, 优先级为 34, 发生优先级抢占, med thread 拿到 cpu 资源。

Med thread 开始执行 m_thread_func 函数:

```
sema_down (&ls->sema);
```

Med thread 对 ls 的信号量进行 P 操作, 此时信号量的值为 0, 同样被阻塞。

回到主线程:

```
thread_create ("high", PRI_DEFAULT + 5, h_thread_func, &ls);
```

创建了一个优先级为 36 的 high thread, 发生优先级抢占, high thread 拿到 cpu 资源。

High thread 开始执行 h_thread_func 函数:

```
lock_acquire (&ls->lock);
```

High thread 请求 ls 中的锁, 由于该锁已被 low thread 所拥有, 所以 high thread 被阻塞, 并将自己的优先级 36 捐赠给 low thread.

回到主线程:

```
sema_up (&ls.sema);
```

主线程对 ls 中的信号量进行 V 操作, 信号量的值升为 1, 信号量的 waiters 队列中优先级最高的 low thread (被 high thread 捐赠) 被唤醒, 发生优先级抢占, low thread 拿到 cpu 资源。

Low thread 继续执行完 P 操作，信号量的值又降为 0，然后继续执行 l_thread_func 函数：

```
msg ("Thread L downed semaphore.");
lock_release (&ls->lock);
```

- Low thread 输出 “Thread L downed semaphore.”
- 释放 ls 中的锁，优先级降为 31；High thread 被唤醒，发生优先级抢占，拿到 cpu 资源。

High thread 继续执行完 lock_acqire，成功拿到 ls 中的锁，然后继续执行 h_thread_func 函数：

```
lock_acquire (&ls->lock);
msg ("Thread H acquired lock.");

sema_up (&ls->sema);
lock_release (&ls->lock);
msg ("Thread H finished.");
```

- High thread 输出 “Thread H acquired lock.”
- 对 ls 中的信号量进行 V 操作，信号量的值升为 1，med thread 被唤醒；
- 然后 high thread 释放锁，输出信息 “Thread H finished.”，结束运行。

Med thread 拿到 cpu 资源：

```
sema_down (&ls->sema);
msg ("Thread M finished.");
```

- Med thread 继续执行完 P 操作，信号量的值又降为 0；
- 然后输出信息 “Thread M finished.”，结束运行。

Low thread 拿到 cpu 资源：

```
msg ("Thread L finished.");
```

Med thread 输出最后一条信息 “Thread L finished.”，然后结束运行。

最后回到主线程：

```
msg ("Main thread finished.");
```

输出 “Main thread finished.”，然后结束整个测试。

结果分析：

```
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.  
  
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ ======
```

如图，信息输出顺序与分析一致，测试通过。

- priority-donate-lower

测试目的：

首先让一个优先级为 31 的主线程拥有一把锁，然后创建一个优先级为 41 的子线程来申请这把锁；子线程被锁阻塞，发生优先级捐赠；然后主线程调用 `thread_set_priority` 函数来修改优先级。简而言之这个测试考察的是捐赠状态下优先级修改的：原始优先级 < 修改的优先级 < 捐赠的优先级的这种情况。测试通过输出中间信息来确认线程调度顺序以及优先级捐赠行为是否正确。

过程分析：

主线程 `test_priority_donate_lower`:

```
lock_init (&lock);  
lock_acquire (&lock);  
thread_create ("acquire", PRI_DEFAULT + 10, acquire_thread_func, &lock);
```

- 主线程首先初始化一把锁，请求并顺利拥有它
- 然后创建了一个优先级为 41 的 acquire thread，发生优先级抢占，acquire thread 拿到 cpu 资源。

acquire thread 执行 `acquire_thread_func` 函数:

```
lock_acquire (lock);
```

Acquire thread 申请这把锁，被阻塞，进入锁的 waiters 队列等待被唤醒，同时将自己的优先级 41 捐赠给主线程。

回到主线程:

```
msg ("Main thread should have priority %d. Actual priority: %d.",  
     | | PRI_DEFAULT + 10, thread_get_priority ());  
  
msg ("Lowering base priority...");
```

主线程输出第一条信息，确认优先级捐赠正确；此时主线程的优先级应该为 41。然后输出第二条信息。

```
thread_set_priority (PRI_DEFAULT - 10);
```

接着主线程调用 `thread_set_priority` 函数，想要将当前优先级设置为 21。这种情况为：原始优先级 < 修改的优先级 < 捐赠的优先级。所以主线程的 `old_priority` 成员值变为 21；`priority` 不变，仍未 41，主线程运行。

```
msg ("Main thread should have priority %d. Actual priority: %d.",
     || PRI_DEFAULT + 10, thread_get_priority ());
lock_release (&lock);
```

- 主线程输出一条信息，确认此时的优先级仍为 41。
- 然后释放锁，优先级降为 21；`acquire` thread 被唤醒，发生优先级抢占，`acquire` thread 拿到 cpu 资源。

`acquire` thread 执行完 `lock_acquire` 函数，顺利申请到锁，然后继续执行 `acquire_thread_func` 函数：

```
lock_acquire (lock);
msg ("acquire: got the lock");
lock_release (lock);
msg ("acquire: done");
```

- `acquire` thread 输出信息“`acquire: got the lock.`”
- 释放锁
- 输出信息“`acquire: done`”，结束运行。

回到主线程：

```
msg ("acquire must already have finished.");
msg ("Main thread should have priority %d. Actual priority: %d.",
     ||| PRI_DEFAULT - 10, thread_get_priority ());
```

输出信息，确认主线程此时的优先级应该为 21，然后结束运行。

结果分析：

```
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.

shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ =====
```

如上图，信息输出顺序与分析的一致，测试通过。

- priority-sema

测试目的：

这个测试主要考察的是不同优先级的线程在信号量的 waiters 队列中等待时，是否安装优先级大小排列并被唤醒。该测试乱序创建了 10 个优先级各不相同的线程，使它们被同一个信号量阻塞，然后检查它们唤醒的顺序；如果安装优先级从高到低依次唤醒，则测试通过。

过程分析：

主线程 test_priority_sema:

```
sema_init (&sema, 0);
thread_set_priority (PRI_MIN);
```

首先初始化信号量，初始值为 0。然后将自己的优先级调至最低。

```
for (i = 0; i < 10; i++)
{
    int priority = PRI_DEFAULT - (i + 3) % 10 - 1;
    char name[16];
    snprintf (name, sizeof name, "priority %d", priority);
    thread_create (name, priority, priority_sema_thread, NULL);
}
```

- 然后是 10 次循环，创建 10 个子线程。子线程的优先级的创建顺序为 27、26、25、24、23、22、21、30、29、28。
- 对于每一次循环，其创建的子线程的优先级都比主线程的优先级要高，发生优先级抢占，子线程拿到 cpu 资源。

子线程执行 priority_sema_thread 函数：

```
static void
priority_sema_thread (void *aux UNUSED)
{
    sema_down (&sema);
    msg ("Thread %s woke up.", thread_name ());
}
```

子线程对信号量进行 P 操作；由于此时信号量的值为 0，所以子线程被阻塞，按顺序插入到信号量的 waiters 队列中，然后回到主线程。如此交替调度，循环 10 次。

回到主线程：

```
for (i = 0; i < 10; i++)
{
    sema_up (&sema);
    msg ("Back in main thread.");
}
```

执行一个循环 10 次的 for 循环。对于每一次循环，主线程首先对信号量进行 V 操作，信号量的值升为 1；waiters 队列中优先级最大的线程被唤醒，发生优先级抢占，子线程拿到 cpu 资源。

子线程接着执行完 sema_down 函数，信号量的值又降为 0；然后继续执行 priority_sema_thread 函数：

```
priority_sema_thread (void *aux UNUSED)
{
    sema_down (&sema);
    msg ("Thread %s woke up.", thread_name ());
}
```

子线程接着输出一条信息，然后结束运行。

回到主线程：

```
for (i = 0; i < 10; i++)
{
    sema_up (&sema);
    msg ("Back in main thread.");
}
```

主线程接着输出一条信息“Back in main thread.”，然后一次循环结束，开始下一次循环。主线程和子线程如此交替调度，如此循环 10 次，然后结束主线程，整个测试结束。

结果分析：

```
Executing 'priority-sema':
(priority-sema) begin
(priority-sema) Thread priority 30 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 29 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 28 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 27 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 26 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 25 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 24 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 23 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 22 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 21 woke up.
(priority-sema) Back in main thread.
(priority-sema) end
Execution of 'priority-sema' complete.

shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ ===
```

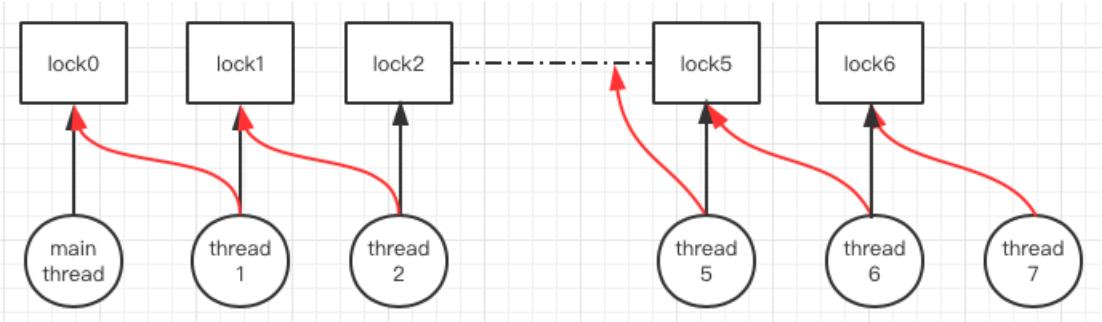
如上图，子线程安装优先级从高到低的顺序被唤醒，与分析的一致，测试通过。

- priority-donate-chain

测试目的:

- 该测试主要考察是否正确实现嵌套捐赠。
- 主线程首先按照 0-6 的编号创建了 7 把锁；然后 $i = 1 \text{ to } 7$ 进行 7 次循环，对于每次：循环按照 $i * 3$ 的规律创建一个子线程，让它先（成功）申请到锁 i ，然后再申请锁 $i-1$ ，发生嵌套捐赠现象，同时被阻塞；另外还要创建一个优先级为 $i * 3 - 1$ 的子线程 interloper，其作用是在前面创建的子线程结束运行后输出提示信息。注意该 interloper 紧接着在同一循环创建的第一个子线程结束运行后被调用，会在下面分析到。
- 结束上面的循环后，主线程会释放它拥有的锁 0，优先级降为初始优先级。然后子线程 1 会被唤醒，成功申请到锁 0；接着释放锁 0，再释放锁 1，优先级降为初始优先级。然后子线程 2 被唤醒，如此循环下去，直到子线程 7 被唤醒，成功申请到锁 6。
- 然后子线程 7 释放锁 6，结束运行。其对应的 interloper thread 拿到 cpu 输出提示信息，然后子线程 6 被唤醒……如此循环，直到子线程 1 结束运行。最后主线程再拿到 cpu 资源，结束整个测试。
- 如果输出信息顺序与上面分析的一致，则测试通过。

• **示意图如下：**



每个线程均现执行黑色箭头再执行红色箭头。

过程分析:

首先简单分析一下 lock_pair 结构体：

```
struct lock_pair
{
    struct lock *second;
    struct lock *first;
};
```

结构体中就两个 lock*类型指针。First 指向第 i 把锁；Second 指向第 $i-1$ 把锁。

主线程 test_priority_donate_chain：

```
struct lock locks[NESTING_DEPTH - 1]; //创建了7把锁
struct lock_pair lock_pairs[NESTING_DEPTH]; //8个lock_pairs
```

创建了 7 把锁以及 8 个 lock_pair 类型变量。

```
thread_set_priority (PRI_MIN);
```

然后主线程将自己的优先级调至最低，为的是确保会被后面创建的子线程抢占。

```
for (i = 0; i < NESTING_DEPTH - 1; i++)
    lock_init (&locks[i]);
```

循环初始化这 7 把锁。

```
lock_acquire (&locks[0]);
msg ("%s got lock.", thread_name ());
```

- 主线程成功申请到锁 0
- 输出信息“main got lock.”

然后进入一个 for 循环 ($i = 1$ to 7 , 7 次循环) :

```
thread_priority = PRI_MIN + i * 3;
lock_pairs[i].first = i < NESTING_DEPTH - 1 ? locks + i: NULL;
lock_pairs[i].second = locks + i - 1;
```

- `thread_priority` 设置为 $i * 3$;
- `first` 指向第 i 把锁（注意第 7 次循环时指向 `NULL`）；
- `second` 指向第 $i-1$ 把锁。

```
thread_create (name, thread_priority, donor_thread_func, lock_pairs + i);
```

然后传如上面的参数创建子线程；发生优先级抢占，子线程拿到 cpu 资源开始执行 `donor_thread_func` 函数。

`donor_thread_func` 函数:

```
if (locks->first) // 先申请后面的锁
    lock_acquire (locks->first);

lock_acquire (locks->second); // 然后申请前面的锁
```

- 首先做一个判断，`first` 不是空（说明不是子线程 7），则申请 `first` 指向的锁（锁 i ）。因为锁 i 的 `holder` 此时为空，所以可以立即成功申请到锁。
- 申请锁 $i-1$ ，被阻塞，并进行优先级嵌套捐赠；主线程一直到子线程 $i-1$ 的优先级应该都升为 $i * 3$ 。

回到主线程:

```
msg ("%s should have priority %d. Actual priority: %d.",
    | thread_name (), thread_priority, thread_get_priority ());
```

输出该信息以确认主线程的优先级正确，应为 $i * 3$ 。

```
thread_create (name, thread_priority - 1, interloper_thread_func, NULL);
```

然后再循环的最后，再创建一个优先级为 $i * 3 - 1$ 的子线程；由于其优先级比此时主线程的优先级低，所以不会发生优先级抢占。

- 然后再进入下一次循环，如此反复，直到最后一次循环子线程 7 申请锁 6 然后被阻塞。

```
lock_release (&locks[0]);
```

然后主线程释放它所拥有的锁：锁 0，优先级降为最初的 0。锁 0 的 waiters 队列中只有子线程 1，子线程 1 被唤醒，发生优先级抢占，拿到 cpu 资源。

子线程 1 执行完 lock_acquire 函数，成功拿到锁 0，继续执行 donor_thread_func 函数：

```
lock_acquire (locks->second); // 然后申请前面的锁
msg ("%s got lock", thread_name ());

lock_release (locks->second); // 先释放前面的锁
msg ("%s should have priority %d. Actual priority: %d",
     thread_name (), (NESTING_DEPTH - 1) * 3,
     thread_get_priority ());
```

- 输出提示信息“thread 1 got lock”
- 释放锁 0
- 输出信息，以确认子线程 1 此时的优先级仍为 21.

```
if (locks->first) // 然后释放后面的锁
    lock_release (locks->first);
```

- 首先判断 first 是否为空
- 然后释放锁 i，子线程 1 优先级降为初始优先级 3，子线程 2 被唤醒。然后子线程 2 拿到 cpu 资源，重复上面的过程，一直到子线程 7 被唤醒。

```
msg ("%s finishing with priority %d.", thread_name (),
      thread_get_priority ());
```

子线程 7 释放了锁 6 后，输出此信息，确认优先级为最初的 21，然后结束运行。

此时 ready_list 中优先级最高的线程应该是子线程 7 对应的 interloper：

```
interloper_thread_func (void *arg_ UNUSED)
{
    msg ("%s finished.", thread_name ());
}
```

interloper 输出信息，以确认子线程 7 已经运行完，然后结束运行。

- 此时 ready_list 中优先级最高的线程应该是子线程 6，子线程 6 拿到 cpu 资源，重复上面的过程，一直到 interloper 1 结束运行。

回到主线程：

```
msg ("%s finishing with priority %d.", thread_name (),  
     thread_get_priority ());
```

最后输出该信息，确认主线程的优先级为 0，然后结束整个测试。

结果分析:

```
(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.
```

如图，输出结果与分析一致，测试通过。

(二) 实验思路与代码分析

(1) thread 结构体

在 thread 结构体中添加了如下成员变量：

```
int old_priority; /* v3 The priority before donated*/
bool donated; /* v3 Donated or not*/
struct lock *blocked; /* 指向阻塞当前线程的锁 */
struct list locks; /* 当前线程占有的锁队列 */
```

- old_priority 用于记录被捐赠前线程的优先级，恢复线程初始优先级时需要用到
- donated 记录线程当前是否被捐赠了优先级
- blocked 为指向阻塞当前线程的锁
- locks 为线程的锁队列，在更新线程的优先级时起到至关重要的作用

在 init_thread 函数中初始化这些变量：

```
t->old_priority = PRI_MIN;
t->donated = false;
t->blocked = NULL;
list_init(&t->locks);
```

(2) lock 结构体

在 lock 结构体中添加如下成员变量：

```
struct list_elem elem;      /* 锁的小弟 */
int priority;              /* 锁的优先级 */
```

- elem：锁的小弟，将锁放入锁队列排队或从队列中查找锁时会用到
- priority：锁的优先级，定义锁的 waiters 队列中优先级最高的线程

在 lock_init 函数中对这些变量进行初始化：

```
lock->holder = NULL;
lock->priority = PRI_MIN; //锁的优先级的初始化
```

(3) lock_acquire 函数：

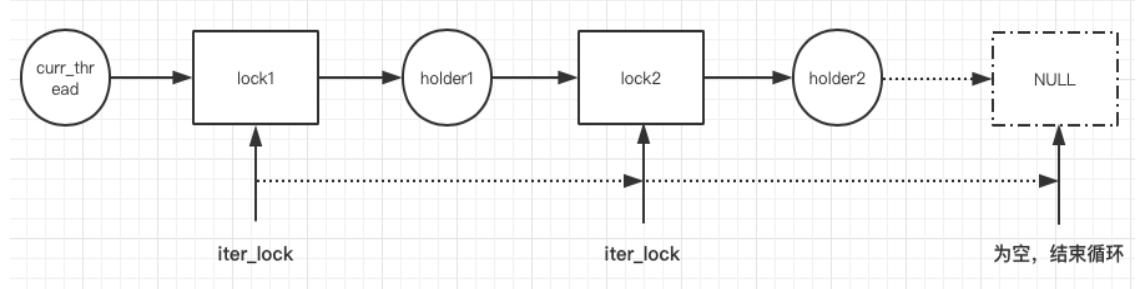
- a、当线程调用 lock_acquire 函数申请锁的时候，首先应该考虑的是锁是否已经被另外的线程所拥有了；所以应该检查锁的 holder 是否为空：为空则可以正常申请锁；否则要进行优先级的嵌套捐赠

```
struct lock *iter_lock = lock;
struct thread *curr_thread = thread_current();

if (lock->holder != NULL)
{
    thread_current ()->blocked = lock;
    while (iter_lock != NULL && iter_lock->priority < curr_thread->priority)
    {
        iter_lock->priority = curr_thread->priority;
        update_thread_priority (iter_lock->holder); // 在该函数里面设置好donated和
        iter_lock = iter_lock->holder->blocked;
    }
}
```

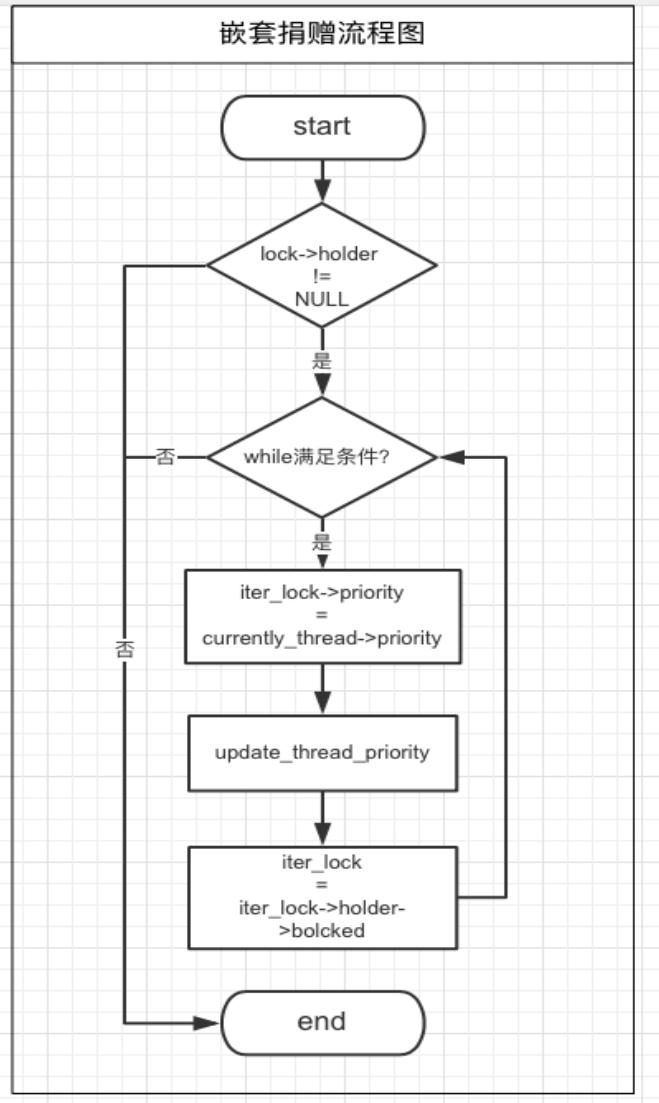
- 先进行一个 if 判断，如果 holder 不为空，则要进行优先级嵌套捐赠
- 因为当前线程会被这把锁阻塞，所以在嵌套捐赠前要设置好当前线程 blocked 变量的值

- iter_lock 表示递归捐赠的过程中，指向的锁；curr_thread 即当前要申请的锁的线程，它将自己的优先级捐赠给前面的锁和线程的优先级，示意图如下：



- 递归捐赠的过程如下：
 - 首先判断 iter_lock 不为空，然后 iter_lock 的优先级一定要低于 curr_thread 的优先级的（否则就不需要继续捐赠了）
 - 进入 while 循环，curr_thread 将自己的优先级捐赠给 iter_lock 的优先级；
 - 调用 update_thread_priority 函数来检查并更新 iter_lock 的 holder 的优先级。这里要明确说明一下为什么不直接更新 holder 的优先级：因为 iter_lock 的优先级虽然更新了，但是它的优先级可能仍然低于 holder 的优先级，这时就不需要捐赠了。update_thread_priority 的功能以及具体实现在下面会详细讲解。
 - 最后更新 iter_lock 指针的指向位置，开始下一轮循环。

流程图如下：



b、处理完 b 操作后进行 P 操作

c、P 操作结束，表示该线程可以被允许成功申请到锁；于是进行一些变量赋值等后续工作。

```

lock->holder = thread_current ();

if (!list_empty (&(lock->semaphore.waiters)))
{
    //list_sort (&(lock->semaphore.waiters), cmp_thread, NULL);
    lock->priority = list_entry (list_front (&(lock->semaphore.waiters)),
                                struct thread, elem)->priority;
}
else
    lock->priority = PRI_MIN;

thread_current ()->blocked = NULL;
list_insert_ordered (&thread_current ()->locks,
                    &lock->elem, cmp_lock, NULL);
  
```

- lock 的 holder 成员要指向 thread_current()
- if-else 语句用于更新锁的优先级。若当前线程曾被 lock 阻塞，则 P 操作完成后，会从 lock 的 waiters 队列中被移除，所以需要更新锁的优先级。如果锁的 waiters 队列不为空，则将锁的优先级调至 waiters 队列中最大的优先级；否则直接将优先级设置为最小
- 此时没有锁阻塞线程了，所以 blocked 变量指向 NULL
- 将线程新获得的锁按照优先级插入到线程的锁队列中
- 需要注意的是，上图这部分操作，必须设置为原子操作，否则测试不会通过（具体原因还在思考中）。

(4) cmp_lock 函数：

注意还需要在 thread.c 中多定义一个 cmp_lock 函数，用以比较锁的优先级的大小，即使它的形式与 cmp_thread 一模一样：

```
bool
cmp_lock(const struct list_elem *lock1, const struct list_elem *lock2,
{
    if (list_entry(lock1, struct lock, elem)->priority >
        list_entry(lock2, struct lock, elem)->priority)
        return true;
    else
        return false;
}
```

(5) update_thread_priority 函数：

上面处理嵌套捐赠的代码中，用到了自定义的 update_thread_priority 函数，它在两种情况下会被调用：(1) 当线程的锁队列中，有锁的优先级发生了改变（有其他的线程申请该把锁），检查（并更新）线程的优先级。若线程的优先级是第一次被捐赠，还有设置其 donated 的值为 true。(2) 当线程从锁队列中释放了一把锁，检查（并更新）线程的优先级。若线程的优先级要恢复到初始优先级，还需要设置其 donated 为 false.

```
int max_priority = PRI_MIN;

if (!list_empty (&t->locks))
{
    list_sort (&t->locks, &cmp_lock, NULL);
    if (list_entry (list_front (&t->locks), struct lock, elem)->priority
        > max_priority)
        max_priority = list_entry (list_front (&t->locks),
                                   struct lock, elem)->priority;
}
```

- 首先获得线程的锁队列中，最大的优先级（如果锁队列为空，则 max_priority 值为 0）。
- 注意到，在调用 list_front 函数前，需要先 sort 一次线程的锁队列。这是因为纵使线程在获得锁使，是按照优先级大小插入锁队列的，但是在嵌套捐赠使，锁队列中个别锁的优先级会发生改变，从而打乱队列的顺序。

```

if (max_priority > t->old_priority)
{
    if (t->donated == false)
    {
        t->donated = true;
        t->old_priority = t->priority;
    }
    t->priority = max_priority;
}
else
{
    if (t->donated == true)
    {
        t->donated = false;
        t->priority = t->old_priority;
    }
}

```

- 如果锁队列中，最大的优先级要比线程的 old_priotiy 大，则更新线程的优先级。同时判断 donated 是否为 false，若为 false，说明没发生捐赠，要更新线程 donated 和 old_priority 的值。
- 如果锁队列中的优先级都比线程的优先级小，并且线程的优先级之前被捐赠过，那么说明线程释放了一把锁，并且锁队列中剩下的锁的优先级都要低于线程最初的优先级，故线程的优先级需要恢复到 old_priority.

```

list_sort (&ready_list, &cmp_thread, NULL);

```

- 因为线程的优先级可能发生了改变，会打乱 ready_list 中的顺序，所以最后还需要 sort 一下 ready_list.

(6) sema_down 函数：

```

while (sema->value == 0)
{
    /* v3 waiters也要按照优先级排列 */
    list_insert_ordered (&sema->waiters, &thread_current ()->elem,
                        (list_less_func*) cmp_thread, NULL);
    thread_block ();
}

```

仅作一处修改，将线程按照优先级大小插入 waiters 队列中。

(7) lock_release 函数：

```

lock->holder = NULL;
list_remove (&lock->elem);
update_thread_priority (thread_current ());

```

在 V 操作前进行一些前续工作。

- 锁被释放，其 holder 指向 NULL
- 将锁从线程的锁队列中移除
- 调用 update_thread_priority 函数检查（更新）线程的优先级

- 同样，这部分操作也需要设置为原子操作。

(8) sema_up 函数:

```
struct thread *max = NULL;
if (!list_empty (&sema->waiters))
{
    list_sort (&sema->waiters, cmp_thread, NULL);
    max = list_entry (list_pop_front (&sema->waiters),
                      struct thread, elem);
    thread_unblock (max);
}
```

- 在唤醒线程之前需要 sort 一下 waiters 队列，因为嵌套捐赠可能会改变 waiters 队列中线程的优先级，从而打乱队列的顺序。

```
sema->value++;
if (max != NULL && max->priority > thread_current ()->priority)
| thread_yield ();
```

- 只要有新的线程进入 ready_list 中，就要考虑优先级抢占的问题，所以在最后调用 thread_yield 函数。作 if 判断的原因与上一次实验的一样：防止相同优先级抢占 cpu 的情况

(9) thread_set_priority 函数:

最后考虑到捐赠状态下修改优先级的情况，还要修改 thread_set_priority 函数。该函数实现很简单，分为捐赠和非捐赠两种情况即可：

```
if (thread_current ()->donated == false)
{
    thread_current ()->priority = new_priority;
    if (new_priority <
        list_entry(list_max(&ready_list, cmp_thread, NULL), struct thread, elem)->priority)
        | thread_yield();
}
```

- 如果没有发生捐赠，则按照之前的做法即可。

```
else
{
    if (new_priority < thread_current ()->priority)
        thread_current ()->old_priority = new_priority;
    else
    {
        thread_current ()->old_priority = new_priority;
        thread_current ()->priority = new_priority;
        if (new_priority <
            list_entry(list_max(&ready_list, cmp_thread, NULL), struct thread, elem)->priority)
            | thread_yield();
    }
}
```

- 如果发生了捐赠，则按照课件上所讲的捐赠状态下优先级捐赠三种情况进行修改即可。
- 同时当修改优先级 > 捐赠优先级时，不要忘记考虑优先级抢占。

3. 实验结果

```
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
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: *** [check] 错误 1
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$
```

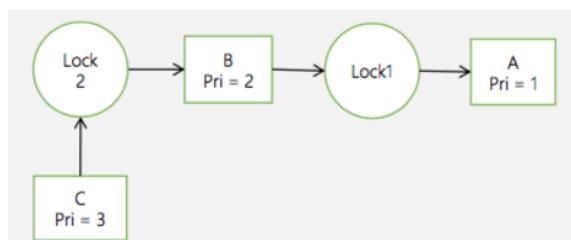
```
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
FAIL tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
```

如上图，多通过了八个 test。

4. 回答问题

问题 1：在就绪队列中的线程的优先级是否可能发生改变？如果能，请描述在哪种情况下可能发生。

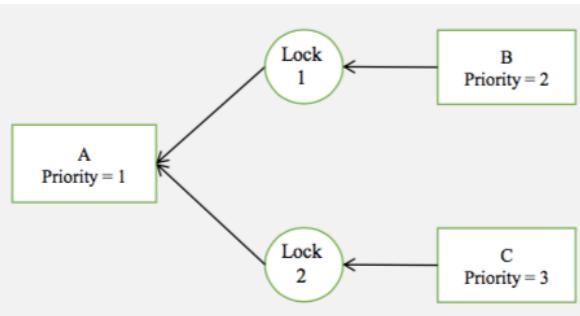
答：是可能发生的。下面借用课件上的图来进行说明：



- 如上图，这是课件上提到的线程加锁的第四种情况。根据上图可以知道，线程 A 先拥有 Lock1；线程 B 拥有 Lock2，然后去申请 Lock1；线程 C 申请 Lock2。
- 那么当线程 B 申请 Lock1 的时候，发生优先级捐赠，线程 A 的优先级升为 2；此时线程 A 是在 ready_list 中的，即就绪队列中的线程的优先级发生了改变。
- 当线程 C 申请 Lock2 时，同样发生优先级捐赠，线程 A、B 优先级均升为 3；此时线程 A 在 ready_list 中，而线程 B 在 Lock1 的 waiters 队列中。
- 所以就绪队列以及 waiters 队列中的线程的优先级均可能发生改变。

问题 2：如何解决一个线程占有了多个锁的优先级捐赠问题？

答：一个线程拥有多把锁的情况即如下图：



- 为线程增加一个锁队列，用以存放它拥有的所有锁，每当线程申请到一把锁，就将锁按照优先级大小插入到锁队列中。
- 我们定义：锁的优先级为它阻塞的所有线程中最大的优先级；线程的优先级为，自己的优先级与它拥有的锁中最大的优先级，这两者中较大的那个。

① 以线程 C 申请 Lock2 为例，我们做如下处理：

- 首先要将线程 C 的 blocked 变量指向 Lock2，因为它即将被 Lock2 阻塞
- 线程 C 的优先级要比 Lock2 的优先级高，所以 Lock2 的优先级要升为 3.
- 线程 A 的优先级比线程 C 的优先级要低，所以也要升为 3.
- 同时还要考虑嵌套捐赠的问题，因为线程 A 也可能正在申请其他的锁
- 然后进行 P 操作阻塞线程 B

② 若线程 A 要释放 Lock2：

- 首先要将 Lock2 从线程 A 的锁队列中 remove 掉
- 然后要检查、更新线程 A 的优先级，因为锁队列中最大的优先级已经不是 3 了
- 最后还要调用 V 操作，唤醒被 Lock2 阻塞的线程 C. (如果 waiters 队列中有优先级更高的，则唤醒之)

以上便是处理一个线程拥有多把锁的情况时，所遵循的规则。

问题 3: 在实现优先级捐赠之后，`thread_set_priority` 中需要考虑多少总情况？分别怎么处理？

答：一共有三种优先级，取全排列一共有 6 中情况，但是实际上只有 3 中情况，这是因为排除了捐赠的优先级小于原始的优先级的 3 种情况（因为根本不可能出现）。下面针对 3 种情况逐一分析：

① 修改的优先级 < 原始的优先级 < 捐赠的优先级

因为“修改的优先级”小于“捐赠的优先级”，线程当前的优先级即使修改了，还是会被捐赠为“捐赠的优先级”，所以仅需将“原始优先级”改为“修改的优先级”即可。

② 原始的优先级 < 修改的优先级 < 捐赠的优先级

修改同①。

③ 原始的优先级 < 捐赠的优先级 < 修改的优先级

“当前优先级”修改过后，不会被捐赠优先级，即等价于没有发生捐赠行为，所要同时将“原始的优先级”和“当前的优先级”修改为“修改的优先级”。

5. 实验感想

在第二课时之前，我曾尝试每次针对一个 test，一次修改一小部分代码，来完成这次的实验，但改到一半会出现各种蜜汁错误，最终以失败告终。

第二次课时之后，听从 TA 的建议，针对所有的情况一次进行修改，再加上有了之前的修改经验，少走了很多弯路；按照课件上的提示，几乎一遍就通过了 8 个测试。

这次的 8 个 test，除了最后一个 chain 稍微复杂一点，都比较好分析。我在分析 test 文件时，若一时没有头绪，会尝试着画出示意图来帮助我自己理解整个过程，如在上文分析 chain 时所体现的一样。

代码修改部分，只要在 coding 前厘清思路，几乎是不会遇到什么大的困难的。我认为唯一的难点是在嵌套捐赠部分。要在实现前思考清楚 while 里面的判断条件是什么，什么时候结束循环；循环体里面该执行那些内容。每次循环，我的做法是先更新锁的优先级，然后再调用自定义的 update_thread_priority 函数来更新锁的 holder 的优先级。之所以将这部分功能封装成函数，一是因为想让 while 循环体看起来更加简洁；而因为除了循环体中，在其他地方也要使用此功能。update_thread_priority 函数是在课上没有提到的，是需要我们自己思考出来的。

另外需要注意的一个细节是，在调用 list_inster_ordered 函数将锁按优先级顺序插入到锁队列中时，需要重写一个比较函数 cmp_lock，即使它的形式与线程的比较函数一模一样；否则会出现报错。

我本来是计划一共写 20 页左右的，但是在分析 test 的时候写着写着没有控制住，废话越写越多，结果多水了 10 页。在下一次实验报告中，我会尝试着在能够清楚表达意思的前提下，更加精简我的语言。