中山大学移动信息工程学院本科生实验报告

（2017 学年春季学期）

课程名称：Operating System　　任课教师：饶洋辉　　批改人(此处为 TA 填写)：

年级+班级	1501	专业（方向）	移动信息工程
学号	15352006	姓名	蔡丽芝
电话	13538489980	Email	314749816@qq.com
开始日期	2017/5/8	完成日期	2017/5/11

1. 实验目的

1）　A 解决优先级反转的问题，实现优先级捐赠

2）　B 通过除了 priority-condvar 之外的 priority 测试指令，make check 结果为”8/27”

2. 实验过程

(一)Test 分析（30 分）

实验涉及到的相关函数的分析(未修改代码之前)：

```
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
```

定义了信号量 semaphore,每个信号量都拥有一个信号量值和一个等待队列，信号量值表示当前临界区还可以进入的线程实体数。

```
void
sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema != NULL);

  sema->value = value;
  list_init (&sema->waiters);
}
```

对信号量进行初始化，初始化信号量值和等待队列

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_push_back (&sema->waiters, &thread_current ()->elem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

Sema_down()函数用于进行 p 操作（原子操作），中断屏蔽，进入一个 while 循环，循环条件是 sema->value = 0，即信号量的值为 0，表示已无可用的临界区资源，因此此时需要将当前正在运行的线程插入到等待队列中，调用 thread_block 函数，把线程从运行状态切换

到阻塞状态，进入等待队列。当发现 sema->value != 0 时，此时线程可以进入到临界区，跳出循环或不会进入到循环中，将可进入临界区的实体数量减一，即 sema->value--.

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  sema->value++;
  intr_set_level (old_level);
}
```

Sema_up()函数进行 v 操作（原子操作),中断屏蔽，进入一个判断语句，如果信号量的等待队列不为空，就从等待队列的头部取出线程放入就绪队列中，唤醒线程，sema->value++.

```
/* Lock. */
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
  };
```

struct lock 定义锁，锁是一个特殊的信号量，每一个锁都有一个信号量和一个线程指针，这个线程指针指向的是拥有这个锁的线程。

```
void
lock_init (struct lock *lock)
{
  ASSERT (lock != NULL);

  lock->holder = NULL;
  sema_init (&lock->semaphore, 1);
}
```

对传进来的锁 lock 进行初始化，首先这个锁的拥有者 holder = NULL，并且将信号量的值初始化为 1，此时临界区中只允许一个线程进入，信号量对与的等待队列为空。

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

  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
}
```

Lock_acquire()函数进行锁的 p 操作，这个函数传入了所要请求锁的指针 lock，首先检查锁是否存在，如果存在再判断当前线程是否已经拥有了这把锁了，很明显如果已经拥有了这把锁，就不存在再次请求这把锁的必要，满足了上面的条件后，调用 sema_down 函数，如果 lock 中 semaphore 的信号量值为 1，此时说明这把锁，没被其他线程申请，线程可获得这把锁，最后将锁的 holder 设为当前线程，然而当信号量的值为 0 时，说明锁已经被其他线程申请，此时申请这把锁的线程会被阻塞。

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

  lock->holder = NULL;
  sema_up (&lock->semaphore);
}
```

Lock_release()函数进行锁的 v 操作，这个函数传入了所要释放的锁的指针 lock，首先仍然是检查锁是否存在，如果存在，再检查是否是当前线程拥有这把锁，这步很重要，必须要保证是当前线程拥有这把锁，然后将 lock->holder 设为 NULL，调用 sema_up 函数，唤醒被阻塞的线程，将阻塞的线程放入就绪队列中

Test 1: priority-donate-one

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

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

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

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

创建了一个锁 lock，将这个锁初始化，当前测试线程调用 lock_acquire()函数获得该锁，并且测试线程的优先级为 PRE_DEFAULT;

```
  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 ());
```

创建了一个名为”acquire1”的线程，优先级为 PRE_DEFAULT+1，优先级高于测试线程，测试线程退出 cpu，acquire1 抢占调用执行 acquire1_thread_func 函数，

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

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

在 acquire1_thread_func 函数中，调用 lock_acquire 函数获取锁，但是锁已经被测试线程获得，因此此时 lock 中的信号量值为 0，在 lock_acquire 函数中会调用 sema_down()函数，正如开始分析的，sema_down 函数会阻塞掉 acquire2,放入等待队列中，acquire1_thread_func 函数被阻塞，msg 中的内容不会输出。此时测试线程获得 cpu，从上次中断的地方开始执行,输出测试线程应该的优先级和实际的优先级,分别为 32,32(①)，因为实现优先级捐赠后，被阻塞的线程 acquire1 会将其优先级赋给测试线程，即 32.

```
  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 ());
```

创建名为”acquire2”，优先级为 33，优先级高于测试线程，抢占调用，执行 acquire2_thread_func 函数

```
static void
acquire2_thread_func (void *lock_)
{
  struct lock *lock = lock_;

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

此函数中调用 lock_acquire 函数获取锁 lock,同理由于 lock 已被测试线程拥有，当前线程会被阻塞，msg 不会输出，测试线程进入 cpu 执行，输出测试线程此时应该的优先级和实际的优先级，33，33(②)，因为实现优先级捐赠后，被阻塞的线程 acquire2 会将其优先级赋给测试线程，即 33

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

测试线程调用 lock_release 函数释放锁，此时在等待队列中的线程中 acquire2 的优先级最高，所以先被唤醒，放入就绪队列中，acquire2 获得锁，输出信息”acquire2 got the lock(③)”，调用 lock_release()函数释放锁，输出 msg”acquire2 done(④)”，acquire1 被唤醒，进入就绪队列中，执行 acquire1_thread_func 函数获得锁 lock，输出信息“acquire1 got the lock(⑤)”，调用 lock_release()函数释放锁，输出 msg“acquire1 done(⑥)”，最后测试线程拿到 cpu 继续执行，输出截图中的两条信息（⑦⑧）。

因此输出信息的打印顺序结果为：（**红字下划线中序号的输出顺序**）

```
(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 ord
er.
(priority-donate-one) This should be the last line before finishing this test.
(priority-donate-one) end
```

Test 2: priority-donate-multiple

```
void
test_priority_donate_multiple (void)
{
  struct lock a, b;

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

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

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

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

创建锁 a，锁 b，当前测试线程的优先级为 31，调用 lock_acquire 函数初始化锁 a，锁 b，调用 lock_aquire 函数获得锁 a，锁 b

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

创建线程 a，优先级为 32 大于测试线程(31)，抢占调用，锁 a 作为参数传入函数 a_thread_func 中，

```
static void
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.");
}
```

调用 lock_acquire 函数申请锁 a，但发现锁 a 已被使用，将自身较高的优先级赋值给拥有锁 a 的线程即测试线程，a 线程被阻塞，msg 信息无法输出。测试线程获得 cpu 执行输出测试线程此刻应该和实际的优先级，为 32,32(①)，测试线程得到了来自 a 线程的优先级捐赠。

```
  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 ());
```

测试线程创建线程 b，优先级为 33 大于测试线程(32)，抢占调用，锁 b 作为参数传入函数 b_thread_func 中

```
static void
b_thread_func (void *lock_)
{
  struct lock *lock = lock_;

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

调用 lock_acquire 函数申请锁 b，然而锁 b 已被使用，将自身较高的优先级赋值给拥有锁 b 的线程即测试线程，b 线程被阻塞，msg 信息无法输出。测试线程获得 cpu 执行输出测试线程此刻应该和实际的优先级，为 33,33(②)，测试线程得到了来自 b 线程的优先级捐赠。

```
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 ());
```

调用 lock_release 函数释放锁 b，此时线程 b 会从等待状态切换到就绪状态，线程 b 获得 cpu，执行 b_thread_func 获得锁 b 输出 msg”b acquired lock b(③)”，调用 lock_release 函数释放锁 b，输出信息”thread b finished(④)”，线程 b 执行完毕后，测试线程进入 cpu，输出上面截图的两条 msg(⑤)，输出应该的优先级和实际优先级，32，32(⑥)。测试线程的优先级变为 32，是因为释放锁 b 后，在测试线程中，其拥有的锁队列中优先级最高的锁为 a，其优先级为 32，在实现优先级捐赠后，锁 a 会将其优先级捐赠给测试线程，所以此时测试线程的优先级为 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 ());
```

测试线程释放锁 a，此时线程 a 从阻塞切换到就绪状态，线程 b 获的 cpu，执行 a_thread_func 函数，获得锁 a，输出 msg”a acquired lock a(⑦)”，调用 lock_release 函数释放锁 a，输出 msg“a finished(⑧)”由于此时测试线程的锁队列为空，测试线程会恢复到原来的优先级，即 31. 线程 a 执行完后，测试线程获得 cpu，执行输出上边截图的两条 msg(⑨)，输出应该的优先级和实际优先级，31，31(⑩)，测试线程将锁 a，锁 b 都释放了，这时恢复到原来的优先级。

因此输出信息的打印顺序结果为：（**红字下划线中序号的输出顺序**）

```
(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.
```

Test 3: priority-donate-multiple2

```
void
test_priority_donate_multiple2 (void)
{
  struct lock a, b;

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

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

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

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

创建两个锁 a，b，测试线程的优先级为 31，调用 lock_init 函数分别初始化锁 a 和锁 b，调用 lock_acquire 函数拥有锁 a 和 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 ());
```

创建线程 a，优先级为 34 高于测试线程，将锁 a 作为参数传入 a_thread_func 中，抢占调用，执行 a_thread_func 函数

```
static void
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.");
}
```

申请锁 a 失败,线程 a 被阻塞，a 线程将其优先级捐赠给测试线程，测试线程拿到 cpu 继续执行输出 msg“Main thread should have priority 34. Actual priority: 34（①）.”

```
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 ());
```

创建线程 c，优先级为 32，由于经过线程 a 的优先级捐赠后，测试线程的优先级为 34 大于线程 c，线程 c 不会抢占调用。创建线程 b，优先级为 36，将锁 b 作为参数传入 b_thread_func 函数中，线程 b 优先级高于测试线程，抢占调用，执行 b_thread_func 函数

```
static void
b_thread_func (void *lock_)
{
  struct lock *lock = lock_;

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

申请锁 b 失败，线程 b 被阻塞，b 线程将其优先级捐赠给测试线程，测试线程拿到 cpu 继续执行输出 msg“Main thread should have priority 36. Actual priority: 36（②）.”

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

调用 lock_release 函数释放锁 a，a 线程被唤醒，回到就绪队列中，然而测试线程的优先级高于线程 a，所以测试线程线获得 cpu 继续执行，输出 msg(“Main thread should have priority 36. Actual priority: 36(③).)

```
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 ());
```

调用 lock_release 函数释放锁 b，此时测试线程的锁队列为空，测试线程优先级恢复为原来的优先级即 31，b 线程被唤醒，回到就绪队列中，在就绪队列中 b 线程的优先级最高，因此线程 b 获得 cpu 继续执行，获得锁 b，输出 msg(“Thread b acquired lock b.(④)”)释放锁 b，输出 msg(“Thread b finished.（⑤）”);线程 b 执行完后，在就绪队列中，线程 a 的优先级(34)高于测试线程的优先级(31)，线程 a 获得 cpu 继续执行函数 a_thread_func，获得锁 a，输出 msg (“Thread a acquired lock a.(⑥)”);释放锁 a，输出 msg(“Thread a finished.（⑦）”)，线程 a 执行完后，在就绪队列中线程 c 的优先级高于测试线程，线程 c 先获得 cpu 执行函数

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

输出 msg (“Thread c finished.（⑧）”);线程 c 执行完后，就绪队列里就剩下测试线程，测试线程执行输出 msg (“Threads b, a, c should have just finished, in that order.（⑨）”)，msg (“Main thread should have priority 31. Actual priority: 31.（⑩）”)

因此输出信息的打印顺序结果为：（**红字下划线中序号的输出顺序**）

```
Boot complete.
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.
```

Test 4: priority-donate-nest

```
struct locks
  {
    struct lock *a;
    struct lock *b;
  };
```

定义了 locks 结构体，在 locks 中有锁 a 和锁 b

```
void
test_priority_donate_nest (void)
{
  struct lock a, b;
  struct locks locks;

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

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

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

  lock_acquire (&a);

  locks.a = &a;
  locks.b = &b;
```

创建锁 a，锁 b，创建一个 locks 结构体,测试线程的优先级为 31，调用 lock_init 函数初始化锁 a 和锁 b，调用 lock_acquire 函数获取锁 a，对结构体 struct locks 进行初始化，将测试线程创建的两个锁分别赋值给 struct locks 内的锁。注意：测试线程只拥有了锁 a，没有拥有锁 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 ());
```

创建线程 medium，优先级为 32，将结构体 locks 作为参数传入函数 medium_thread_func 中，medium 优先级高，抢占调用

```
static void
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.");
}
```

调用 lock_acquire 函数申请锁 b，因为锁 b 未被其他线程占有，所以 medium 线程申请成功，拥有锁 b，调用 lock_acquire 函数申请锁 a，medium 将自身的优先级捐赠给锁 a 的拥有者测试线程，同时 medium 被阻塞，测试线程获得 cpu，执行调用 thread_yield()函数，测试线程退出 cpu 重新回到就绪队列中，因为此时测试线程的优先级最高，所以调度的仍

然是测试线程，测试线程继续执行，输出信息 msg ("Low thread should have priority 32. Actual priority: 32. (①)")

```
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 ());
```

创建线程 high，优先级为 33，将锁 b 作为参数传入函数 high_thread_func,high 优先级高抢占调度，执行 high_thread_func 函数

```
static void
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.");
}
```

调用 lock_acquire 函数申请锁 b，被阻塞同时 high 线程将自身优先级捐赠给锁 b 的拥有者线程 medium，线程 medium 将自己的优先级捐赠给锁 a 的拥有者测试线程，此时测试线程优先级为 33，在就绪队列中，测试线程获得 cpu 继续执行，调度 thread_yield 函数，同理由于在就绪队列中测试线程的优先级仍然最高，所以重新调度的依旧是测试线程，输出 msg ("Low thread should have priority 33. Actual priority: 33(②).)

```
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 ());
```

测试线程调用 lock_release 函数释放锁 a，此时测试线程的 locks 队列为空，测试线程回复到原来的优先级即 31，同时 mediu 线程被唤醒，从等待队列中回到就绪队列中，由于 medium 线程的优先级高于测试线程，发生优先级抢占，medium 测试线程获得 cpu，重新执行 medium_thread_func 函数，获得锁 a，输出 msg ("Medium thread should have priority 33. Actual priority: 33.(③)"),输出 msg ("Medium thread got the lock.(④)");释放锁 a，此时 medium 线程的 locks 队列中只剩下锁 b，medium 的优先级被赋值为锁 b 的优先级即 32,调用 thread_yield 函数,就绪队列中 medium 的优先级高于测试线程(31),medium 依旧获得 cpu,调用 lock_release 函数释放锁 b,此时 medium 线程中的锁队列为空,medium 恢复原来的优先级 32 同时线程 high 被唤醒从等待队列进入就绪队列中,调用 thread_yield 函数，线程 medium 退出 cpu 进入就绪队列中，此时线程 high 的优先级高于线程 medium，因此，线程 high 获得 cpu，线程 high 继续执行 high_thread_func 函数，获得锁 b 输出 msg ("High thread got the lock.(⑤)")，释放锁 b，输出 msg ("High thread finished(⑥).");线程 high 执行完后，在就绪队列中的线程 medium 优先级高获得 cpu，输出 msg ("High thread should have just finished.(⑦)");msg ("Middle thread finished.)⑧)")。最后测试队列执行，输出 msg('Medium thread should just have finished.(⑨)')
Msg('Low thread should have priority 31. Actual priority: 31(⑩)');
因此输出信息的打印顺序结果为：（**红字下划线中序号的输出顺序**）

```
Executing 'priority-donate-nest':
(priority-donate-nest) begin
(priority-donate-nest) Low thread should have priority 32.  Actual priority: 32.
(priority-donate-nest) Low thread should have priority 33.  Actual priority: 33.
(priority-donate-nest) Medium thread should have priority 33.  Actual priority:
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.
```

Test 5: priority-donate-sema

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

定义了一个结构体 lock_and_sema，其中包含一个锁变量 lock,信号量 sema;

```
void
test_priority_donate_sema (void)
{
  struct lock_and_sema ls;

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

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

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

创建结构体 lock_and_sema ls，测试线程的优先级为 31，初始化结构体 ls 的锁，并且初始化结构体 ls 的信号量，初始信号量值为 0。

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

创建线程 low，优先级为 32，将结构体 ls 作为参数传入函数 l_thread_func 中，线程 low 的优先级高于测试线程，抢占调度，执行 l_thread_func 函数

```
static void
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.");
}
```

调用 lock_acquire 函数申请结构体 ls 中的锁，获得锁，输出 msg（"Thread L acquired lock(①)."）调用 sema_down 函数，因为 ls->sema = 0，所以线程 low 被阻塞，测试线程获得 cpu，重新在上次中断的地方开始执行

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

创建线程 med,优先级为 34，将结构体 ls 传入函数 m_thread_func,线程 med 优先级高于测试线程，抢占调度，执行 m_thread_func 函数

```
static void
m_thread_func (void *ls_)
{
  struct lock_and_sema *ls = ls_;

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

调用 sema_down 函数被阻塞，测试线程获得 cpu，继续执行。

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

创建线程 high，优先级为 36，将结构体 ls 作为参数传入函数 h_thread_func,线程 high 的优先级高于测试线程，抢占调度

```
static void
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.");
}
```

调用 lock_acquire 获取结构体 ls 中的锁，被阻塞同时线程 high 将自身的优先级捐赠给线程 low，此时 low 的优先级为 36，测试线程获得 cpu，执行

```
  sema_up (&ls.sema);
  msg ("Main thread finished.");
```

调用 sema_up 函数进行 v 操作，在 ls->sema 的等待队列中，由于线程 low 的优先级高于线程 med，所以线程 low 先被唤醒，进入就绪队列中，线程 low 的优先级高，获得 cpu 运行，继续执行 l_thread_func 函数，输出 msg（"Thread L downed semaphore.(②)"）.释放结构体中的锁同时线程恢复原来的优先级，线程 high 被唤醒进入到就绪队列中，获得 ls 结构体中的锁，输出 msg（"Thread H acquired lock.(③)"）;调用 sema_up 函数，唤醒等待队列中的线程 m 进入就绪队列中，线程 high 调用 lock_release 函数释放锁，输出 msg（"Thread H finished.(④)"）;线程 high 执行完后，在就绪队列中，线程 med 的优先级高于测试线程，高于线程 low，所以线程 med 获得 cpu，执行 m_thread_func 函数输出 msg（"Thread M finished(⑤)."）；线程 low 获得 cpu，输出 msg（"Thread L finished(⑥)."）;最后测试线程获得 cpu，继续执行，输出 msg("Main thread finished.⑦")；

因此输出信息的打印顺序结果为：（**红字下划线中序号的输出顺序**）

```
Boot complete.
Executing 'priority-donate-sema':
(priority-donate-sema) begin
(priority-donate-sema) Thread L acquired lock.
(priority-donate-sema) Thread L downed semaphore.
(priority-donate-sema) Thread H acquired lock.
(priority-donate-sema) Thread H finished.
(priority-donate-sema) Thread M finished.
(priority-donate-sema) Thread L finished.
(priority-donate-sema) Main thread finished.
(priority-donate-sema) end
Execution of 'priority-donate-sema' complete.
```

Test 6: priority-donate-lower

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

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

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

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

创建锁 lock，测试线程的优先级为 31，初始化锁 lock，调用 lock_acquire 函数获得锁 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 ());
```

创建线程 acquire，优先级为 41，将锁 lock 作为参数传入 acquire_thread_func 函数中，acquire 线程优先级高，抢占调用

```
static void
acquire_thread_func (void *lock_)
{
  struct lock *lock = lock_;

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

申请锁不成功，线程 acquire 被阻塞同时将 acquire 自身的优先级捐赠给锁 lock 的拥有者测试线程，测试线程优先级变为 41，测试线程获得 cpu，继续执行输出 msg（"Main thread should have priority 41. Actual priority: 41.(①)"）

```
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 ());
```

测试线程输出 msg（"Lowering base priority...(②)");降低优先级至优先级变为 31，由于优先级捐赠的实现，为保证测试线程能释放锁，测试线程的优先级不会发生改变，改变的是测试线程的 old_priority，因此输出 msg（"Main thread should have priority 41. Actual priority: 41(③)."），测试线程的优先级仍然为 41。

```
  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 ());
```

测试线程释放锁 lock，恢复到原来的优先级也就是 PRE_DEFAULT-10(21)，线程 acquire 被唤醒，获得锁 lock，输出 msg（"acquire: got the lock(④)");释放锁 lock，输出 msg（"acquire: done(⑤)");线程 acquire 执行往后，测试线程获得 cpu，执行输出 msg（"acquire must already have finished.(⑥)"）；输出 msg（"Main thread should have priority 21. Actual priority:21(⑦)."），

因此输出信息的打印顺序结果为：（**红字下划线中序号的输出顺序**）

```
Executing 'priority-donate-lower':
(priority-donate-lower) begin
(priority-donate-lower) Main thread should have priority 41.  Actual priority: 4
1.
(priority-donate-lower) Lowering base priority...
(priority-donate-lower) Main thread should have priority 41.  Actual priority: 4
1.
(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: 2
1.
(priority-donate-lower) end
Execution of 'priority-donate-lower' complete.
```

Test 7: priority-sema

```
void
test_priority_sema (void)
{
  int i;

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

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

调用 sema_init 函数初始化信号量，信号量的初始化值为 0，调用 thread_set_priority 函数降低自身优先级至最低。

```
  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);
    }
```

进入第一次循环，创建优先级为 27 的线程 priority 27,优先级高于测试线程，抢占调用，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 ());
}
```

Sema_down 执行 p 操作，线程 priority 27 被阻塞，测试线程获得 cpu，继续创建线程与上述相同，因此在上面截图的 10 次循环中，测试线程创建了优先级分别为 27,26,25,24,23,22,21,30,29,18 的线程，都被 p 操作阻塞，插入到等待队列中，最后一个创建的线程被阻塞后，测试线程获得 cpu 继续执行

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

进入第一次循环，调用 sema_up 将等待队列的头部线程唤醒放入就绪队列中，在等待队列中已经按照优先级排好序，在头部的是优先级最大的线程，因此第一次 v 操作唤醒的线程 priority30,线程 priority30 在就绪队列中优先级高位于头部，获得 cpu 执行输出信息：Thread priority 30 wokeup”，执行完毕后，测试线程获得 cpu，在上次中断的地方开始执行，输出信息：Back in main thread， 重复上述操作，按照优先级大小依次唤醒在等待队列中被阻塞的线程。

因此运行顺序是：

```
simmon@15352006lizhicai: ~/pintos/src/threads/build
(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.
```

Test 8: priority-donate-chain

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

创建声明结构体 lock_pair，该结构体中包含两个 lock 指针。

```
void
test_priority_donate_chain (void)
{
  int i;
  struct lock locks[NESTING_DEPTH - 1];
  struct lock_pair lock_pairs[NESTING_DEPTH];

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

  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 ());
```

创建一个数据类型为 lock 的 locks 数组，其 size = 7，创建一个数据类型为 struct lock_pair 的 lock_pairs 数组，其 size 为 8；调用 thread_set_priority 函数降低优先级至 0， 调用 lock_init 初始化 locks 数组中的每一个锁,调用 lock_acquire 函数获得锁 locks[0],输出 msg(“main got lock(①)”);

```
  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);
    }
```

进入一个 7 次循环中，循环中的 thread_priority 依次等于 3,6,9,12,15,18,21，然后对应的 locks_pairs[i].first 记录的是 locks[i]的锁，locks_pairs[i].second 记录的是 locss[i-1]的锁，这里注意一点：locks_pairs[7].first = locsk[7]=NULL. 接着调用 thread_create 函数创建优先级为 thread_priority 的线程，并将 locks_pairs[i]作为参数传入到 doner_thread_func 函数中，由于测试线程的优先级最低为 0，因此每次创建新的线程的时候，测试线程都会退出 cpu，新线程抢占调用。执行 donor_thread_func 函数

```
static void
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 ());
}
```

举第一次进入循环，创建线程为 priority 3 为例子，首先判断 locks 结构体中 locks->first 是否为空，不会空时，申请后成功获得 locks 中的 first 锁，接着申请获得 second 锁，second 锁即为 locks[0](被测试线程拥有),因此申请锁失败，被阻塞，测试线程继续第二次循环，分析可知，每一次创建线程抢占调用执行 donor_thread_func 函数时，申请获得锁 first 均成功，申请锁 second 时，由于 second 是前一个，而前一个锁的拥有者一定是前一次循环创建的线程，因此每一次调用 lock_acquire 函数获得锁 second 时都会被阻塞，第 7 次循环时，由于 locks[7] = NULL.所以第 7 次执行 donor_thread_func 函数时，创建的线程没有获得锁 first,但是申请锁 second 依旧失败，被阻塞。

每一次循环创建线程申请锁 second 被阻塞后，测试线程获得 cpu 继续执行，输出 msg ("%s should have priority %d. Actual priority: %d.") 因为优先级捐赠，所以测试线程的优先级每一次都提升了，当创建线程的此时大于 1，存在锁循环嵌套的情况时，也会进行嵌套的捐赠，会将优先级最高被阻塞的线程一直捐赠到测试线程中，因此：每一次输出 msg 时，输出的优先级都为创建线程时赋给线程的优先级，即 thread_priority.接着每次循环最后都会调用 thread_create 函数创建线程 interploper i，新创建的优先级为 thread_priority-1，由于测试线程的优先级被提升了为 thread_priority，所以新创建的线程不会发生优先抢占。至此，循环创建 7 个线程后，并且每个创建的线程都被阻塞，同时创建了 7 个 interloper 线程放入就绪队列中。此时测试线程的优先级为 21。

```
lock_release (&locks[0]);
msg ("%s finishing with priority %d.", thread_name (),
                                       thread_get_priority ());
```

测试线程调用 lock_release 函数释放锁 locks[0],测试线程优先级恢复到原来即最低为 0，同时唤醒被阻塞在等待队列中的线程 thread 1,thread 1 优先级高获得 cpu，继续执行 donor_thread_func 函数，获得锁 second，也就是 locks[0],输出信息 thread 1 got lock 随后调用 lock_release 函数释放锁 second，由于该线程仍让占有锁 first，所以该线程仍然被后面的线程捐赠，同理后面的线程也被其后面的线程捐赠，所以该线程的优先级为 21，输出信息 thread 1 should have priority 21, Acutal priority 21. 当线程的 locks->first 不为空时，调用 lock_release 函数释放锁 first，此时 thread 1 优先级恢复为原来的优先级 3 同时唤醒等待队列中位于头部的线程 thread 2，thread2 优先级高抢占调用获得 cpu，继续执行 donor_thread_func,获得上个创建线程释放的锁，输出 thread 2 got the lock,释放锁 second,同理该线程仍然占有锁 first，所以该线程被后面的线程捐赠，同理后面的线程也被其后面的线程捐赠，所以该线程的优先级也为 21， 输出信息 thread 2 should have priority 21, Acutal priority 21.释放锁 first，同时优先级恢复原来的优先级且唤醒下一个线程，过程与上述相同，直到唤醒最后一个线程 thread 7 时，

thread 7 的锁 first 为空，因此会直接输出 threa 7 fininshing with priority 21，线程 7 执行往后，在就绪队列中，此时队列从左到右优先级依次减小，为：
interloper 7，thread 6，interloper 6，thread 5，interloper 5，thread 4，interloper 4，thread 3，interloper 3，thread 2，interloper 2，thread 1，interlper 1，main thread
因此 interloper7 获得 cpu，执行调用

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

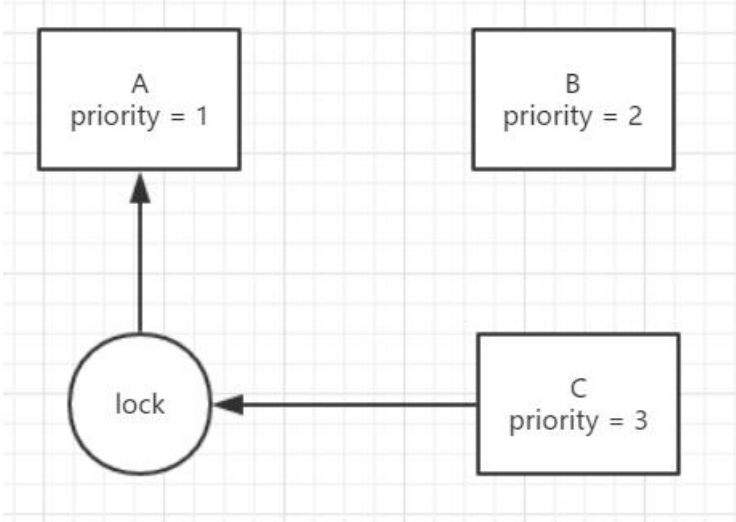
输出 interloper 7 finished。
接着 thread 6 获得 cpu 输出 thread 6 finishing the priority 18。
接下的线程按照上述就绪队列中的顺序依次获得 cpu 输出相应的信息。
因此结果的输出顺序如图：

```
Executing 'priority-donate-chain':
(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.
```

(二)　分析及实现（30 分）

分析：为什么要实现优先级捐赠？

因为存在一种优先级反转的现象，何为优先级反转？(参考 ppt)



锁 A 拥有锁 lock，A 的线程优先级最低，因为 C 的优先级高于 A 的优先级，所以 C 会抢占调用，导致线程 A 无法将拥有的锁释放，又因为线程 C 需要锁，但是由于 A 无法释放锁，C 得不到锁，所以 C 也无法执行，被挂起，然而线程 B 不需要 A 的锁，优先级也比 A 的高，所以 B 会顺利执行，此时线程 B 的优先级小于线程 C，但是线程 B 却先于线程 C 开始运行，这种现象就叫做优先级反转。

为什么要解决这种优先级反转的现象？(参考 ppt)

因为优先级反转会造成任务在调度的时候，无法根据优先级的大小来判断运行的时间，导致时间上无法预测任务的执行时间，造成时间判断上的不确定性，严重的时候可能会使系统崩溃。

优先级捐赠的主要思想为：我们通过一些方法，将高优先级线程的优先级捐赠给拥有高优先级线程所需锁的低线程，使得两个线程的优先级一样高，等到低优先级线程执释放锁后，再将低优先级线程的优先级恢复。(参考 ppt)

① 分析 test1 donate-priority-one 可知当线程 A 拥有锁 a，线程 B 需要锁 a，且线程 B 的优先级大于线程 A 时，线程 B 可以将自己的高优先级捐赠给线程 A，当线程 A 执行至释放锁 a 时，线程 A 恢复原来的优先级，线程 B 获得锁 a，则可顺利执行。因此很明显我们需要在 struct thread 中新增添一个变量 old_priority 用于存放线程原始的优先级，以便优先级的恢复。

② 分析 test2 donate-priority-multiple 可知线程 A 拥有锁 1 和锁 2，线程 B 需锁 1，线程 C 需锁 2，且优先级 C>B>A 时，因为线程 A 经历了多次优先级捐赠，因此线程 A 需要一个数据结构锁队列，存放 A 拥有的锁，并且每个锁拥有一个优先级，表示被阻塞线程中的最高优先级，因此此时线程被捐赠后的优先级，就是队列中优先级最高的锁的优先级。

③ 分析 test4 donate-priority-nest 可知当线程 A 获得锁 1，线程获得锁 2 同时需要锁 1，线程 C 需要锁 2，优先级 C>B>A，这种循环嵌套的情况下，首先线程 B 将优先级捐赠给 A，然后线程 C 将优先级捐赠给线程 B 再捐赠给线程 A，此时为了能够将线程 C 的优先级捐赠给 A，我们需要再 struct thread 中设一个指针变量 blocked 来指向阻塞当前线程的锁，当发生优先级捐赠的同时，不断更新 blocked，直到 blocked = NULL,

则可以停止捐赠。
④ 分析 test5 priorioty-sema 可知当进行 v 操作时，优先级高的先唤醒。即信号量的等待队列是优先级有序的队列。

通过上述分析
在 thread.h 中 struct thread 结构体中新增变量

```
/*priority donation*/
int old_priority;                   /* the priority when not donated*/
struct list locks;                  /* the list of the locks which the thread acquire*/
bool donated;                       /* whether the thread has been donated*/
struct lock* blocked;               /* point to the lock which block this thread*/
/* over */
```

构类型 locks 列表用于存放线程拥有的所有锁，用于解决当一个线程占用多把锁时，释放其中一把锁时优先级变化问题。bool 类型的 donated，用于表示线程的捐赠状态，解决优先级的捐赠和恢复。struct lock 结构类型的 blocked，blocked 指向阻塞当前线程的锁。

在 sync.h 的 struct lock 中新增成员变量

```
/* priority donation*/
int lock_priority;          /* the highest priority of the thread which is blocked by this lock*/
struct list_elem holder_elem; /* used to standy by in the struct list locks*/
```

int 类型的 lock_priority,表示的是锁拥有的优先级，解决线程多把所锁的问题。
struct list_elem 结构体类型的 holder_elem 用于在线程 locks 队列中排队。

修改 lock_aquire 函数

```
struct thread *thrd = lock->holder;
struct thread *current_thread = thread_current();
struct lock *another = lock;
current_thread->blocked = another;
```

指针 thrd 指向的是锁的拥有者，current_thread 指向的是当前在 cpu 中运行的线程，another 指向的是锁，当前在 cpu 中运行的线程的 blocked 为 another(这里在进行 p 操作后会重新赋值)

```
while(thrd != NULL && current_thread->priority > thrd->priority)
```

为了解决嵌套捐赠优先级，我们需要让线程在满足某些条件的时候，优先级能够一直捐赠下去，因此需要加入一个 while 循环
(以下思路参考 ppt)

```
/* check whether we need to donate the high priority to the thread which own the lock*/
while(thrd != NULL && current_thread->priority > thrd->priority)
{
  thrd->donated = true;
  thrd->priority = current_thread->priority;       // donate the priority
  /*whether we need to donate priority to lock*/
  if(another->lock_priority < current_thread->priority)
  {
    another->lock_priority = current_thread->priority;
    /*resort the struct list lock */
    list_sort(&thrd->locks, compare_lock_priority, NULL);
  }

  /*solve the donate chain problem */
  /* whether we need to donate priority again*/
  if(thrd->status == THREAD_BLOCKED && thrd->blocked != NULL)
  {
     another = thrd->blocked;
     thrd = another->holder;
  }
  else
     break;
}
```

循环条件是：拥有锁的线程的指针存在(这个锁被一个别的线程占有)，并且当前正在 cpu 中运行的线程的优先级大于那个拥有锁的线程

•判断是否需要继续进行优先级嵌套捐赠(用红色框框住的代码)
  •当线程 thrd 是被阻塞，且线程 thrd 的 blocked 不为空(表示 thrd 被锁所阻塞)
    •将阻塞线程 thrd 的锁赋值给 another
    •将锁 another 的拥有者赋给 thrd
  •如果不符合上述条件，则表明线程不需要再进行捐赠，退出循环 break;
满足上述条件进入循环（思路参考 ppt）
•设置 donated 状态：因为满足条件进入循环，说明当前线程的优先级大于占有锁的线程，当前 thrd 需要被捐赠优先级，所以 donated = true
•优先级更新：当前线程将优先级捐赠给占有所需锁的线程。
•锁记录的最大优先级的更新：判断锁的优先级是否小于当前线程的优先级，成立则进入下面
  •将当前线程的优先级赋给锁
  •调用 list_sort 函数，将线程的 locks 队列按优先级大小重新排列好，使其有序
•线程就绪队列的更新
  •因为经过上述的过程，线程的优先级发生了改变了，所以线程的就绪队列也需要更新。
  在 thread.c 中

```
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
  {
    list_sort(&ready_list, compare_thread_priority, NULL);
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
  }
}
```

  在调度时获取就绪队列中的头元素时，对就绪队列按照优先级的大小重新排序

P 操作后，申请锁成功

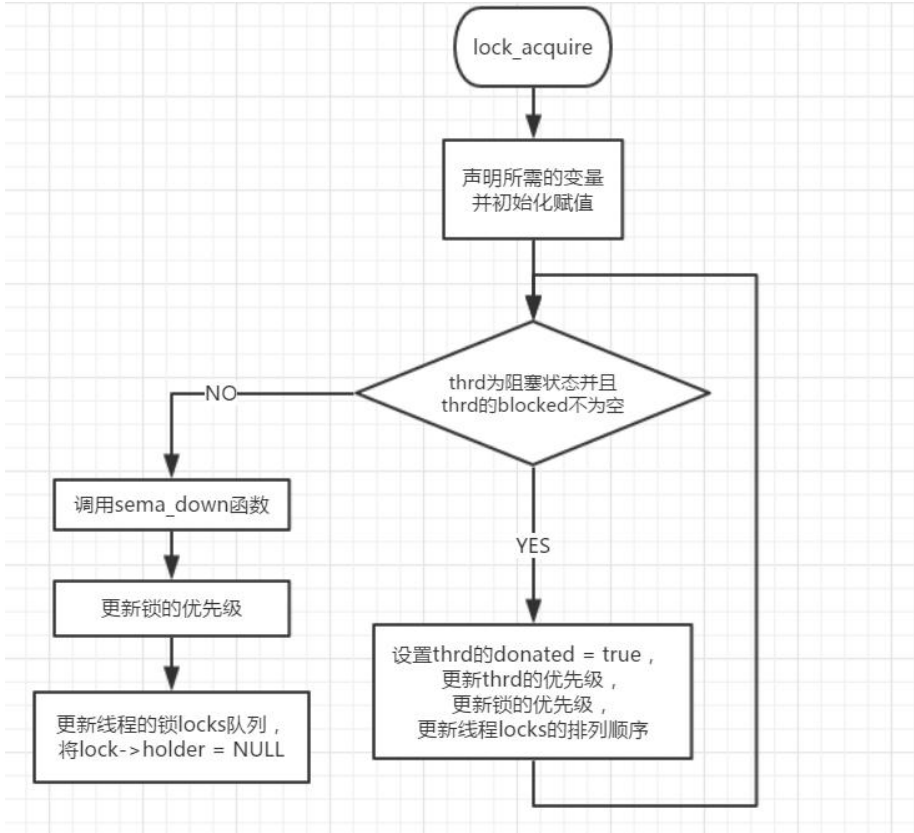
```
old_level = intr_disable ();
/*added to do the priority donation*/
if(!list_empty(&lock->semaphore.waiters))
{
  list_sort(&lock->semaphore.waiters, compare_thread_priority, NULL);
  lock->lock_priority = list_entry(list_begin(&lock->semaphore.waiters), struct thread, elem)->priority;
}
else
  lock->lock_priority = -1;
/*resort the struct list lock*/
list_insert_ordered(&current_thread->locks, &lock->holder_elem, compare_lock_priority, NULL);
lock->holder = thread_current ();
current_thread->blocked = NULL;
intr_set_level (old_level);
```

•锁的相关变量的设置：因为锁 lock 的优先级是被这个锁所阻塞的线程的最高优先级，所以当 lock 中信号量对应的等待队列不为空时，先将 lock 中信号量的等待队列按优先级顺序排好序，然后将等待队列中的第一个线程的优先级赋值给 lock，当等待队列为空时，lock 的优先级变为默认值-1；
•线程的相关变量的设置：
  •当前线程已经拥有 lock 这个锁，因此更新线程的 locks 列表,按照锁的优先级大小有序插入
  •线程获得想要的锁，不被任何锁阻塞，所以 blocked = NULL;
•红色框框住的代码，关闭了中断，因为涉及到线程和锁的信息更新， 不允许被打断。

在上面按照优先级大小排线程 locks 列表顺序时，用到了 compare_lock_priority

```
/* lock compare function */
bool
compare_lock_priority(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
  return list_entry(a, struct lock, holder_elem)->lock_priority > list_entry(b, struct lock, holder_elem)->lock_priority;
}
```

lock_acquire 函数修改后流程图：



Lock_release 函数修改

```
old_level = intr_disable();
list_remove(&lock->holder_elem);
lock->holder = NULL;
lock->lock_priority = -1;
if(list_empty(&current_thread->locks))
{
  current_thread->donated = false;
  current_thread->priority = current_thread->old_priority;
}
else
{
  l = list_front(&current_thread->locks);
  another = list_entry(l, struct lock, holder_elem);
  if(current_thread->old_priority < another->lock_priority)
  {
    current_thread->priority = another->lock_priority;
    current_thread->donated = true;
  }
  else
  {
    current_thread->priority = current_thread->old_priority;
    current_thread->donated = false;
  }
}
sema_up (&lock->semaphore);
intr_set_level (old_level);
```

•线程的相关变量设置：将该锁从线程拥有的锁列表中移除

•锁的相关变量设置

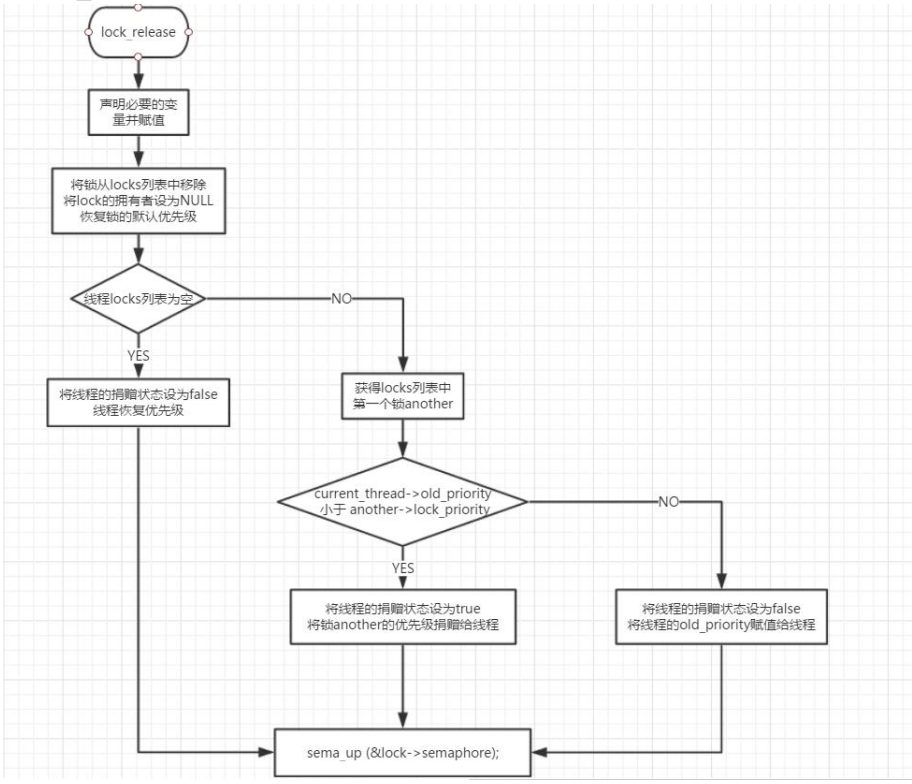
  •锁的拥有者：释放锁后，lock->holder = NULL;

  •锁的优先级：释放锁后，锁不被任何线程占有，恢复默认即优先级为-1

•处理多个锁的情况：判断当前线程是否还占有别的锁

- 锁 locks 列表为空
  - 将当前线程的捐赠状态设为 false
  - 线程进行优先级恢复
- 锁 locks 列表不为空，表明线程仍然占有别的锁
  - 比较 locks 列表中第一个锁的优先级与当前线程原来 old_priority 的大小
    - 当前线程的 old_priority < 锁的优先级：表明线程需要被捐赠
      - 将当前线程的捐赠状态设为 true；
      - 将锁的优先级捐赠给当前线程的 priority
    - 当前线程的 old_priority > 锁的优先级：线程本身优先级高，不需被捐赠
      - 将当前线程的捐赠状态设为 false；
      - 将线程原有的 old_priority 捐赠给当前线程。

Lock_release 函数修改后流程图：



sema_up 函数修改

分析 test 可知，lock_release 释放锁时，唤醒等待队列中的线程使按优先级有序唤醒的。

```
old_level = intr_disable ();

if (!list_empty (&sema->waiters))
{
  list_sort(&sema->waiters, compare_thread_priority, NULL);

 thread_unblock (list_entry (list_pop_front (&sema->waiters), struct thread, elem));
}
sema->value++;
unblock_yield();
intr_set_level (old_level);
```

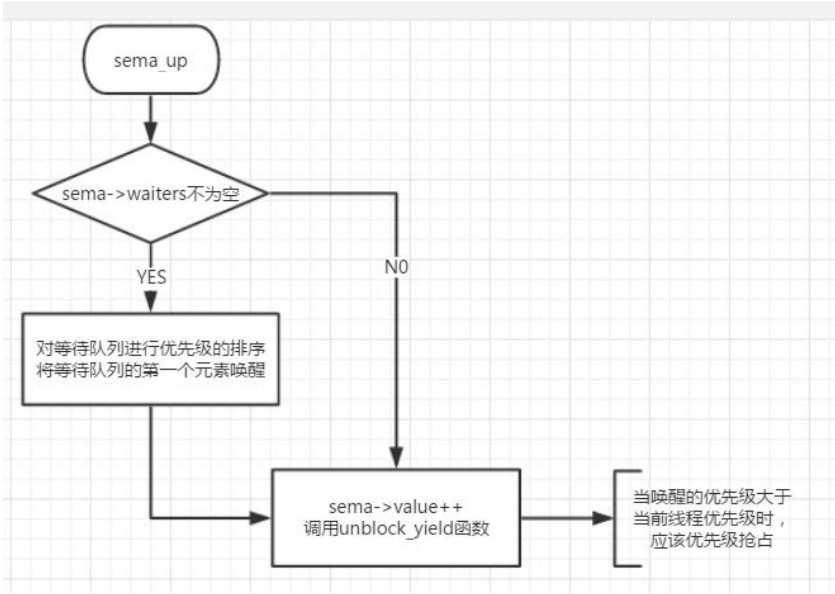
- 在调用 thread_unblock 函数唤醒线程之前，先使用 list_sort 函数将等待队列中的线程按优先级大小排好序来。
- 将线程从等待队列中唤醒出来放入就绪队列，当新唤醒的线程的优先级高于就绪队列中的线

程时，需要发生优先级抢占,调用 unblock_yield 函数

```
/* if the the priority of unblocked thread is higher than the current
 running thread, then the current running thread should yield the cpu */
void unblock_yield()
{
  if(list_begin (&ready_list) != list_end(&ready_list))
    if(thread_current()->priority < list_entry(list_begin(&ready_list), struct thread, elem)->priority)
       thread_yield();
}
```

• 当前线程优先级小于就绪队列头部线程的优先级，就调用 thread_yield 函数。

修改后 sema_up 的流程图



修改 thread_set_priority 函数

分析：当线程调用 thread_set_priority 函数设置自身优先级的时候，我们必须考虑设置的优先级是赋给 priority 还是 old_priority,为了解决问题，一开始我们就设置了变量 donated 表示线程的捐赠状态，明显如果线程不处于捐赠状态，此时只需要将优先级直接赋给两个变量 priority 和 old_priority,但是如果线程处于捐赠状态，就必须要进行判断，不要因为优先级捐赠线程有了高的优先级，然后因为设置函数，有把自己的优先级降低了。

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  if(thread_mlfqs)
    return;
  enum intr_level old_level = intr_disable();
  if(thread_current ()->donated == false)
  {
    thread_current ()->priority = thread_current ()->old_priority = new_priority;
  }
  else if(new_priority < thread_current ()->priority)
  {
    thread_current ()->old_priority = new_priority;
  }
  else
    thread_current ()->priority = thread_current ()->old_priority = new_priority;
  if(list_begin(&ready_list) != list_end(&ready_list))
    if(new_priority < list_entry(list_begin(&ready_list), struct thread, elem)->priority)
  thread_yield ();
  intr_set_level(old_level);
}
```

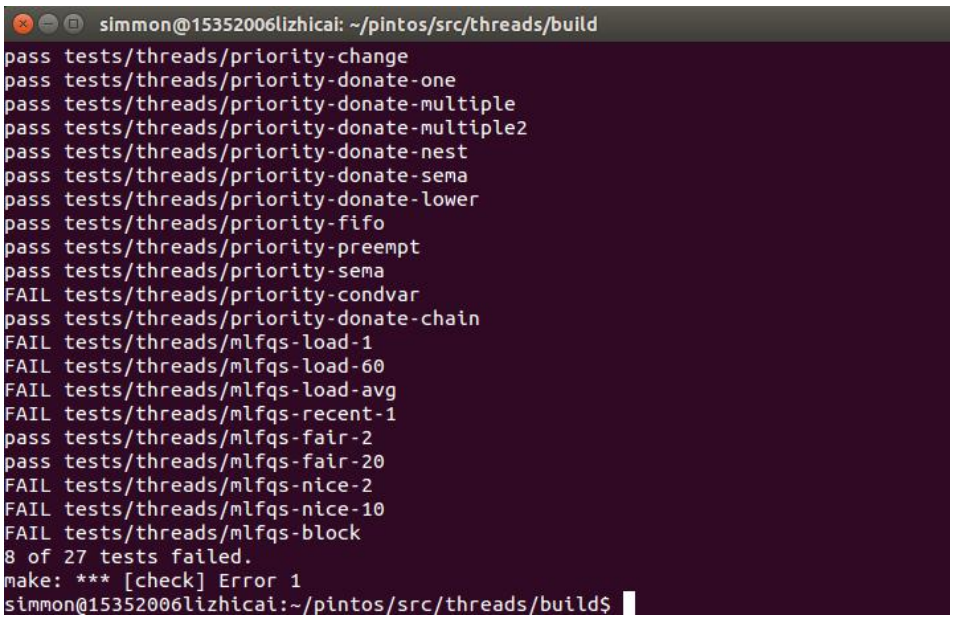
- 当线程处于非 donated 状态：将线程的 priority 和 old_priority 都设置为要设置的优先级
- 线程处于捐赠状态：
  - 线程的优先级大于要设置的优先级：为了不改变线程的捐赠状态，确保线程仍能继续执行
    - 将优先级赋给线程的 old_priority，保证线程的优先级捐赠状态
  - 线程的优先级小于要设置的优先级：

  因为线程本身即将设置的优先级就高，所以直接将线程的优先级和原来的优先级设为需要设的优先级即可。

修改后 thread_set_priority 伪代码

```
if(线程捐赠状态为 false)
do：线程的优先级 = 线程的原来本身优先级 = 新设置优先级
else if(新设置优先级 < 线程的优先级)
do：线程的原来本身优先级 = 新设置优先级
else
do: 线程的优先级 = 线程的原来本身优先级 = 新设置优先级
```

3. 实验结果

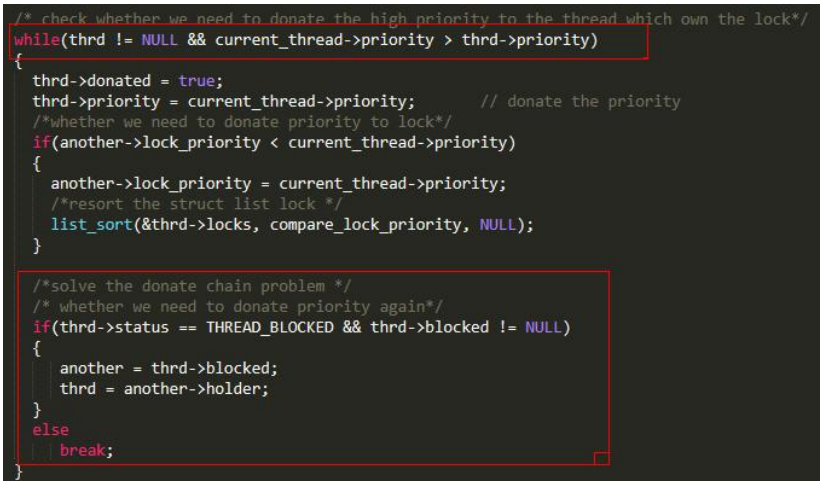


每个测试样例的运行结果，已经贴在每个 test 分析后。

4. 回答问题（15 分）

➢ 1. 如何解决嵌套捐赠的问题 ?

答：首先在线程 thread 结构体中中增添一个指针变量 struct lock *blocked,这个 blocked 指针指向的是阻塞这个线程的锁，为了实现循环嵌套问题，我们可以通过写一个 while 循环如图红色框框住的代码。



```
/* check whether we need to donate the high priority to the thread which own the lock*/
while(thrd != NULL && current_thread->priority > thrd->priority)
{
  thrd->donated = true;
  thrd->priority = current_thread->priority;       // donate the priority
  /*whether we need to donate priority to lock*/
  if(another->lock_priority < current_thread->priority)
  {
    another->lock_priority = current_thread->priority;
    /*resort the struct list lock */
    list_sort(&thrd->locks, compare_lock_priority, NULL);
  }

  /*solve the donate chain problem */
  /* whether we need to donate priority again*/
  if(thrd->status == THREAD_BLOCKED && thrd->blocked != NULL)
  {
     another = thrd->blocked;
     thrd = another->holder;
  }
  else
     break;
}
```

thrd 在这里表示的是占有锁的线程，another 是 struct lock 类型，两段红色框框住的代码是由一个 while 循环和一个判断语句构成，首先进入 while 循环，可进行优先级捐赠的条件是，thrd 不为空，即这个锁是必须是被 thrd 占有的，并且当前线程的优先级是大于锁占有者线程 thrd 的优先级。进入 while 循环后，进行优先级捐赠。随后进行判断如果 thrd 是被阻塞的，且 thrd 的 blocked 不为空，说明 thrd 被另外一个锁给阻塞住。这时候进行嵌套迭代再将阻塞 thrd 的锁赋给 another，再将锁的占有者赋给 thrd，再一次进入循环递归。这时就可以达到嵌套捐赠优先级的效果。

- 2. 如何解决一个线程占有了多个锁的问题？

答：当一个线程占有多个锁时，为了保证线程在释放一个个锁时不会造成混乱。我们可以在线程结构体中新增添一个锁列表，这样每个线程都拥有了自己的锁队列，并且在结构体锁中新增加一个变量优先级，每个锁都持有自己的优先级，这个优先级表示的是被这个锁所阻塞的线程的最高优先级。

线程获取锁的时候：①如果锁被其他线程占有，该线程被阻塞，我们可以加一个判断，判断线程的优先级是否大于锁的优先级，如果大于，就将线程的优先级赋给锁。并且由于锁的优先级发生变化，所以需要将占有这个锁的线程的 locks 列表按优先级大小重新排好序。这样每一次当不同的线程被这个锁阻塞的时候，就可以保证锁的优先级是被其自己阻塞的线程的最高优先级。②如果锁能被线程成功占有的话，则将这个锁按照优先级大小的顺序插入到线程的 locks 列表中。

线程释放锁的时候：①如果线程的 locks 列表为空，则直接将线程恢复原来的优先级即可

②如果线程的 locks 列表不为空，比较线程的优先级与线程的 locks 列表中最高优先级锁的优先级大小，如果线程的优先级小于 locks 列表的第一个元素锁的优先级，就直接将第一个锁的优先级赋给线程即可，否则将线程自身的本来优先级赋给线程优先级，因为此时线程原本的优先级大于锁的优先级，且 locks 列表里的锁是按优先级排好序的，所以第一个锁的优先级最大，此时线程原本的优先级大于锁的优先级，说明被阻塞线程的优先级小于当前线程，原本当前线程就不会被抢占，所以这种情况下，线程并不需要被捐赠。

通过上述的方式，则可解决一个线程拥有多个锁的问题。

- 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？

答：先考虑两种大方向的情况：

在线程结构体 thread 中新增添 bool 变量 donated 来表示线程是否被捐赠。

① 线程不处于捐赠状态：这种情况下进行普通的优先级设置即可,直接将修改的优先级赋值给线程的 old_priority 和 priority 即可。

② 线程处于捐赠状态：

A 修改的优先级小于捐赠的优先级：此时只将修改的优先级赋给线程的原始优先级，因为若

将修改的优先级赋值给线程当前的优先级，会破坏了线程的捐赠机制，从而使线程无法正常执行，出现错误。
B 修改的优先级大于捐赠的优先级：将修改的优先级赋值给线程当前的优先级和原始的优先级。

5. 实验感想

这一次的实验难度对我来说挺大，首先第一个问题是当锁是循环嵌套的，如何解决这种情况，一开始真的看的好懵逼。。，即使画了图，通过图像来分析，看了好久 ppt 也是理解不能，后来是通过和同学的探讨中，加上一边通过代码一边通过图形分析过程才弄懂的。第二个问题是：当线程拥有多把锁时，虽然我感觉理解起来思路比第一个问题循环嵌套容易，但是过程中出现了很多小问题，比如说忘记更新锁的优先级和拥有者 holder，在需要获取线程 locks 列表或者是等待队列中的优先级最高元素的时候，直接就取了第一个元素，没有进行排序，忽略了线程在优先级捐赠过程可能出现优先级改变的情况。第三个问题是：真个优先级捐赠过程中优先级的改变和恢复，这个过程一开始自己真的弄得好晕，不知道何时优先级该捐赠，何时优先级该恢复。后来也是通过图像把锁和线程画出来，借助图形真的有助于自己理解锁和线程的优先级变化。
还有我发现可能是测试样例太少了，忽略了一些情况时，样例依然能通过，比如说，在 lock_acquire 函数中，当改变线程优先级的时候，线程的就绪队列应该进行更新。但是我一开始没有将线程的就绪队列更新，也能 pass。
还有最后就是写报告，写报告还是很有用的，在写报告的过程发现了很多问题，也理清了很多思路，就是写的真的有点多。