

目录

1. 实验目的	3
2. 实验过程	3
(一) Test 分析	3
1. Priority-donate-one	3
2. Priority-donate-multiple	5
3. Priority-donate-multiple2	8
4. Priority-donate-nest	11
5. Priority-donate-semaphore	15
6. Priority-donate-lower	17
7. Priority-semaphore	19
8. Priority-donate-chain	20
9. *Priority-donate-lower2(额外的测试，针对降低优先级之后出现捐赠的情况)	23
(二) 分析及实现	26
1. 测试要求	26
2. 代码修改	28
1) 获取锁	28
2) 释放锁	29
3) 修改优先级-方法 1	30
4) 修改优先级-方法 2 (基于方法 1 推论实现)	33
3. 实验结果	34
1) 完成后 make check 结果	34
2) Priority-donate-one 结果	35

3)	Priority-donate-multiple 结果.....	35
4)	Priority-donate-multiple2 结果.....	35
5)	Priority-donate-nest 结果.....	35
6)	Priority-donate-sema 结果.....	35
7)	Priority-donate-lower 结果.....	35
8)	Priority-sema 结果.....	36
9)	Priority-donate-chain 结果	36
4.	回答问题	37
5.	实验感想	38

1. 实验目的

- 1) 理解操作系统中的优先级反转情况对于操作系统的危害。
- 2) 为解决优先级反转的情况实现一个捐赠优先级的算法。
- 3) 学习如何针对样例分析编程与 debug。

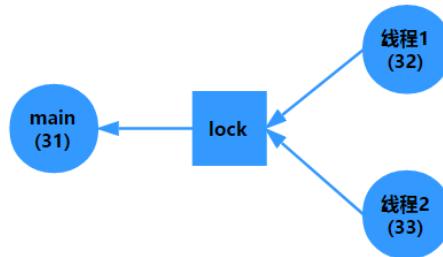
2. 实验过程

(一) Test 分析

1. Priority-donate-one

1) 测试内容与要求

创建一个默认优先级的测试主线程，然后主线程获得锁，之后创建两个高优先级的子线程，子线程调用创建时传入的函数，同样去获取锁，之后子线程会像主线程捐献优先级，当主线程释放锁时，其子线程应以优先顺序获取锁。



以下为其要求输出：

```
(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
```

2) 测试分析

我们先看一下测试主线程做了什么：

```
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.");
```

测试主线程开始优先级为（未定义值 31）。首先主线程创建一个锁并使用这个锁。然后创建子线程 1 其优先级为 32（发生抢占），传入创建函数 acquire1_thread_func，我们看看这个创建函数做了什么：

```

acquire1_thread_func (void *lock_)
{
    struct lock *lock = lock_;

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

```

我们发现其实这个函数做的事情其实很简单，就是去获取锁，得到锁之后输出，然后在释放锁之后再输出提示。那么我们现在将其和主线程关联起来，此时的主线程拥有锁，所以子线程在 `lock_acquire (lock);` 的时候会被阻塞，所以此时子线程不会获得锁，也不会输出信息，换句话说就是回到主线程继续执行，所以我们回到主线程，现在主线程创建完子线程 1 之后应该执行输出：

```

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

```

输出的内容是 32，和当前主线程的优先级，那么我们回看到.ck 文件

```
(priority-donate-one) This thread should have priority 32. Actual priority: 32.
```

我们发现这个时候的输出的当前主线程的优先级是 32，并不是原来的 31，所以此处就是测试子线程 1 的优先级捐赠给了测试主线程，才会输出 32。

之后主线程继续执行创建子线程 2，其优先级为 33（再次抢占主线程），并传入创建函数 `acquire2_thread_func` 其和 1 没什么两样就是其输出信息不同而已。相似的子线程 2 的执行情况和子线程 1 一致 `lock_acquire (lock);` 时会被阻塞。所以此时回到主线程继续执行输出：

```

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

```

输出的内容是 33，和当前主线程的优先级，那么我们再回看到.ck 文件：

```
(priority-donate-one) This thread should have priority 33. Actual priority: 33.
```

现在输出的主线程优先级是 33，那么就是子线程 2 的优先级捐赠给主线程，所以我们得出一个结论在 `lock` 的等待队列中的最高优先级的线程将会把其优先级捐献给主线程。

此时我们在看回.ck 文件：

```

(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.

```

我们发现，子线程的执行顺序应该是 2 先执行，意味着在释放锁的时候，`lock` 的等待队列中的线程时按照优先级从大到小进行 `unblock` 的。

同时我们看到未修改前的.output 文件：

```

(priority-donate-one) begin
(priority-donate-one) This thread should have priority 32. Actual priority: 31.
(priority-donate-one) This thread should have priority 33. Actual priority: 31.
(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

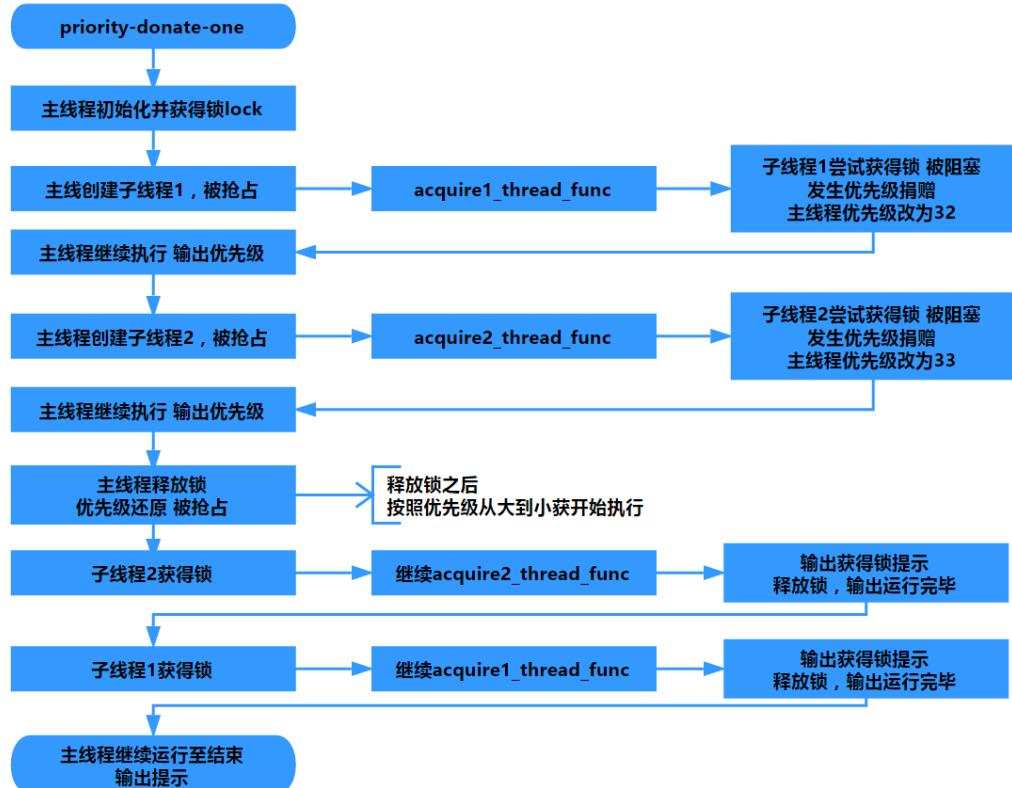
```

我们可以发现两个子线程的输出全部没有被读入，也就是说在线程释放锁之后直接输出结束，导致内存释放两个测试子线程的输出也不再被读入。这里就导致了如下输出结果，因为锁的内存被释放了所以子线程 1 获得之后也没

办法再次释放。此处 1 获得的原因是当前锁还是按照 FIFO 的顺序执行。

```
(priority-donate-one) end  
Execution of 'priority-donate-one' complete.  
(priority-donate-one) acquire1: got the lock  
Kernel PANIC at ../../threads/synch.c:232 in lock_release(): assertion `lock_held_by_current_thread (lock)' failed.  
Call stack: 0xc00280ae.
```

整个测试执行顺序如下流程：



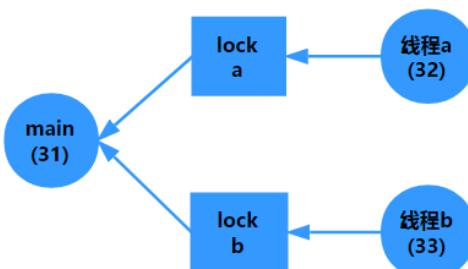
3) 测试结论

如果想要通过这个测试那么就需要我们在锁的部分增加一个捐献最大优先级的功能，同时锁的等待队列也许按照优先级大小进行排序，实现起来跟之前的 ready 队列原理相同（此处也可以是锁的 unblock 操作有序，此时其实就是在修改 PV 操作，因为 lock 是基于信号量实现的，将线程 block 和 unblock 也是由 PV 操作完成的，所以此时就是修改 PV 操作）。同时为主线程要最后退出应该在释放完锁之后恢复原来的优先级，同时还要确保其在优先级低于 ready 中的最高的线程时需要被抢占。

2. Priority-donate-multiple

1) 测试内容与要求

主线程获取锁 A 和 B，然后创建两个较高优先级的子线程。这两个子线程分别获取锁 A 和 B，从而出现优先级捐赠，使主线程优先级上升。主线程轮流释放锁，放弃其捐赠的优先级，使其优先级下降。



以下为其要求输出：

```
(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.
```

2) 测试分析

我们先看一下测试主线程做了什么：

```
lock_init (&a);  
lock_init (&b);  
  
lock_acquire (&a);  
lock_acquire (&b);  
  
thread_create ("a", PRI_DEFAULT + 1, a_thread_func, &b);  
msg ("Main thread should have priority %d. Actual priority: %d.",  
     PRI_DEFAULT + 1, thread_get_priority ());  
  
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 ());
```

测试主线程开始优先级为（未定义值 31）。首先主线程创建两个锁 AB 并获得这两个锁。然后创建子线程 a 其优先级为 32（发生抢占），传入创建函数 a_thread_func，我们看看这个创建函数做了什么：

```
a_thread_func (void *lock_)  
{  
    struct lock *lock = lock_;  
  
    lock_acquire (lock);  
    msg ("Thread a acquired lock a.");  
    lock_release (lock);  
    msg ("Thread a finished.");  
}
```

这个函数做的事情和前一测试相同，获取锁，得到锁之后输出，然后在释放锁之后在输出提示。那么此时的主线程拥有锁，所以子线程在 lock_acquire (lock);

（获取锁 A）的时候会被阻塞，所以此时还是回到主线继续执行输出执行输出：

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

```
(priority-donate-multiple) Main thread should have priority 32. Actual priority: 32.
```

同时看到.ck 文件，输出的内容是 32，和当前主线程的优先级为 32，所以此时也是发生了优先级捐赠。

之后主线程继续执行创建子线程 b，其优先级为 33（再次抢占主线程），并传入创建函数 b_thread_func 其和 a 的没什么两样就是其输出信息不同而已。相似的子线程 b 的执行情况和子线程 a 一致 lock_acquire (lock);（获取锁 B）时会被阻塞。所以此时主线继续执行输出：

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

```
(priority-donate-multiple) Main thread should have priority 33. Actual priority: 33.
```

再回看到.ck 文件，输出的内容是 33，和当前主线程的优先级为 33。因为现在输出的主线程优先级是 33，那么就是子线程 B 的优先级通过锁 B 捐献给主线程，因为此时的测试主线程的优先级只有 A 捐献的 32，不如 B 的高，所以此时主线的优先级被修改为 33。因此我们得出一个结论线程如果出现优先级捐赠，那么此时的优先级会是其所拥有的所有的锁的优先级的最大值。

此时我们在看回主线程的执行部分：

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

此时主线程先释放锁 B，我们看到.ck 文件的这个部分的输出。

```
(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.
```

此时是线程 b 开始执行，然后因为主线程释放了锁 B，所以其因为锁 B 出现的优先级捐赠行为应该被还原，所以此时的优先级回退变成锁 A 导致的优先级捐赠，就是变回了 32。因此此时的主线程被线程 b 抢占，当线程 b 执行完之后，主线程继续执行。

之后我们在继续看回主线程执行部分：

```
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());
```

此时主线程释放锁 A，我们看到.ck 文件的这个部分的输出。

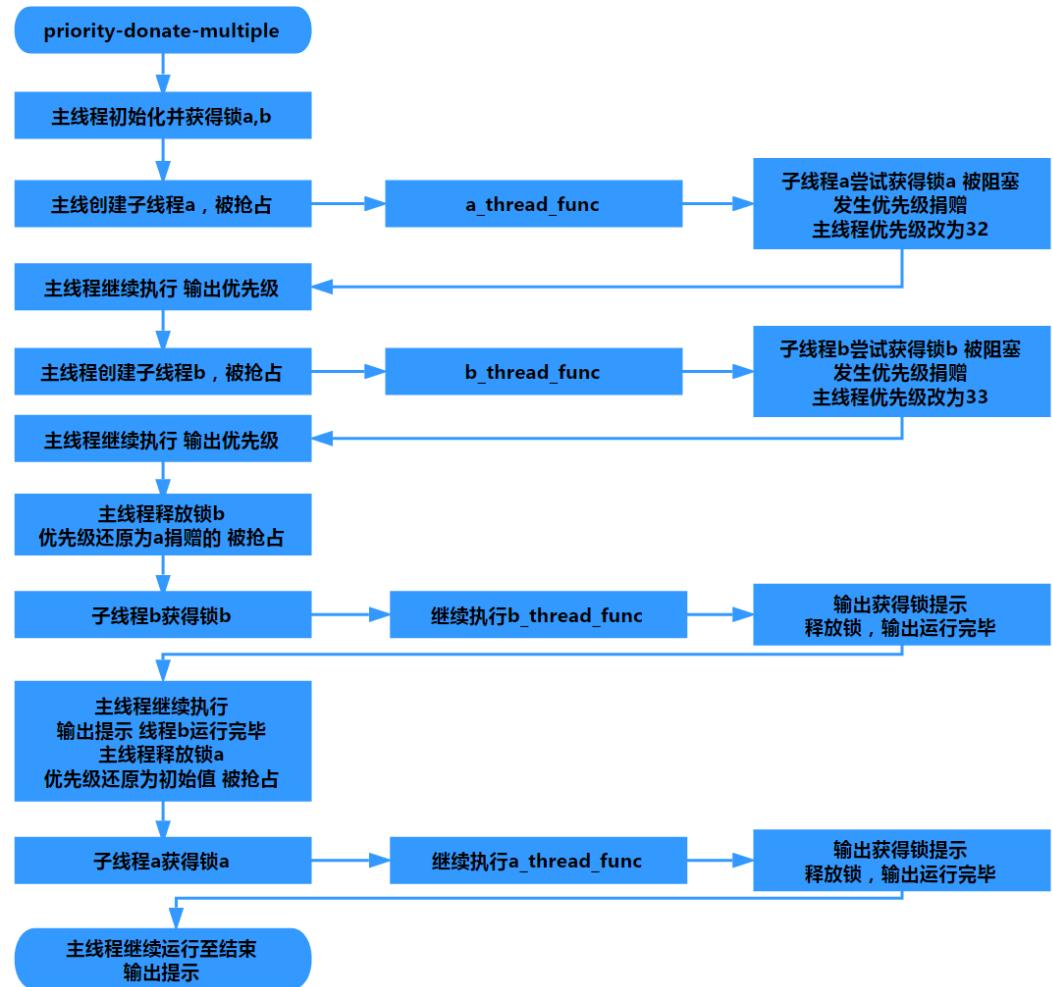
```
(priority-donate-multiple) Thread a acquired lock a.
(priority-donate-multiple) Thread a finished.
(priority-donate-multiple) Thread a should have just finished.
(priority-donate-multiple) Main thread should have priority 31. Actual priority: 31.
```

此时线程 a 开始执行，因为主线程释放了锁 A，所以主线程的优先级被还原为了 31，所以此时 a 抢占主线程开始执行，然后输出相提示，最后回到优先级最低的主线程继续执行至结束。

至此我们看到此时执行完毕，我们能发现在主线程释放锁之后出现的优先级需要被还原，如果没有主线程没有其他锁，就还原成原先优先级，不然就需要判断其还拥有的锁是否还会出现优先级捐赠，有的话就还原成捐赠优先级。

我们看到.output 文件可以发现两个子线程的输出全部没有被读入，也就是说在线程释放锁之后直接输出结束，导致内存释放两个测试子线程的输出也不再被读入。单跑未修改前的测试，我们能发现一个相同的问题，其的输出也出现了非法访问锁的问题。

整个测试执行顺序如下流程：



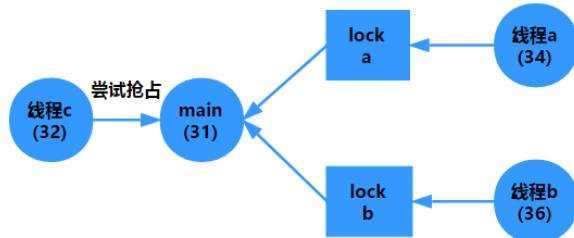
3) 测试结论

如果想要通过这个测试那么就需要我们在锁捐赠优先级的部分进行修改，使得捐赠优先级是当前线程所有的锁中最大优先级，同时锁释放之后还要判断当前锁的优先级是否需要还原或者是其所拥有的最大的优先级的锁的继续捐赠优先级，同样的主线程优先级更改之后还要判断其当前优先级是不是需要被抢占。

3. Priority-donate- multiple2

1) 测试内容与要求

主线程获取锁 A 和 B，然后创建三个较高优先级的子线程 a,b,c。这 a,b 两个子线程分别获取锁 A 和 B，从而出现优先级捐赠使主线程优先级上升。c 不获取锁，用来确保主线程的是真的出现优先级捐赠，不会因为原先优先级而被抢占，同时测试释放低于捐赠优先级的锁不会出现捐赠优先级的改变。



以下为其要求输出：

```
(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 order.  
(priority-donate-multiple2) Main thread should have priority 31. Actual priority: 31.
```

2) 测试分析

我们先看一下测试主线程做了什么：

```
lock_init (&a);  
lock_init (&b);  
  
lock_acquire (&a);  
lock_acquire (&b);  
  
thread_create ("a", PRI_DEFAULT + 3, a_thread_func, &a);  
msg ("Main thread should have priority %d. Actual priority: %d.",  
     PRI_DEFAULT + 3, thread_get_priority ());  
  
thread_create ("c", PRI_DEFAULT + 1, c_thread_func, NULL);  
  
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 ());
```

测试主线程开始优先级为（未定义值 31）。首先主线程创建两个锁 AB 并获得这两个锁。然后创建子线程 A 其优先级为 32（发生抢占），传入创建函数 a_thread_func，我们看看这个创建函数做了什么：

```
a_thread_func (void *lock)  
{  
    struct lock *lock = lock;  
  
    lock_acquire (lock);  
    msg ("Thread a acquired lock a.");  
    lock_release (lock);  
    msg ("Thread a finished.");  
}
```

这个函数做的事情和之前测试一样，去获取锁，得到锁之后输出，然后在释放锁之后在输出提示。此时的主线程拥有锁，所以子线程在 lock_acquire (lock); (获取锁 A) 的时候会被阻塞，所以此时回到主线程执行输出执行输出：

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

```
(priority-donate-multiple2) Main thread should have priority 34. Actual priority: 34.
```

那么我们看到 ck 文件，输出的内容是 34，和当前主线程的优先级 34，这个时候的输出的当前主线程的优先级是 34，所以此处就是测试子线程 a 的优先级通过锁 A 捐赠给了测试主线程，使得主线程优先级变成 34。

之后主线程执行的是创建线程 c，因为此时的主线程接受了线程 a 的优先级捐赠，所以线程 c 无法抢占只能在 ready 中继续等待主线程运行。

之后主线程继续执行创建子线程 b，其优先级为 36（再次抢占主线程），并传入创建函数 b_thread_func 其和 a 的没什么两样就是其输出信息不同而

已。相似的子线程 b 的执行情况和子线程 a 一致 `lock_acquire (lock);` (获取锁 B) 时会被阻塞。所以此时主线继续执行输出:

```
msg ("Main thread should have priority %d. Actual priority: %d.",  
    PRI_DEFAULT + 5, thread_get_priority ());  
(priority-donate-multiple2) Main thread should have priority 36. Actual priority: 36.
```

再看到.ck 文件，输出的内容是 36，和当前主线程的优先级 36. 现在输出的主线程优先级是 36，那么就是子线程 b 的优先级通过锁 B 捐赠给主线程，因为此时的测试主线程的优先级只有 a 捐献的 34，不如 b 的高，所以此时主线的优先级被修改为 36。此时的结论和之前一个测试相同。

此时我们在看回主线程的执行部分:

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

此时主线程先释放锁 A 然后在释放锁 B，我们看到.ck 文件的这个部分的输出。

```
(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 order.  
(priority-donate-multiple2) Main thread should have priority 31. Actual priority: 31.
```

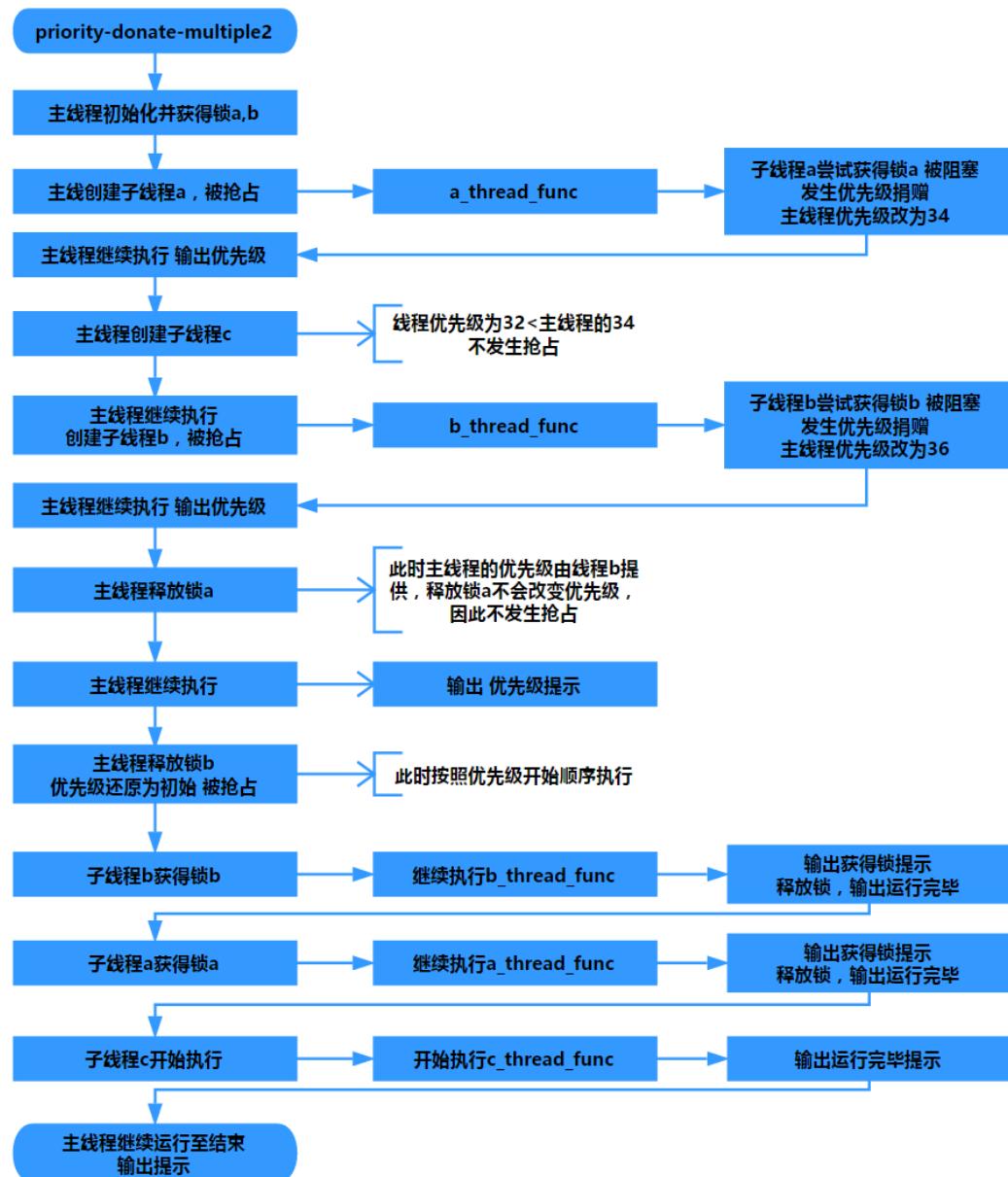
我们可以看到先释放锁 A 的时候线程 a 并没有继续执行，说明了释放锁 A 并不会影响当前线程的优先级，因为此时的捐赠优先级是由锁 B 捐赠的，所以不会受到锁 A 的影响，此时线程的优先级不变，所以此时线程 a 在 ready 中等待，之后再释放锁 B，这时主线程的优先级会被还原成原先优先级，因为此时主线程已经没有锁了，所以此时主线程的优先级还原，为所有线程中最低的，被抢占。之后按照当前 ready 中优先级大小运行，所以此时是 b 先运行完毕之后，a 继续运行，之后才是 c，最后一句输出就是验证主线程优先级被还原。所以此处就是线程释放最高优先级的锁的时候才需要修改优先级不然就保持原来的优先级即可。

我们看到未修改时.output 文件:

```
(priority-donate-multiple2) Main thread should have priority 34. Actual priority: 31.  
(priority-donate-multiple2) Thread c finished.  
(priority-donate-multiple2) Main thread should have priority 36. Actual priority: 31.  
(priority-donate-multiple2) Main thread should have priority 36. Actual priority: 31.  
(priority-donate-multiple2) Threads b, a, c should have just finished, in that order.  
(priority-donate-multiple2) Main thread should have priority 31. Actual priority: 31.
```

可以发现此时 c 线程会抢占主线程，因为主线程没有出现优先级捐献，a,b 子线程的输出全部没有被读入，也就是说在线程释放锁之后直接输出结束，导致内存释放两个测试子线程的输出也不再被读入。单跑未修改前的测试，我们能发现一个相同的问题，其的输出也出现了非法访问锁的问题。

整个测试执行顺序如下流程:



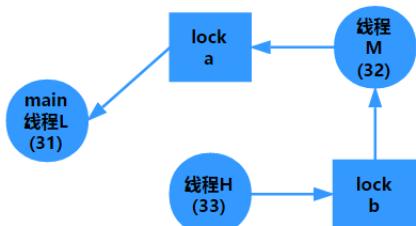
3) 测试结论

如果想要通过这个测试那么就需要在之前的基础上增加在线程的优先级捐赠部分需要确保不会因为原先优先级低而被抢占，应该是以捐赠的优先级来判断是否应该被抢占，之后还要确保只要释放线程锁获得的低于当前捐赠优先级的锁不会出现优先级更改。

4. Priority-donate-nest

1) 测试内容与要求

主线程获取锁 A。中优先级线程 M 获取锁 B 再获取锁 A 然后阻塞。之后高优先级线程 H 获得锁 B 时阻塞。用来测试递归捐赠优先级。



以下为其要求输出：

```
(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.
```

2) 测试分析

我们先看一下测试主线程做了什么：

```
lock_init (&a);  
lock_init (&b);  
  
lock_acquire (&a);  
  
locks.a = &a;  
locks.b = &b;  
thread_create ("medium", PRI_DEFAULT + 1, medium_thread_func, &locks);  
thread_yield ();  
msg ("Low thread should have priority %d. Actual priority: %d.",  
    PRI_DEFAULT + 1, thread_get_priority());  
  
thread_create ("high", PRI_DEFAULT + 2, high_thread_func, &b);  
thread_yield ();  
msg ("Low thread should have priority %d. Actual priority: %d.",  
    PRI_DEFAULT + 2, thread_get_priority());
```

测试主线程开始优先级为（未定义值 31）。首先主线程创建两个锁 AB，只获得锁 A。然后创建中优先级子线程（m 线程），其优先级为 32（发生抢占），传入创建函数 medium_thread_func，我们看看这个创建函数做了什么：

```
medium_thread_func (void *locks)  
{  
    struct locks *locks = locks_;  
  
    lock_acquire (locks->b);  
    lock_acquire (locks->a);  
  
    msg ("Medium thread should have priority %d. Actual priority: %d.",  
        PRI_DEFAULT + 2, thread_get_priority());  
    msg ("Medium thread got the lock.");  
  
    lock_release (locks->a);  
    thread_yield ();  
  
    lock_release (locks->b);  
    thread_yield ();  
  
    msg ("High thread should have just finished.");  
    msg ("Middle thread finished.");  
}
```

m 线程线程获取锁 B 之后获取锁 A，此时会被阻塞。回到主线程继续执行。此时主线程执行输出：

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

```
(priority-donate-nest) Low thread should have priority 32. Actual priority: 32.
```

此时主线程优先级为 32，意味着此时出现优先级捐赠。然后主线程创建高优

先级子线程（抢占主线程），之后执行函数 high_thread_func，我们转到这个函数：

```
high_thread_func (void *lock_)
{
    struct lock *lock = lock_;

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

这个时候高优先级线程去获取锁 B，被阻塞，回到主线程继续执行，
msg ("Low thread should have priority %d. Actual priority: %d.",
 PRI_DEFAULT + 2, thread_get_priority ());

(priority-donate-nest) Low thread should have priority 33. Actual priority: 33.

此时要求输出的内容是 33，所以意味着现在高优先级的进程其优先级通过锁 B 捐赠给了中优先级线程，然后中优先级线程再通过锁 A 将优先级线程捐赠给了测试主线程，所以此处测试的是优先级的递归捐赠。

之后主线程继续执行：

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

此时主线程释放锁 A，所以优先级应该被还原，然后被中优先级线程抢占，之后中优先级线程也可以开始执行，此时看到中优先级传入的创建函数其输出 medium_thread_func：

```
msg ("Medium thread should have priority %d. Actual priority: %d.",
    PRI_DEFAULT + 2, thread_get_priority ());
msg ("Medium thread got the lock.");
(priority-donate-nest) Medium thread should have priority 33. Actual priority: 33.
(priority-donate-nest) Medium thread got the lock.
```

我们可以看到其输出的优先级是 33，不是创建时的 32，所以证明中优先级线程是高优先级的线程通过锁 B 捐献了优先级给了中优先级线程。继续看到 medium_thread_func：

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

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

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

先释放锁 A 在启用线程调度，但是因为此时锁 A 的优先级是由中优先级线程提供的，且锁 A 也没有阻塞任何线程，所以释放 A 不会出现任何情况。

之后释放锁 B，所以此时中优先级线程的优先级会被还原，然后高优先级线程解除阻塞并抢占中优先级线程，所以此时我们能看到高优先级线程输出：

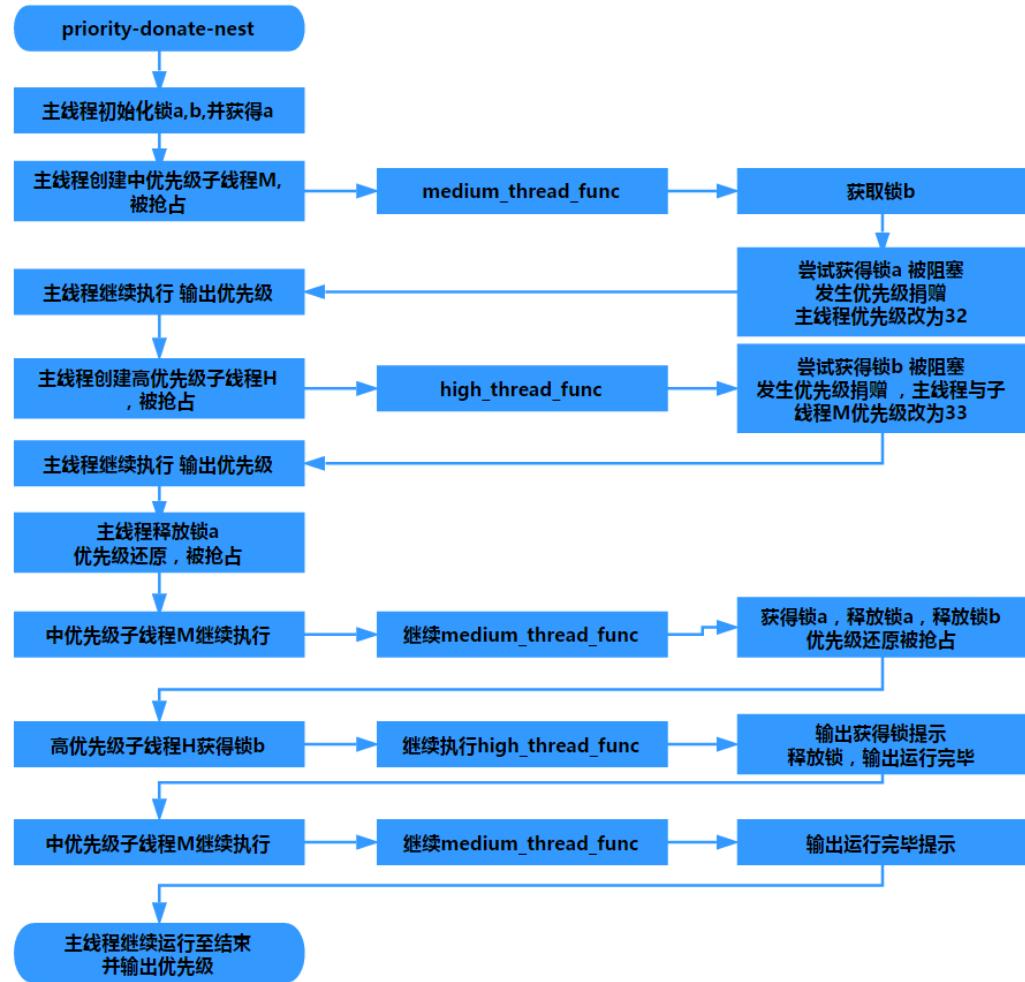
```
(priority-donate-nest) Medium thread got the lock.
(priority-donate-nest) High thread got the lock.
(priority-donate-nest) High thread finished.
(priority-donate-nest) High thread should have just finished.
```

高优先级线程运行完毕后，中优先级线程是当前 ready 队列中优先级最高的线程，所以其继续运行，最后才交由主线程继续运行。所以此时释放部分和之前一样只要判断优先级是否更改或者还原，然后在判断需不需要抢占即

可。

我们可以看到这个测试中使用了很多次 yield 函数进行调度，是为了确保捐赠优先级是可以不会因为原优先级而被调出出 running 队列，同时在未修改时 yield 充当了释放抢占的功能，使得没有出现之前那种主线程直接运行到结束而使创建的子线程出现访问不存在的锁的问题。

测试流程如下：



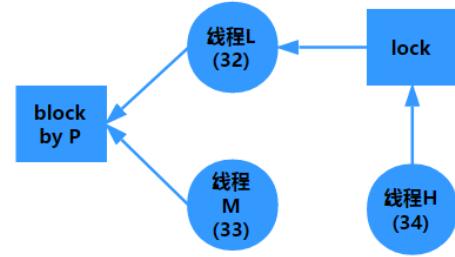
3) 测试结论

如果想要通过这个测试那么就需要在之前的基础上，增加循环捐献优先级，也就是整个当出現了一个优先级捐献的时候需要判断这个这个优先级的捐赠是否会影响到阻塞的当前线程的锁，如果会的会就应该将优先级捐赠，这个过程循环进行直至锁的拥有者不在被某一个锁阻塞，同时这个测试不用考虑释放锁之后的调度，我们之前的测试虽然已经考虑到了，但是这个测试已经为我们写好了调度。

5. Priority-donate-sema

1) 测试内容与要求

低优先级线程 L 获取锁，然后使用 P 操作阻塞自身。中优先级线程 M 使用 P 操作 block 自身。高优先级线程 H 尝试获取锁，被阻塞并将其优先级捐赠给 L。之后主线程进行 V 操作释放一个被 P 操作阻塞的线程，要求其释放 L 线程。



以下为其要求输出：

```
(priority-donate-semaphore) Thread L acquired lock.  
(priority-donate-semaphore) Thread L downed semaphore.  
(priority-donate-semaphore) Thread H acquired lock.  
(priority-donate-semaphore) Thread H finished.  
(priority-donate-semaphore) Thread M finished.  
(priority-donate-semaphore) Thread L finished.  
(priority-donate-semaphore) Main thread finished.
```

2) 测试分析

同样先看到主线程：

```
lock_init (&ls.lock);  
sema_init (&ls.sema, 0);  
thread_create ("low", PRI_DEFAULT + 1, l_thread_func, &ls);  
thread_create ("med", PRI_DEFAULT + 3, m_thread_func, &ls);  
thread_create ("high", PRI_DEFAULT + 5, h_thread_func, &ls);  
sema_up (&ls.sema);  
msg ("Main thread finished.");
```

主线程没有什么额外操作，就是注册了一个锁和一个 value=0 的信号量，创建低优先级线程 L，主线程被抢占，转到 L 内调用函数：

```
l_thread_func (void *ls_){  
    struct lock_and_sema *ls = ls_;  
  
    lock_acquire (&ls->lock);  
    msg ("Thread L acquired lock.");  
    sema_down (&ls->sema);  
    msg ("Thread L downed semaphore.");  
    lock_release (&ls->lock);  
    msg ("Thread L finished.");  
}
```

L 线程获取锁，输出提示 Thread L acquired lock，之后使用一次 P 操作阻塞自身同时输出提示，此时回到主线程继续执行，主线程创建中优先级进程 M，主线程被抢占，转到 M 调用函数：

```
m_thread_func (void *ls_){  
    struct lock_and_sema *ls = ls_;  
  
    sema_down (&ls->sema);  
    msg ("Thread M finished.");  
}
```

没什么特别只有使用一次 P 操作阻塞自身，所以此时再次回到主线程，主线程创建高优先级进程 H，主线程被抢占，转到 H 调用函数：

```

h_thread_func (void *ls_)
{
    struct lock_and_sema *ls = ls_;

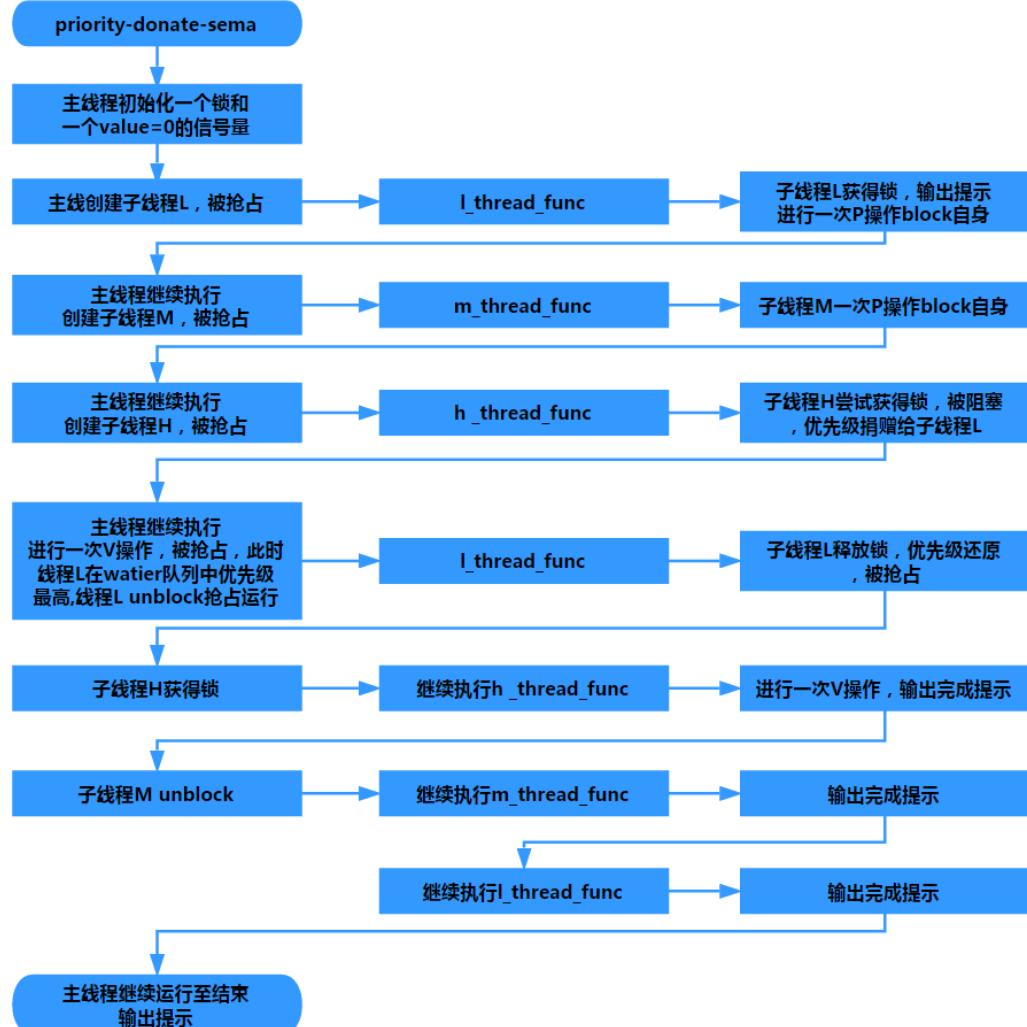
    lock_acquire (&ls->lock);
    msg ("Thread H acquired lock.");

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

```

这里 H 去获取锁，结果被阻塞，然后发生优先级捐赠给 L，所以此时 L 的优先级应该为 34，之后再次回到主线程。主线程执行一次 V 操作释放一个被 P 操作阻塞的进程，我们看到前面的输出要求，能发现之后是 L 线程输出，所以此时释放的线程 L，所以得出结论在 V 操作释放线程的时候应该释放优先级最大的那个线程。所以此时线程 L 继续执行，释放锁，优先级还原，被线程 H 抢占，之后 H 执行 V 操作线程 M 被 unblock，因为 H 优先级最高所以不会被抢占，线程 H 继续执行，之后线程 M 优先级高于线程 L，所以线程 M 先执行执行，再是线程 L 执行，最后测试主线程执行。

测试流程如下：



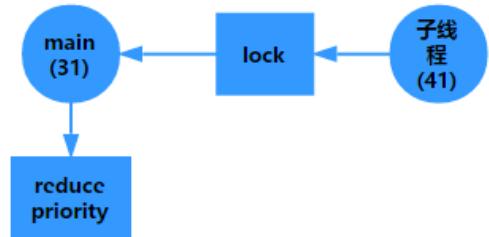
3) 测试结论

如果想要通过这个测试那么就需要在之前的优先级捐赠与还原的基础上，增加信号量的 waiter 队列排序（或者是 V 操作有序 unblock），然后还需要对于 V 操作之后判断是否抢占，实际上此处对于信号量的操作就是对 lock 的 block 队列有序出队的操作，此处操作我们之前已经要求实现。

6. Priority-donate-lower

1) 测试内容与要求

主线程获取锁。创建一个较高优先级的子线程，去获取锁，从而使其向主线程捐赠优先级。然后主线程降低优先级，但是直到主线程释放锁才这个优先级才生效。



以下为其要求输出：

```
(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.
```

2) 测试分析

同样先看到主线程：

```
lock_init (&lock);  
lock_acquire (&lock);  
thread_create ("acquire", PRI_DEFAULT + 10, acquire_thread_func, &lock);  
msg ("Main thread should have priority %d. Actual priority: %d.",  
    PRI_DEFAULT + 10, thread_get_priority ());
```

主线程先初始化并获得一个锁，之后创建一个优先级为 41 的子线程，主线程被抢占，我们转到子线程的 acquire_thread_func：

```
acquire_thread_func (void *lock_){  
    struct lock *lock = lock_;  
  
    lock_acquire (lock);  
    msg ("acquire: got the lock");  
    lock_release (lock);  
    msg ("acquire: done");  
}
```

很简单的获取锁的动作，然后子线程被 block，子线程优先级高于主线程所以出现了优先级捐赠，所以此时主线程的优先级变为 41，这时回到主线程输出：

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

```
(priority-donate-lower) Main thread should have priority 41. Actual priority: 41.
```

不出意外要求输出是捐赠的优先级 41；继续看到主线程执行：

```

msg ("Lowering base priority...");
thread_set_priority (PRI_DEFAULT - 10);
msg ("Main thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT + 10, thread_get_priority ());
lock_release (&lock);
msg ("acquire must already have finished.");
msg ("Main thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT - 10, thread_get_priority ());

```

主线程降低优先级，然后输出，我们看到要求输出的内容：

```

(priority-donate-lower) Lowering base priority...
(priority-donate-lower) Main thread should have priority 41. Actual priority: 41.

```

此时输出的优先级还是 41，这说明此时的捐赠的优先级是不能被修改的，所以此时主线程的优先级还是原先的捐赠的优先级。之后主线程释放掉锁，此时主线程应该还原优先级，我们在看到输出：

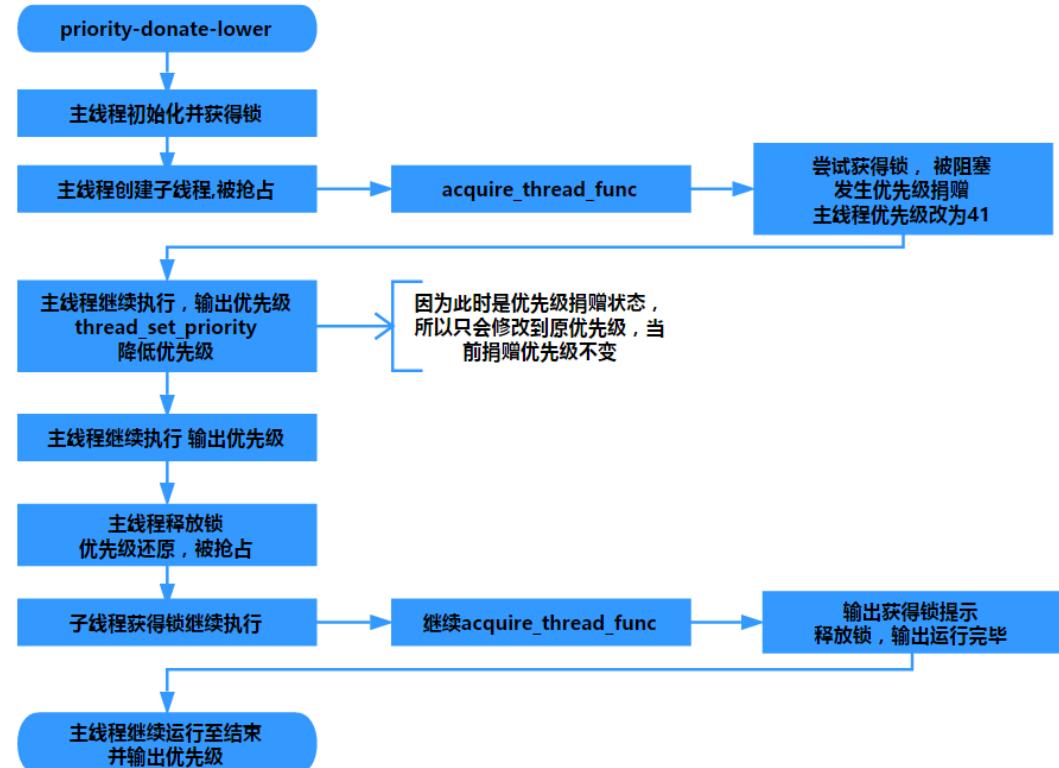
```

(priority-donate-lower) acquire must already have finished.
(priority-donate-lower) Main thread should have priority 21. Actual priority: 21.

```

此处输出的优先级就是主线的之前降低了的优先级，也就是说在捐赠优先级的情况下降低优先级是不会改变当前的优先级的，只会在释放掉锁的时候，也就是还原的优先级的时候才会变成这个降低的优先级。

测试流程如下图：



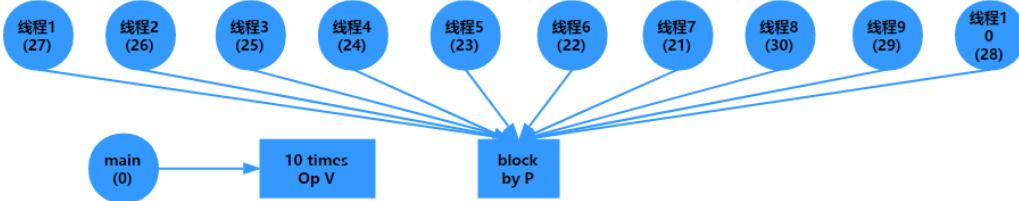
3) 测试结论

如果想要通过这个测试那么就需要在之前的优先级捐赠的基础上，对于调用修改线程优先级的时候增加判断，如果出现了优先级捐赠的情况调低的优先级就应该暂时不对现在优先级生效，直到拥有者释放掉锁才改变拥有者的优先级。此时我们能联想到如果需要调高优先级大于原捐赠优先级那么现优先级应该出现更改，这是线程就不需要在用那个捐赠的优先级了。

7. Priority-sema

1) 测试内容与要求

主线程初始化一个 value=0 的信号量，之后以 27,26,25,24,23,22,21,30,29,28 的优先级顺序创建线程，之后把这子线程全部通过 P 操作 block，之后使用 10 次 V 操作，要求每次 V 操作按照优先级有序唤醒。



2) 测试分析

同样先看到主线程：

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

主线程先初始化一个信号量 value=0，然后降低自己的优先级为最低，在以 27,26,25,24,23,22,21,30,29,28 的优先级顺序创建子线程，创建这些子线程的时候都发生主线程被抢占。我们看到 priority_sema_thread：

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

此处就是每个子线程创建的时候都会执行一次 P 操作 block 自身。所有每次都会回到主线程创建下一子线程。之后我们看到创建完所有子线程之后的主线程：

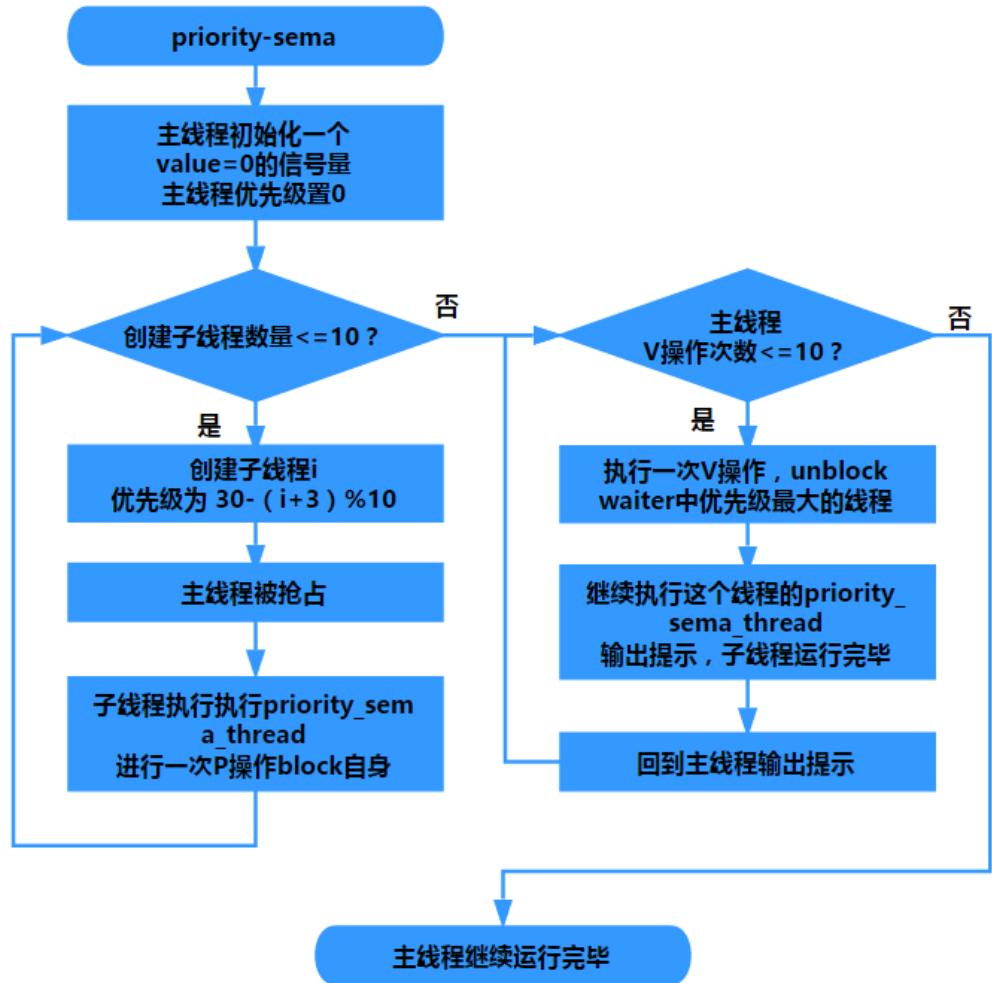
```
for (i = 0; i < 10; i++)
{
    sema_up (&sema);
    msg ("Back in main thread.");
}
```

此处执行了 V 操作释放一个子线程，我们看到输出模式：

```
(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.
```

我们能发现 V 操作的唤醒是按照优先级大小有序唤醒的，同时一点主线程进行一次 V 操作之后就立马被子线程所抢占，直到子线程输出完毕后，主线程继续执行。

测试流程如下：



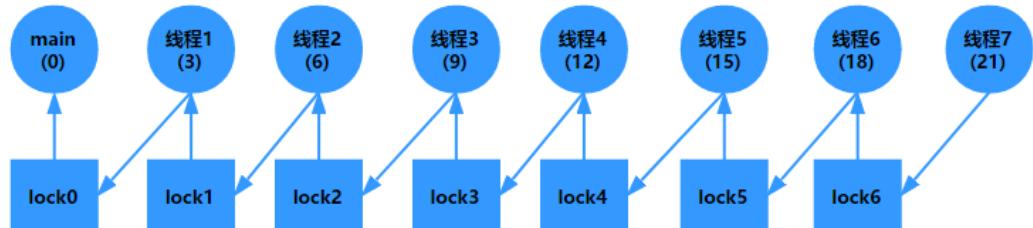
3) 测试结论

如果想要通过这个测试，那么就需要改造 V 操作的唤醒顺序或者说 P 操作进入 waiting 队列的顺序，之后使用 V 操作唤醒的时候还要根据优先级判断是否需要进行抢占。

8. Priority-donate-chain

1) 测试内容与要求

主线程初始化一个链锁，并获得其中的 0 号锁，之后创建子线程每个子线程的优先级都是前一个+3，并且获得的链锁中的后一个锁，之后再去获取前一个锁，导致自己被 block，创建完所有子线程后，主线程释放链锁中的 0 号锁。测试其会不会按照锁的顺序输出。



2) 测试分析

同样先看到主线程:

```
thread_set_priority (PRI_MIN);

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

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

for (i = 1; i < NESTING_DEPTH; i++)
{
    char name[16];
    int thread_priority;

    snprintf (name, sizeof name, "thread %d", i);
    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_create (name, thread_priority, donor_thread_func, lock_pairs + i);
    msg ("%s should have priority %d. Actual priority: %d.",
         thread_name (), thread_priority, thread_get_priority ());

    snprintf (name, sizeof name, "interloper %d", i);
    thread_create (name, thread_priority - 1, interloper_thread_func, NULL);
}
```

主线程开始先降低自己的优先级为 0，然后初始化一个锁数组，此处当成一个锁链来用，并获得 0 号锁，之后创建子线程，子线程的优先级是 3,6,9……，给第 i 个子线程传入两个锁一个是 i-1 号锁，一个是 i 号锁（最后一个子线程没有 i 号锁），因为此时优先级比主线程高所以会发生抢占，之后我们转到创建函数：

```
donor_thread_func (void *locks_)
{
    struct lock_pair *locks = locks_;

    if (locks->first)
        lock_acquire (locks->first);

    lock_acquire (locks->second);
    msg ("%s got lock", thread_name ());

    lock_release (locks->second);
    msg ("%s should have priority %d. Actual priority: %d",
         thread_name (), (NESTING_DEPTH - 1) * 3,
         thread_get_priority ());

    if (locks->first)
        lock_release (locks->first);

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

首先一个判断其有没有 i 号锁，有的话就获取（此处只有 7 号线程没有 7 号锁），之后去获取 i-1 号锁，当然此时会被阻塞，出现优先级捐赠给前一线程，我们看到创建循环里的输出：

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

```
(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.
```

这句话的每次输出都是主线的优先级，因为此时的运行线程时主线程且高优先级的子线程在不断递归捐赠的原因，所以现在主线程的优先级逐渐增大。我们还能看到创建循环里最后又创建了一个比当前优先级少 1 的线程，因为其优先级不够抢占要求所以其在 ready 队列中等待。我们看看其输出函数：

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

这个简单，其实就是用来跟在和他一起创建的进程后面告诉我们这个线程运行完了而已。

当主线程创建完所有的子线程之后 `lock_release (&locks[0]);` 主线程释放掉 0 号锁，优先级回到 0，被抢占，那么此时按照锁的获取顺序此时子线程 1 可以开始运行，输出 got lock，释放掉刚刚 block 他的 0 号锁，之后输出：

```
msg ("%s should have priority %d. Actual priority: %d",  
     thread_name(), (NESTING_DEPTH - 1) * 3,  
     thread_get_priority());  
(priority-donate-chain) thread 1 got lock  
(priority-donate-chain) thread 1 should have priority 21. Actual priority: 21
```

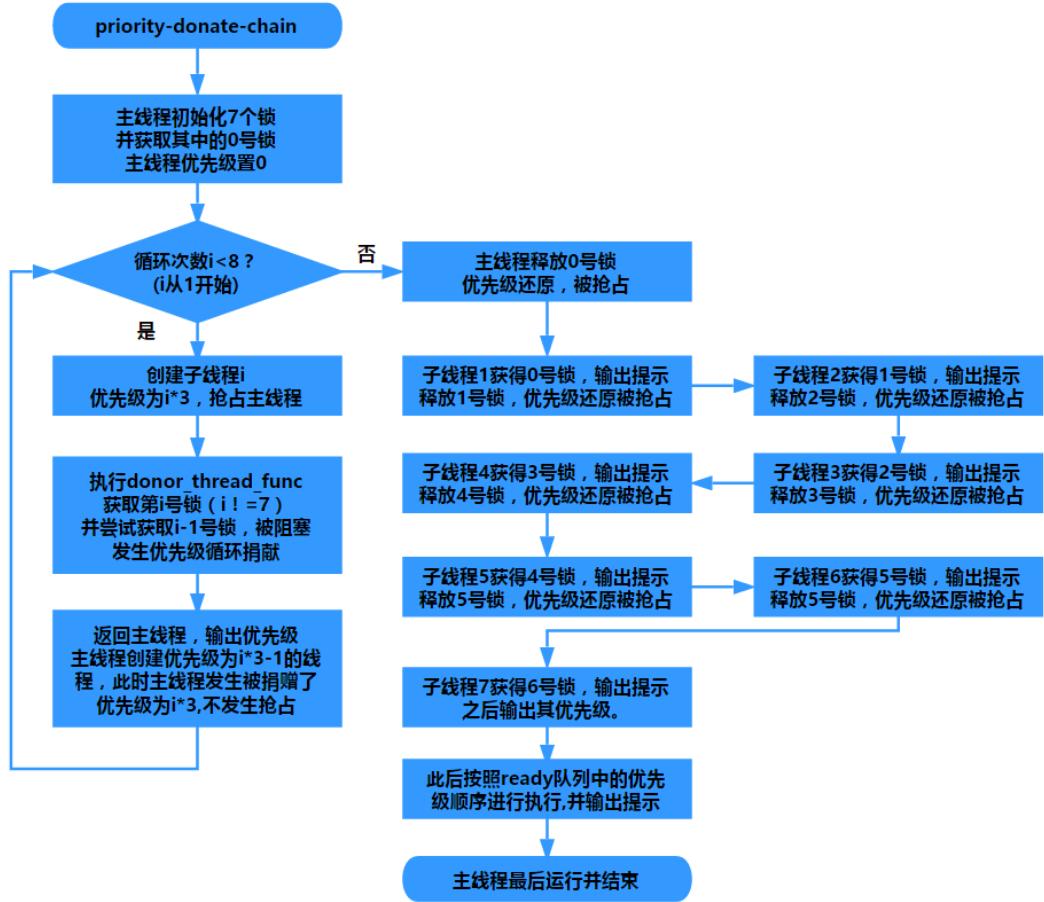
我们可以看到此时其输出的线程 1 的优先级还是 21，因为这个链锁的优先级全都是由最后的子线程 7 捐赠的，之后锁 1 释放掉他原来就有的那个锁，也就是 1 号锁。那么此时其会因为释放掉这个锁从而优先级恢复会原优先级而被下一个进程抢占，所以此时的输出顺序为：1->2->3->4->5->6->7。

子线程 2-7 都是在重复之前子线程 1 的过程，只是 7 号子线程因为没有拥有一个 7 号锁且他是之前整个链锁的优先级捐赠者，所以到 7 号线程就是终止了没有被抢占，之后就是每个线程按照 ready 中的优先级开始继续执行性输出如下这句：

```
msg ("%s finishing with priority %d.", thread_name(),  
     thread_get_priority());  
(priority-donate-chain) thread 7 finishing with priority 21.  
(priority-donate-chain) interloper 7 finished.
```

同时后面都跟着一句，这句跟着的话我们之前说过，就是在主线程创建子线程的 for 循环中最后每个子线程后面又接着创建的一个优先级少 1 的子线程，他只能在 ready 队列中带着。因为此时的这种输出顺序是按照创建时优先级大小顺序执行并输出的，所以他跟在比他高 1 优先级的那个子线程后面输出。此时的输出顺序应为 7->6->5->4->3->2->1，最后主线程才会输出因为其优先级为 0。

测试流程如下：



3) 测试结论

这个测试其实直接只要有循环捐赠优先级和释放完所有锁之后还原优先级并判断抢占就能通过（其实就是一个 nest 的延长版），只要之前的循环捐赠优先级写好了这个就能通过。

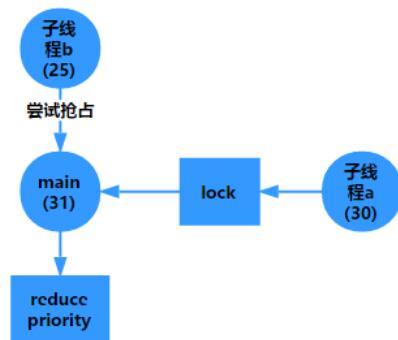
9. *Priority-donate-lower2(额外的测试，针对降低优先级之后出现捐赠的情况)

1) 测试介绍

我们在经过分析和实现之后发现一种该出现 `set_priority` 之后变成捐赠的情况，但是在这些测试中，全部没有涉及这种情况，为了检验自己的思路正确性编写了这个测试。

本测试由自己编写.c 文件和.ck 文件，并加入 Makefile 中(修改与添加部分见附件，以下只会展示代码部分实现)，修改的文件为 src/tests/threads 中的 Rubirc.priority / Make.test / tests.c / tests.h(已经包含在附件中直接覆盖即可食用)。

测试针对在使用 `set_priority` 中的现优先级=原优先级>新优先级情况，也就是当前主线程拥有锁，再其锁队列中 block 一个比他优先级更小的子线程 a，当主线程降低优先级置小于 a 的优先级的时候主线程应该被捐赠。



2) 测试流程介绍

以下是我编写的主线程的执行部分

```
lock_init (&lock);
lock_acquire (&lock);
msg ("Main thread create a thread have priority 30.");
thread_create ("a", PRI_DEFAULT -1, a_thread_func, &lock);
msg ("Main thread have priority 31 and sleep 10 ticks.");
timer_sleep(10);
msg ("Main thread wake and set the priority = 20.");
thread_set_priority(20);
msg ("Main thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT -1, thread_get_priority ());
thread_create ("b", PRI_DEFAULT -6, b_thread_func, NULL);
msg ("Main thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT -1, thread_get_priority ());
lock_release (&lock);
msg ("Main thread should have priority %d. Actual priority: %d.",
     PRI_DEFAULT -11, thread_get_priority ());
msg("a , b must already have finished.");
```

首先主线程注册一个锁并获取，之后创建一个优先级为 30 的子线程 a，此时主线程的优先级为 31>a 的 30，所以此时不会出现抢占，之后主线程休眠 10ticks。此时主线程被 block，子线程 a 开始执行，子线程 a 获取锁被 block。10ticks 过后主线程唤醒，使用 thread_set_priority(20); 将自身优先级改为 20，此时应该出现优先级捐赠，所以此时应该输出的主线程优先级是 30，之后主线程创建一个优先级为 25 的线程 b，那么此时 b 因为主线程的捐赠优先级为 30，所以不会发生抢占，然后主线程释放锁，优先级还原，被线程 a 抢占。子线程 a 运行完毕之后，子线程 b 开始运行，最后主线程运行并结束测试。两个线程函数运行内容如下（主要是输出提示的功能）：

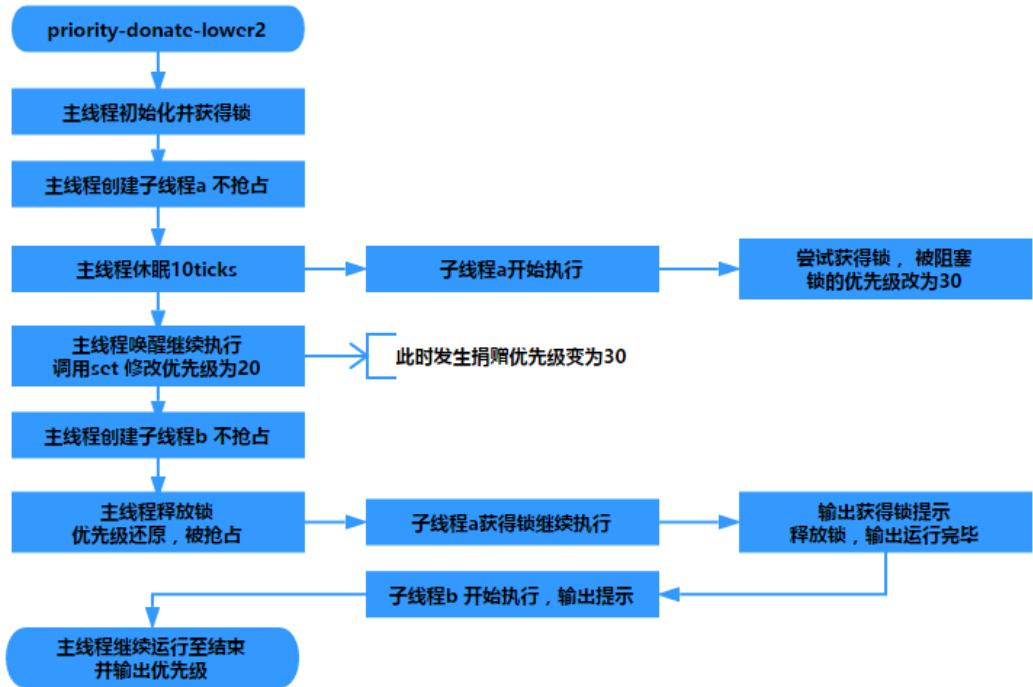
```
static void
a_thread_func (void *lock_)
{
    struct lock *lock = lock_;
    msg("a: begin try acquire lck and it have priority %d.", thread_get_priority ());
    lock_acquire (lock);
    msg ("a: got the lock.");
    lock_release (lock);
    msg ("a: done.");
}

static void
b_thread_func (void *arg_ UNUSED)
{
    msg("b: done.");
}
```

.ck 文件编写如下：

```
(priority-donate-lower2) begin
(priority-donate-lower2) Main thread create a thread have priority 30.
(priority-donate-lower2) Main thread have priority 31 and sleep 10 ticks.
(priority-donate-lower2) a: begin try acquire lck and it have priority 30.
(priority-donate-lower2) Main thread wake and set the priority = 20.
(priority-donate-lower2) Main thread should have priority 30. Actual priority: 30.
(priority-donate-lower2) Main thread should have priority 30. Actual priority: 30.
(priority-donate-lower2) a: got the lock.
(priority-donate-lower2) a: done.
(priority-donate-lower2) b: done.
(priority-donate-lower2) Main thread should have priority 20. Actual priority: 20.
(priority-donate-lower2) a , b must already have finished.
(priority-donate-lower2) end
```

测试流程如下：



3) 运行效果

- 未修改前：此时因为没有出现优先级捐赠，主线程休眠之后子线程 a 会出现主开始运行，然后因为子线程 a 获取锁被 block 而返回主线程继续执行并降低优先级，此时主线程在创建子线程 b,又被抢占，子线程 b 运行完毕，之后回到主线程，继续运行，然后主线程释放掉锁，此时没有修改所以不出现抢占主线程运行到结束并释放内存。结果如下：

```
(priority-donate-lower2) begin
(priority-donate-lower2) Main thread create a thread have priority 30.
(priority-donate-lower2) Main thread have priority 31 and sleep 10 ticks.
(priority-donate-lower2) a: begin try acquire lck and it have priority 30.
(priority-donate-lower2) Main thread awake and set the priority = 20.
(priority-donate-lower2) Main thread should have priority 30. Actual priority: 20.
(priority-donate-lower2) b: done.
(priority-donate-lower2) Main thread should have priority 30. Actual priority: 20.
(priority-donate-lower2) Main thread should have priority 20. Actual priority: 20.
(priority-donate-lower2) a , b must already have finished.
(priority-donate-lower2) end
```

- 直接展示我们在运行完成之后的 make check 结果（如下）：此时我们添加的新的测试样例出现在 make check 中，测试总数也变为 28

```

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-donate-lower2
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-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 28 tests failed.
```

(二) 分析及实现

1. 测试要求

通过之前的 test 分析我们知道需要通过测试需要修改如下几点：

- 1) 在一个线程获取一个锁的时候，如果这个锁的拥有者优先级比自己低，就提高其捐赠自己的优先级，并且如果这个锁的拥有者还被别的锁阻塞，将会递归地捐赠优先级。
- 2) 如果一个线程拥有多个锁，那么如果出现优先级捐赠的情况，应该是保留最大的捐赠优先级。
- 3) 当一个线程释放掉一个锁的时候，如果其没有拥有其他锁，或者其他锁能提供的优先级比原优先级低，那么将其还原为原优先级，否者其优先级改为其拥有的锁中的最高的优先级。
- 4) 当释放掉一个锁，其所提供的捐赠优先级不是最高捐赠有优先级，那么此时拥有者的优先级不会改变。
- 5) 对一个线程进行设置优先级的时候，如果这个线程处于被捐赠状态（原优先级！= 现优先级），此时判断其这个新的优先级会不会比捐赠优先级的高，如果是的话那么就应该同时修改现优先级和原优先级，否则修改原优先级；如果处于非捐赠状态，那么同时修改现优先级和原优先级，之后判断一下当前调整会不会导致优先级捐赠。
- 6) 将信号量的等待队列改为优先级有序。
- 7) 以上所有出现优先级换变化，或者出现线程被 unblock 的情况，都应该判断是否需要抢占。

综上结论我们需要可以总结出需要修改的两个流程：

- 获取锁：

我们在获取锁的时候如果这个锁没有拥有者，那么就只获取就可以了，有的话就需要判断会不会出现捐赠状态，这里就需要考虑嵌套的情况也就是第一点的后半部分；于此同时还需要设置锁的优先级为当前阻塞队列中的最大线程优先级，之后就是该线程被 P 操作 block，该线程被 unblock 的时候锁，即是该线程获得锁的时候。

- 释放锁：

释放锁的时候我们会同时 unblock 一个被 block 的线程，根据我们的 test 分析，我们需要 unblock 一个优先级最大的线程，因为 unblock 是通过 V 操作实现的，所以这里我们在 V 操作 unblock 的时候要先对被 block 的队列（也就是信号量的 waiter 队列）sort，之后再来 unblock，同时在锁这边我们需要更新锁的优先级（当前被 block 的最大优先级），然后还需要将当前锁从拥有者的锁队列中移除。

首先我们先增加一些参数，以应对我们之后我们需要操作时的需要，对应这两个操作我们，我们需要修改线程结构体和锁的结构体，增加几个参数具体如下：

```
int old_priority;           /* Old_Priority. ****NEW**** */
int64_t ticks_blocked;    /* Blocked flag. */
struct list_elem allelem;  /* List element for all threads list. */

/* Shared between thread.c and synch.c. */
struct list_elem elem;     /* List element. */
struct list locks;         /* List lock. ****NEW**** */
struct lock *lock_blocked; /* thread is blocked by these lock. */
```

此处我们增加了一个用于记录原优先级的 old_priority，一个 lock 结构体指针，用于记录此线程被哪个锁阻塞，一个 list 用于记录当前线程拥有的锁。对应初始化 init_thread()：

```
t->priority = priority;
t->old_priority=priority;
t->magic = THREAD_MAGIC;
t->lock_blocked=NULL;
list_init(&t->locks);
```

此处我们将现优先级和原优先级同时进行初始为同一个值，因为我们是用原优先级=现优先级来判断是否出现捐赠的。同时此处对于 locks 这个队列的初始化很重要，如果此处初始化没有添加就会造成运行时所有测试全部不通过。

之后我们对锁结构体进行改造：

```
struct lock
{
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
    struct list_elem elem;      /* List element. */
    int priority;               /* Priority. */
};
```

此处增加一个 list_elem，用于 list 中的操作（此处主要作用是对应线程结构体中的 list locks 队列，所有对于 locks 的修改基本基于这个），之后增加一个锁的优先级，定义为锁当前阻塞的线程中的最大优先级（初始化为 0）。

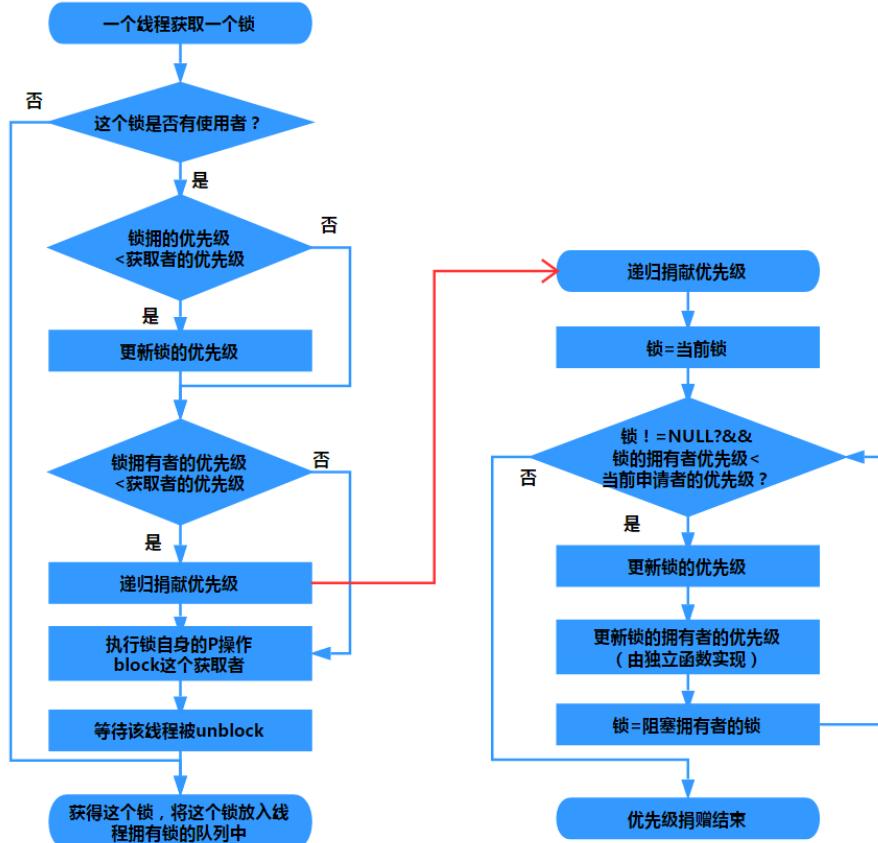
对应初始函数 lock_init()：

```
lock_init (struct lock *lock)
{
    ASSERT (lock != NULL);
    lock->priority=PRI_MIN;
    lock->holder = NULL;
    sema_init (&lock->semaphore, 1);
}
```

2. 代码修改

1) 获取锁

我们从之前的分析中得出获取锁的操作流程如下：



现在我们就需要改造获取锁的函数 lock_acquire()，这个函数现在要实现的是优先级捐赠问题，此处需要处理的是递归捐赠，同时捐赠只有是大于当前锁的拥有者的优先级才能捐赠，同时处理锁的优先级为被阻塞的线程中的最大优先级。这部分对应前面的 1 和 2，捐赠部分修改如下：

```

struct thread *cur=thread_current();
struct lock *lock_=lock;
enum intr_level old_level = intr_disable ();
if(lock_->holder!= NULL){
    cur->lock_blocked = lock;
    if(lock_->priority < cur->priority) lock_->priority = cur->priority;
    while (lock_ != NULL && lock_->holder->priority < cur->priority)
    {
        lock_->priority = cur->priority;
        //thread_donate_priority (lock_->holder,lock_->priority); //method one
        thread_change_priority (lock_->holder); //method two
        lock_ = lock_->holder->lock_blocked;
    }
}
intr_set_level (old_level);
sem_down (&lock->semaphore);

```

此处修改的 if(lock_->holder!= NULL) 意味着当前锁为有人使用，如果无人使用，那么我们直接跳过这个捐赠部分，直接去获取锁即可。

如果有人使用锁那么当前获取这个锁的线程就应该被这个锁阻塞那么此时我们之前在线程结构体中添加的阻塞标记就赋值为这个锁 (cur->lock_blocked = lock) 我们先用一个 if 来判断是否要更新锁的优先级，

因为我们定义的锁优先级是阻塞进程中最大优先级，所以此处用一个 if 进行判断。之后我们使用一个 while 循环实现递归捐赠优先级：

```
while(锁不为空 && 锁的拥有者的优先级 < 当前线程的优先级){  
    锁的优先级改为当前线程的优先级;  
    锁的拥有者的优先级改为当前线程的优先级;  
    锁=阻塞锁拥有者的锁;  
}
```

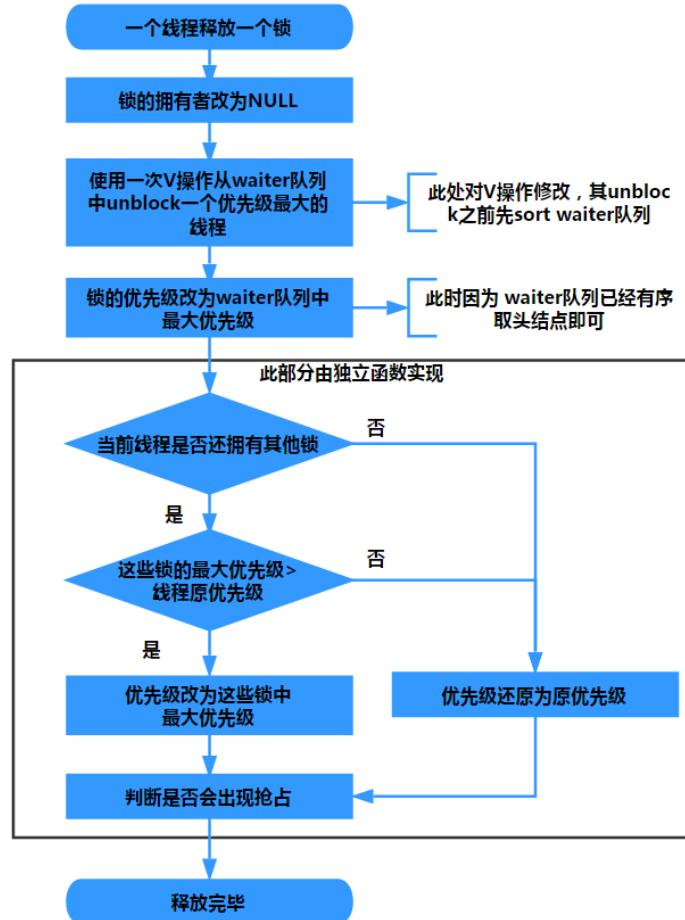
做完优先级捐赠之后我们，只需要让其被 P 操作 block 就可以了。之后我们来看获得锁的部分：

```
lock->holder = cur;  
cur->lock_blocked = NULL;  
list_push_back(&cur->locks, &lock->elem);
```

修改锁的拥有者为当前线程，当前线程不在被这个锁阻塞，之后将这个锁放入当前线程拥有锁的 list 中。

2) 释放锁

修改完获得锁的函数之后我们需要修改释放锁的函数，对应之前 3 和 4 点，实现从拥有者的锁队列中移除当前锁，以及更新锁当前的优先级，释放锁的操作流程如下：



对应修改函数 lock_release()：

```

lock_release (struct lock *lock)
{
    ASSERT (lock != NULL);
    ASSERT (lock_held_by_current_thread (lock));
    lock->holder = NULL;
    sema_up (&lock->semaphore);

    enum intr_level old_level = intr_disable ();
    lock->priority=list_empty(&lock->semaphore.waiters)?PRI_MIN:list_entry(
        list_begin(&lock->semaphore.waiters),struct thread,elem)->priority;
    list_remove (&lock->elem);
    intr_set_level (old_level);
    //thread_back_priority(thread_current());                                //method one
    thread_change_priority (thread_current());                               //method two
}

```

我们在释放锁的时候将锁的拥有者改为 NULL，之后使用 V 操作释放一个被 block 的线程，此时不会出现优先级捐赠，原因是 V 操作释放的是 waiter 队列中最大优先级的线程，也就是当前之后锁的拥有者的优先级是整个锁的 block 队列中最高的；之后我们需要更新锁的优先级，此时只需要将 waiter 队列的头拿出来即可，原因是每次 V 操作前都 sort waiter 队列，然后又拿出了第一个，所以现在 waiter 的头是优先级最高的，所以现在锁的优先级改为其优先级，当然锁为空时还原优先级即可，之后我们需要从拥有者的拥有的锁 list 中将这个锁移除，之后我们就需要将拥有者的优先级进行还原。

至此锁部分修改完毕，之后我们修改 V 操作，保证其 unblock 的线程的优先级是 waiter 队列中最大的，所以我们修改 seam_up()：

```

if (!list_empty (&sema->waiters))
{
    list_sort (&sema->waiters, &thread_cmp_priority, NULL);
    thread_unblock (list_entry (list_pop_front (&sema->waiters), struct thread, elem));
}

```

在 unblock 前增加对 waiter 队列 sort，确保队列的头是优先级最大的线程，所以此时之列使用 list_pop_front 移除并获得这个线程，然后我们在最后增加一个调度判断确定是否需要抢占。

至此所有锁和信号量修改完毕，那么我们现在实现优先级捐赠和还原。

3) 修改优先级-方法 1

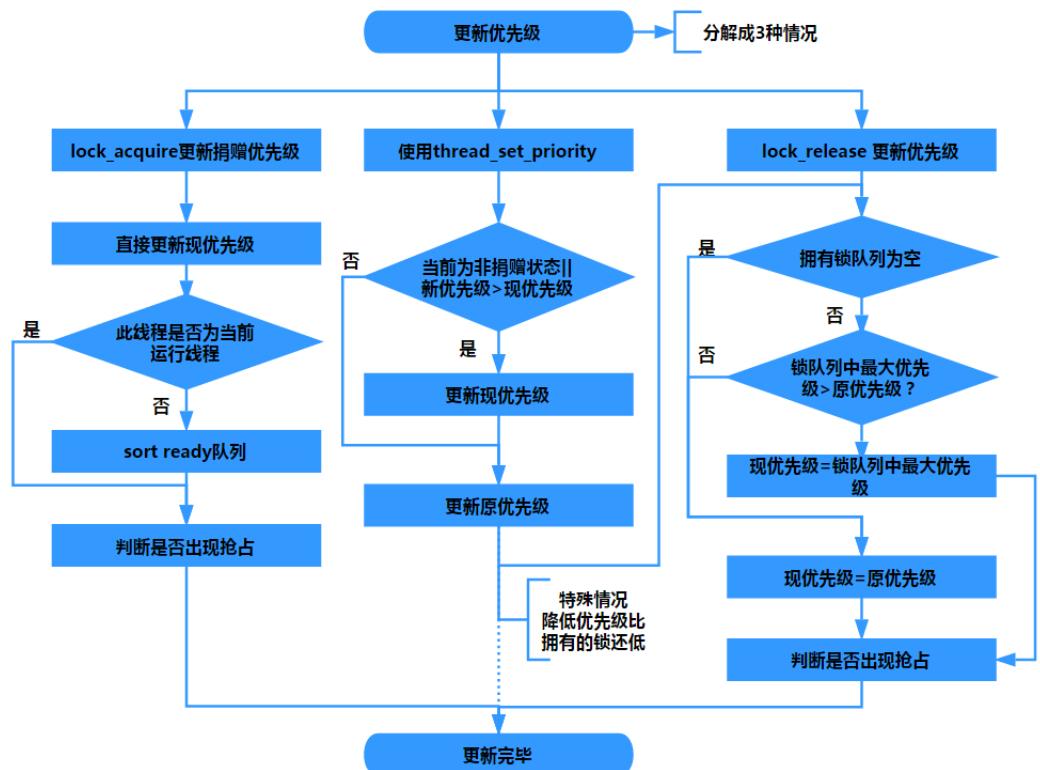
(方法一分析了所有修改优先级的情况，并对所有情况拆分实现，所以也较为冗长，提交代码中使用方法二，方法一被注释)

首先我们分析一下修改优先级的几种情况：

- 直调用 thread_set_priority()修改优先级
 - 新优先级>=现优先级>=原优先级：此时的情况不论是否捐赠，因为新的优先级大于先优先级，所以就算当前是捐赠情况在这个新的优先级到来的时候就会覆盖原优先级和现优先级，然后使线程退出捐赠状态。(test 中不存在这种情况)
 - 现优先级=原优先级>新优先级：此时为非捐赠状态，需要修改现优先级和原优先级，然后要判断是否会出现捐赠（此种情况不存在 test 中）。

- 现优先级>新优先级：直接修改原优先级。
- 在 lock_acquire() 中的递归捐献优先级：这种情况比较简单，因为我们在 lock_acquire() 中已经判断是否会出现捐赠，会捐赠才会调用修改，所以这里只需要将现优先级修改为传入的捐赠优先级，同时加之判断他是不是在 ready 队列，是的话就需要重新 sort ready 队列（实际上这个也不需要判断），之后都需要判断抢占。
- 在 lock_release() 释放锁的时候优先级回滚：此时我们需要判断其锁队列是不是空，空的话就直接恢复原优先级，否则的话比一下锁队列中的最大优先级是不是大于原优先级，是的话就捐赠，不是的话恢复原优先级。

情况分解后流程如下：



为了简化逻辑我们将 thread_set_priority() 我们新增一个函数
thread_donate_priority() 用于获得锁的时候，出现捐赠时修改优先级：

```

thread_donate_priority (struct thread *t,int priority)
{
    enum intr_level old_level = intr_disable ();
    t->priority=priority;
    if(t!=thread_current())
        list_sort (&ready_list,&thread_cmp_priority,NULL);
    intr_set_level (old_level);
    thread_preempt_priority();
}
  
```

我们这里直接修改优先级，因为我们在 lock_acquire() 中调用他的时候已经判断了大小所以我们直接赋值即可，同时我们需要判断一下这个队列是不是在 ready 中如果是的话我们修改完优先级应该重新 sort 一下 ready 队

列。因为此处更改优先级可能会破坏 ready 队列的顺序，同样的我们改完优先级需要判断抢占。

那么现在需要编写还原优先级函数 thread_back_priority():

```
thread_back_priority (struct thread *t)
{
    enum intr_level old_level = intr_disable ();
    if(list_empty(&t->locks))
        t->priority=t->old_priority;
    else
    {
        list_sort (&t->locks,&lock_cmp_priority,NULL);
        t->priority=list_entry (list_begin(&t->locks), struct lock, elem)->priority;
        t->priority=t->old_priority > t->priority? t->old_priority:t->priority;
    }
    intr_set_level (old_level);
    thread_preempt_priority();
}
```

我们需要判断线程拥有的锁队列是不是空如果是的话意味着当前线程没有别的锁了，那么我们直接恢复原优先级就可以了，但是如果其不为空那么我们就需要判断一下队列中优先级最大的锁和线程的原优先级哪个大那么就把大的赋值给优先级。同样的我们需要在修改优先级之后判断抢占。我们需要 sort 锁的最大优先级那么我们需要写一个 cmp 函数用于比较锁的优先级：

```
lock_cmp_priority(const struct list_elem *elem1,
                  const struct list_elem *elem2,void *aux)
{
    return list_entry(elem1,struct lock,elem)->priority
           > list_entry(elem2,struct lock,elem)->priority;
}
```

到了现在我们只剩下一条第五点没有修改，那么这里就需要修改 thread_set_priority() 函数：

```
thread_set_priority (int new_priority)
{
    struct thread *t = thread_current();
    enum intr_level old_level = intr_disable ();
    if(t->priority==t->old_priority||new_priority>t->priority)
        t->priority=new_priority;
    t->old_priority=new_priority;
    thread_back_priority();
    intr_set_level (old_level);
}
```

我们增加一个逻辑判断如果当前的优先级=原优先级那么意味着没有出现优先级捐赠的情况，如果出现了优先级捐赠但是新的优先级大于捐赠优先级那么我们就不需要捐赠优先级了，所以这两种情况都可以修改现优先级。不论什么时候调用 set 函数我们都应该能修改到原优先级，因为最后的还原是还原成旧优先级的。之后我们需要考虑一个情况，如果当前线程拥有锁，而此时优先级又被调低到低于锁的优先级，那么此时我们就应该捐赠，所以我们在后面调用 thread_back_priority() 来判断最大的锁的优先级是否大于现在优先级是的话进行捐赠即可。

因为我们很经常使用到判断抢占所以我们封装抢占判断为一个函数，直接调用 thread_preempt_priority():

```

thread_prempt_priority ()
{
    enum intr_level old_level = intr_disable ();
    if(list_entry(list_begin(&ready_list), struct thread, elem)
        ->priority > thread_current()->priority)
        thread_yield();
    intr_set_level (old_level);
}

```

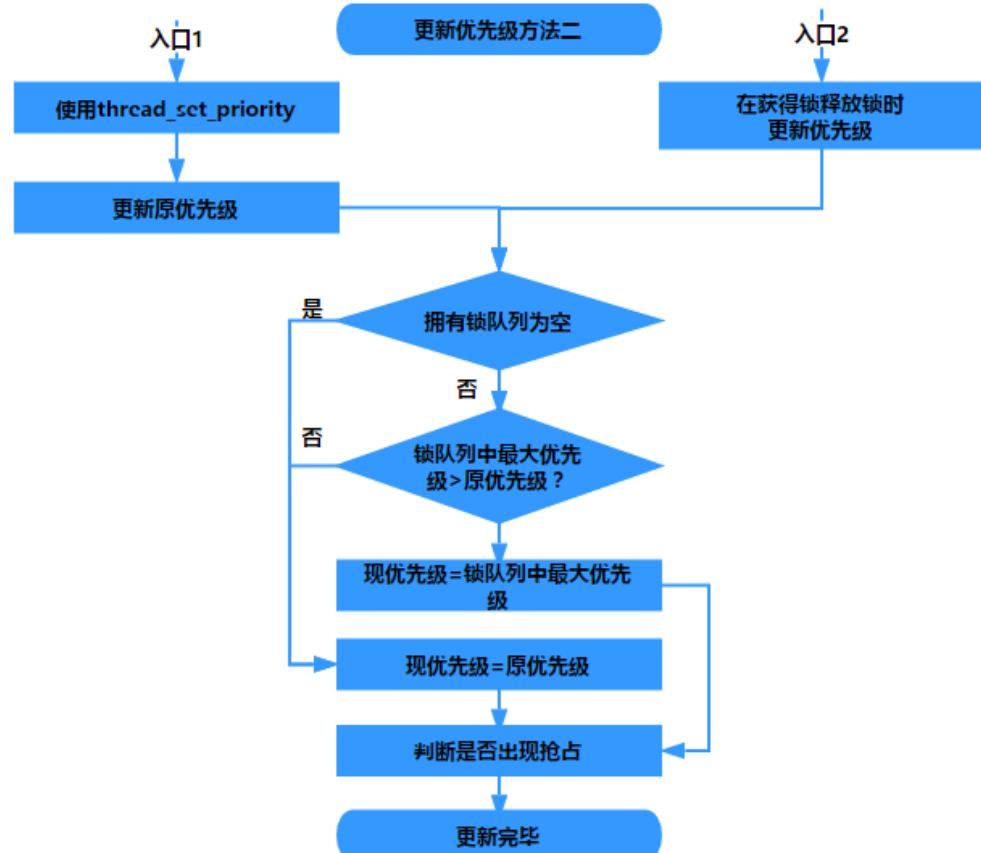
直接判断 ready 队列中的第一个线程，也就是最大优先级的线程与当前优先级的大小来判断是否需要抢占。

4) 修改优先级-方法 2

从上面方法一的修改中，首先我们知道只有在使用 `set_priority` 函数的才会修改到原优先级；其次现优先级=Max(原优先级，拥有锁队列中的最大优先级)，这个结论的原因是如果出现捐赠优先级的情况，那么此时的优先级就是=锁队列中的最大的锁的优先级（我们之前定义锁的优先级为被这个锁阻塞的最大优先级的线程的优先级），同时如果非捐赠情况那么就是现优先级=原优先级，所以我们可以不用判断是不是出现了优先级捐赠，我们直接取原优先级和拥有锁队列中的最大优先级中的最大值即可。

综上所述我们可以直接得出第二种也是最简单的修改优先级的方法，不需要多少判断，直接在获取锁和释放锁的时候调用这个函数，让这个函数自己做出选择，同时如果调用 `set_priority` 函数那么此时先修改原优先级在调用这个函数。

流程逻辑如下：



函数实现如下：

```
thread_set_priority (int new_priority)
{
    struct thread *t = thread_current();
    t->old_priority=new_priority;                                //method two
    thread_change_priority(t);
}
void
thread_change_priority(struct thread *t)                         //method two
{
    enum intr_level old_level = intr_disable ();
    if(list_empty(&t->locks))
        t->priority=t->old_priority;
    else
    {
        list_sort (&t->locks,&lock_cmp_priority,NULL);
        t->priority=list_entry (list_begin(&t->locks), struct lock, elem)->priority;
        t->priority=t->old_priority > t->priority? t->old_priority:t->priority;
    }
    if(t!=thread_current())
        list_sort (&ready_list,&thread_cmp_priority,NULL);
    intr_set_level (old_level);
    thread_preempt_priority();
}
```

Set 的修改我们之前分析的时候已经说了就是直接修改原优先级在调用我们新写的函数。现在解释一下这个函数，首先锁队列为空那么不用考虑了，现优先级=原优先级；锁队列非空的现优先级=Max(原优先级，拥有锁队列中的最大优先级)。之后如果当前线程在 ready 队列中最好 sort 一下 ready，因为其顺序可能会改变。我们除了 test 中修改优先级调用 set 之外，锁的获取释放都是直接调用这个实现的 change 函数（此函数功能和方法一中的 back 函数实际上是一样的，实现也是一样的）。

3. 实验结果

1) 完成后 make check 结果

```
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-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] Error 1
root@15352008慕荣裕:~/pintos/src/threads/build#
```

2) Priority-donate-one 结果

```
(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.

root@15352008:~/pintos/src/threads/build# =====
```

3) Priority-donate-multiple 结果

```
(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.

root@15352008:~/pintos/src/threads/build# =====
```

4) Priority-donate-multiple2 结果

```
(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 order.
(priority-donate-multiple2) Main thread should have priority 31. Actual priority: 31.
(priority-donate-multiple2) end
Execution of 'priority-donate-multiple2' complete.

root@15352008:~/pintos/src/threads/build# =====
```

5) Priority-donate-nest 结果

```
(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.

root@15352008:~/pintos/src/threads/build# =====
```

6) Priority-donate-sema 结果

```
(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.

root@15352008:~/pintos/src/threads/build# =====
```

7) Priority-donate-lower 结果

```
(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.

root@15352008慕荣裕:~/pintos/src/threads/build# =====
```

8) Priority-sema 结果

```
(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.

root@15352008慕荣裕:~/pintos/src/threads/build# =====
```

9) Priority-donate-chain 结果

```
(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.

root@15352008慕荣裕:~/pintos/src/threads/build# =====
```

4. 回答问题

➤ 1. 如何解决嵌套捐赠的问题？

我们可以使用类递归的方法实现嵌套捐赠的问题，我们只要考虑到当前捐赠完优先级之后会不会影响到阻塞当前锁的拥有者的锁的优先级（造成捐赠优先级的更新），如果会影响的话我们就要去修改那个锁的优先级和那个锁的拥有者的优先级，同时还要更新阻塞当前锁的拥有者的锁，循环上述过程即可解决。

所以我们对于这个问题我们就需要为线程增加一个结构体来记录阻塞这个线程的锁，这样就能往回找到阻塞的源头。对于嵌套的解决方法如下：

```
while(锁不为空 && 锁的拥有者的优先级 < 当前线程的优先级){  
    锁的优先级改为当前线程的优先级；  
    锁的拥有者的优先级改为当前线程的优先级；  
    锁=阻塞锁拥有者的锁；  
}
```

➤ 2. 如何解决一个线程占有了多个锁的问题？

我们可以对线程结构体增加一个锁队列，用来记录这个线程锁拥有的锁，同时我们为锁增加一个优先级用来记录他所阻塞的最大优先级线程的优先级。

因为我们之前提到过，出现捐赠优先级的时候，当前线程的优先级是取他所有的锁里的最大优先级，所以此时我们直接对这个队列取其最大的优先级的锁的优先级赋值给现优先级即可，此处用 sort 取队列头实现。

当我们在释放锁的时候，我们需靠考虑其是否还有所，这些锁会不会造成优先级捐赠，那么此时我们还是取锁队列中的最大锁的优先级与线程的原优先级比较，谁大就把谁赋值给现优先级。

➤ 3. 在实现优先级捐赠之后，在 `thread_set_priority` 中需要考虑哪几种情况？分别怎么处理？

如果按照最直接的做法我们根本不需要考虑情况，当我们调用 `set` 的时候一定先改动原优先级，因为不论是否捐赠状态原优先级在调用了 `set` 之后是一定会被修改为传入的优先级的，之后我们只要考虑现优先级的修改，但是我们知道现优先级不是等于原优先级就是等于锁队列中最大优先级的锁的优先级（也就是捐赠优先级），此时我们只要取二者中的最大值即可（这部分调用函数实现）。

如果按照我的实验最开始的思路来说，要考虑 `thread_set_priority` 中的情况，因为我的 `set` 函数主要负责修改原优先级，不负责修改捐赠优先级，所以只会出现 4 种情况：

1) 非捐赠状态：

- a) 新优先级 \geq 现优先级=原优先级时，同时将两个优先级改为新优先级。
- b) 现优先级=原优先级 $>$ 新优先级，同时将两个优先级，再判断是否会因为优先级降低而出现捐赠状态，如果会的话降现优先级改为捐赠优先级（比较新优先级与其锁队列中的最大优先级）。

2) 捐赠状态：

- a) 新优先级 \geq 现优先级，同时将两个优先级改为新优先级，解除捐赠。
- b) 现优先级 $>$ 新优先级，只修改原优先级。

所以实际上应被总结为 3 种情况，只要新优先级 \geq 现优先级，都是同时将两个优先级改为新优先级，不论他们现在是否捐赠状态，经过这次修改都会直接变成非捐赠状态，可直接视为一种情况。