Nachos Phase 1

1. 由于有一个说明说一个thread只能被join一次，所以实现非常简单: 在KThread中添加一个变量private KThread joinedBy, 然后在被join的时候记录被谁join了，并让这个thread睡觉。最后，在KThread.finish()里面（此时该线程即将结束）唤醒上述thread即可。

注: 在做题途中不断看到nachos注释中暗示join可以用一个ThreadQueue来实现。这种实现允许一个线程被多次join, 并且提供了priority donating, 非常诱人，所以在做task 5的时候实现了一下。具体方法是把joinedBy的类型变成ThreadQueue, 然后修改join方法相关部分:

if (status != statusFinished) {

if (joinedBy == null) createJoinQueue();

joinedBy.waitForAccess(currentThread);

sleep();

}

同样需要修改finish方法相关部分:

if (currentThread.joinedBy != null) {

KThread thread;

while ((thread = currentThread.joinedBy.nextThread()) != null) {

thread.ready();

}

}

（修改后的代码: KThread.join(), KThread.finish()）

测试: 这里首先fork了2个thread: join 和 join2, 分别具有优先级3和2，这两个thread将join同一个工作线程pong(本测试是在做完task 5后修改得版本，所以测试了由ThreadQueue带来的多重join功能是否正常). 带这两个thread都进入block状态后，fork另2个工作线程ping 和 pong, 分别具有优先级2和1(这样也就顺便测试了Priority Donating机制，这部分将在task 5中详细说明). 然后主线程用忙等的方法等待ping结束后再去join ping, 测试在ping结束后的join是否正常。最后，再join一下首先被fork的两个threads, 结束测试。

（测试代码: KThread.selfTest()）

测试结果:

采用RoundrobinScheduler时:

KThread.selfTest()

\*\*\* join test (#7) @ null joining pong thread (#6) @ null

\*\*\* join2 test (#8) @ null joining pong thread (#6) @ null

\*\*\* thread ping thread (#5) @ null looped 0 times

\*\*\* thread pong thread (#6) @ null looped 0 times

\*\*\* thread ping thread (#5) @ null looped 1 times

\*\*\* thread pong thread (#6) @ null looped 1 times

\*\*\* thread ping thread (#5) @ null looped 2 times

\*\*\* thread pong thread (#6) @ null looped 2 times

\*\*\* thread ping thread (#5) @ null looped 3 times

\*\*\* thread pong thread (#6) @ null looped 3 times

\*\*\* thread ping thread (#5) @ null looped 4 times

\*\*\* thread pong thread (#6) @ null looped 4 times

\*\*\* thread ping thread (#5) @ null finished

\*\*\* thread pong thread (#6) @ null finished

\*\*\* joining ping thread (#5) @ null

\*\*\* thread ping thread (#5) @ null joined

\*\*\* joining join test (#7) @ null

\*\*\* pong thread (#6) @ null joined by join test (#7) @ null

\*\*\* pong thread (#6) @ null joined by join2 test (#8) @ null

\*\*\* thread join test (#7) @ null joined

评价: join方面完成(从时序可以看出，finished总在joined之前)。

采用PriorityScheduler时:

KThread.selfTest()

\*\*\* join test (#7) @ p: 3 ep: 3 joining pong thread (#6) @ null

\*\*\* join2 test (#8) @ p: 2 ep: 2 joining pong thread (#6) @ p: 1 ep: 3

\*\*\* thread pong thread (#6) @ p: 1 ep: 3 looped 0 times

\*\*\* thread pong thread (#6) @ p: 1 ep: 3 looped 1 times

\*\*\* thread pong thread (#6) @ p: 1 ep: 3 looped 2 times

\*\*\* thread pong thread (#6) @ p: 1 ep: 3 looped 3 times

\*\*\* thread pong thread (#6) @ p: 1 ep: 3 looped 4 times

\*\*\* thread pong thread (#6) @ p: 1 ep: 3 finished

\*\*\* pong thread (#6) @ p: 1 ep: 1 joined by join test (#7) @ p: 3 ep: 3

\*\*\* thread ping thread (#5) @ p: 2 ep: 2 looped 0 times

\*\*\* pong thread (#6) @ p: 1 ep: 1 joined by join2 test (#8) @ p: 2 ep: 2

\*\*\* thread ping thread (#5) @ p: 2 ep: 2 looped 1 times

\*\*\* thread ping thread (#5) @ p: 2 ep: 2 looped 2 times

\*\*\* thread ping thread (#5) @ p: 2 ep: 2 looped 3 times

\*\*\* thread ping thread (#5) @ p: 2 ep: 2 looped 4 times

\*\*\* thread ping thread (#5) @ p: 2 ep: 2 finished

\*\*\* joining ping thread (#5) @ p: 2 ep: 2

\*\*\* thread ping thread (#5) @ p: 2 ep: 2 joined

\*\*\* joining join test (#7) @ p: 3 ep: 3

\*\*\* thread join test (#7) @ p: 3 ep: 3 joined

评价: join方面完成，理由同上。同时，原先优先级较低的pong得到join的优先级(可以从其ep的输出看到), 从而在ping之前运行。并且在joinedBy队列清空后，回到原先优先级，说明priority donating功能运作正常(但该功能重新计算优先级规则比较复杂，这个测试不完全).

2. 实现一个Condition语义，但不用Semaphore而是直接实现之。这道题只要理解Condition语义，做起来还是很简单的。

首先是sleep, 这个函数的功能是让当前thread释放掉锁，然后去睡觉，直到被唤醒。此时该thread应该重新取得这个锁随后返回。这里的实现依旧用了ThreadQueue记录等待中的线程。

public void sleep() {

Lib.assert(conditionLock.isHeldByCurrentThread());

boolean intStatus = Machine.interrupt().disable();

conditionLock.release();

waitQueue.waitForAccess(KThread.currentThread());

KThread.currentThread().sleep();

conditionLock.acquire();

Machine.interrupt().setStatus(intStatus);

}

而wake的时候只需要取得nextThread然后将其ready即可:

public void wake() {

Lib.assert(conditionLock.isHeldByCurrentThread());

boolean intStatus = Machine.interrupt().disable();

KThread thread = waitQueue.nextThread();

if (thread != null) thread.ready();

Machine.interrupt().setStatus(intStatus);

}

wakeAll可以用多次调用wake来完成，不过这样需要重复开关interrupt, 所以这里直接写了个循环:

public void wakeAll() {

Lib.assert(conditionLock.isHeldByCurrentThread());

boolean intStatus = Machine.interrupt().disable();

KThread thread = null;

while ((thread = waitQueue.nextThread()) != null)

thread.ready();

Machine.interrupt().setStatus(intStatus);

}

测试: fork几个thread, 其中每个thread每次循环中都会sleep在一个condition上，然后main thread调用wake和wakeAll, 看thread是否被唤醒。特别地，第一次wakeAll被安排在没人sleep的时候，以测试是否会出错。观察时序考察正确性。作为对比，同样的代码用Condition运行一遍，写在Condition的测试函数中。

（测试代码: Condition.selfTest()和Condition2.selfTest()）

测试结果:

这里只给出PriorityScheduler的结果:

Condition.selfTest()

\*\*\* wake all involked

\*\*\* thread pong (#11) @ p: 3 ep: 3 sleep (0)

\*\*\* thread ping (#10) @ p: 2 ep: 2 sleep (0)

\*\*\* wake all involked

\*\*\* thread pong (#11) @ p: 3 ep: 3 waked (0)

\*\*\* thread pong (#11) @ p: 3 ep: 3 sleep (1)

\*\*\* thread ping (#10) @ p: 2 ep: 2 waked (0)

\*\*\* thread ping (#10) @ p: 2 ep: 2 sleep (1)

\*\*\* wake involked

\*\*\* thread pong (#11) @ p: 3 ep: 3 waked (1)

\*\*\* thread pong (#11) @ p: 3 ep: 3 sleep (2)

\*\*\* wake involked

\*\*\* thread ping (#10) @ p: 2 ep: 2 waked (1)

\*\*\* thread ping (#10) @ p: 2 ep: 2 sleep (2)

\*\*\* wake involked

\*\*\* thread pong (#11) @ p: 3 ep: 3 waked (2)

\*\*\* thread pong (#11) @ p: 3 ep: 3 finished

\*\*\* wake involked

\*\*\* thread ping (#10) @ p: 2 ep: 2 waked (2)

\*\*\* thread ping (#10) @ p: 2 ep: 2 sleep (3)

\*\*\* wake involked

\*\*\* thread ping (#10) @ p: 2 ep: 2 waked (3)

\*\*\* thread ping (#10) @ p: 2 ep: 2 sleep (4)

\*\*\* wake involked

\*\*\* thread ping (#10) @ p: 2 ep: 2 waked (4)

\*\*\* thread ping (#10) @ p: 2 ep: 2 finished

Condition2.selfTest()

\*\*\* wake all involked

\*\*\* thread pong (#13) @ p: 3 ep: 3 sleep (0)

\*\*\* thread ping (#12) @ p: 2 ep: 2 sleep (0)

\*\*\* wake all involked

\*\*\* thread pong (#13) @ p: 3 ep: 3 waked (0)

\*\*\* thread pong (#13) @ p: 3 ep: 3 sleep (1)

\*\*\* thread ping (#12) @ p: 2 ep: 2 waked (0)

\*\*\* thread ping (#12) @ p: 2 ep: 2 sleep (1)

\*\*\* wake involked

\*\*\* thread pong (#13) @ p: 3 ep: 3 waked (1)

\*\*\* thread pong (#13) @ p: 3 ep: 3 sleep (2)

\*\*\* wake involked

\*\*\* thread pong (#13) @ p: 3 ep: 3 waked (2)

\*\*\* thread pong (#13) @ p: 3 ep: 3 finished

\*\*\* wake involked

\*\*\* thread ping (#12) @ p: 2 ep: 2 waked (1)

\*\*\* thread ping (#12) @ p: 2 ep: 2 sleep (2)

\*\*\* wake involked

\*\*\* thread ping (#12) @ p: 2 ep: 2 waked (2)

\*\*\* thread ping (#12) @ p: 2 ep: 2 sleep (3)

\*\*\* wake involked

\*\*\* thread ping (#12) @ p: 2 ep: 2 waked (3)

\*\*\* thread ping (#12) @ p: 2 ep: 2 sleep (4)

\*\*\* wake involked

\*\*\* thread ping (#12) @ p: 2 ep: 2 waked (4)

\*\*\* thread ping (#12) @ p: 2 ep: 2 finished

评价: 两个Condition的输出完全一致，另外，第一个无用的wakeAll也没有影响后面sleep的线程，同时也没有抛出异常等，也正常。

3. 实现Alarm, 这里用一个TreeMap来保存(时间, 线程)对，然后让thread去睡觉。TreeMap的插入复杂度和搜索复杂度都是O(log n), 故较采用LinkedList更优(LinkedList插入和弹出中至少有一个O(n))，而找到第一个元素采用firstKey()应该是O(1), 虽然所后取得value还是O(log n). 其次为了避免一个时间值出现2次从而第二次waitUntil导致第一次的waitUntil记录的线程被失踪，每个value都是LinkedList, 其中保存了所有在这个时间点上需要唤醒的Thread. 这个方法能保证正确性但不够轻巧。另外，由于时钟中断500个tick来一次，所以每次可能会有不止一个alarm到期，所以将弹出的代码放在while循环中，弹出到没有到期的闹钟为止。

（代码: Alarm.waitUntil和Alarm.timerInterrupt()）

测试: 第一次fork 2个线程，其中一个线程循环5次每次等待6000个tick, 另一个线程循环10次，每次等待2000个tick, 观察其出现频率和时间。第二次类似，但等待时间都不满500个tick, 分别为100和200, 观察其频率和时间。

注: 两个时钟恰好相同比较难以得到，但做过一次强制测试: 让waitUntil的参数表示等待的绝对时间，而不是当前时间基础上的差额，然后运行上述测试，结果正确。

（测试代码：Alarm.selfTest()）

测试结果:

Alarm.selfTest()

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 5270

\*\*\* thread ping (#14) @ p: 1 ep: 1 set alarm at 5280

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 7560

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 7560

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 9610

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 9610

\*\*\* thread ping (#14) @ p: 1 ep: 1 alarmed at 11650

\*\*\* thread ping (#14) @ p: 1 ep: 1 set alarm at 11650

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 11660

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 11660

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 14140

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 14140

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 16200

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 16200

\*\*\* thread ping (#14) @ p: 1 ep: 1 alarmed at 17720

\*\*\* thread ping (#14) @ p: 1 ep: 1 set alarm at 17720

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 18210

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 18210

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 20220

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 20220

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 22730

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 22730

\*\*\* thread ping (#14) @ p: 1 ep: 1 alarmed at 23750

\*\*\* thread ping (#14) @ p: 1 ep: 1 set alarm at 23750

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 24760

\*\*\* thread pong (#15) @ p: 1 ep: 1 set alarm at 24760

\*\*\* thread pong (#15) @ p: 1 ep: 1 alarmed at 27290

\*\*\* thread ping (#14) @ p: 1 ep: 1 alarmed at 29810

\*\*\* thread ping (#14) @ p: 1 ep: 1 set alarm at 29810

\*\*\* thread ping (#14) @ p: 1 ep: 1 alarmed at 35830

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 35880

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 set alarm at 35890

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 alarmed at 36330

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 set alarm at 36330

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 36340

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 36340

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 alarmed at 36820

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 set alarm at 36820

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 36830

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 36830

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 alarmed at 37300

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 set alarm at 37300

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 37310

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 37310

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 alarmed at 37810

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 set alarm at 37810

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 37820

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 37820

\*\*\* thread ping hf (#16) @ p: 1 ep: 1 alarmed at 38320

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 38330

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 38330

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 38850

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 38850

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 39340

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 39340

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 39850

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 39850

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 40350

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 set alarm at 40350

\*\*\* thread pong hf (#17) @ p: 1 ep: 1 alarmed at 40850

评价: thread的出现频率和时间都大致吻合预期，通过。注意set的时钟不满500的两个thread将在alarm到达后才能设定下一次的时间，所以相当于两个thread同时以500为单位调用waitUntil, 测试结果也说明了这一点。

4. Communicator一开始走了段不短的弯路。但最后仍然很顺利地做出来了，非常简洁。最后的实现中的想法如下:

那个lock保护着对private int word的访问, 因此无论speaker还是listener都只需要判断word内容的合法性就能知道下一步应该如何了，所以还维护了一个变量:

private boolean wordSet;

来表示word里是否有内容。剩下的逻辑就非常简单了:

·对于speaker来说，只能在!wordSet的时候写入。

·对于listener来说，只能在wordSet的时候读出。

最后，为了保证speaker说完后会等待对应的listener听完才离开，还需要添加一个condition. 这样最后的实现用了3个condition:

public void speak(int word) {

lock.acquire();

while (wordSet)

speaker.sleep();

this.word = word;

wordSet = true;

listener.wake(); // wake a listener waiting for message

synchronizer.sleep(); // wait until listener came

lock.release();

}

public int listen() {

lock.acquire();

while (!wordSet)

listener.sleep();

int word = this.word;

wordSet = false;

speaker.wake(); // wake a speaker waiting to speak

synchronizer.wake(); // wake a finished speaker

lock.release();

return word;

}

测试: 测试分3步进行。

首先先fork 3个speaker, 过一段时间后fork 3个listener看结果是否正常(包括返回时机以及返回时得到的信息是否正确).

然后先fork 3个listener, 过一段时间后fork 3个speaker看结果是否正常。

最后混合fork 3个speaker和3个listener测试。

（测试代码：Communicator.selfTest()）

测试结果:

Communicator.selfTest()

\*\*\* 1. speakers come first

\*\*\* thread speaker 1 (#18) @ p: 1 ep: 1 speaking 100

\*\*\* thread speaker 2 (#19) @ p: 1 ep: 1 speaking 200

\*\*\* thread speaker 3 (#20) @ p: 1 ep: 1 speaking 300

\*\*\* thread listener 1 (#21) @ p: 1 ep: 1 listening

\*\*\* thread listener 1 (#21) @ p: 1 ep: 1 heard 100

\*\*\* thread listener 2 (#22) @ p: 1 ep: 1 listening

\*\*\* thread listener 3 (#23) @ p: 1 ep: 1 listening

\*\*\* thread speaker 1 (#18) @ p: 1 ep: 1 spoken 100

\*\*\* thread listener 2 (#22) @ p: 1 ep: 1 heard 200

\*\*\* thread speaker 2 (#19) @ p: 1 ep: 1 spoken 200

\*\*\* thread listener 3 (#23) @ p: 1 ep: 1 heard 300

\*\*\* thread speaker 3 (#20) @ p: 1 ep: 1 spoken 300

\*\*\* 2. listeners come first

\*\*\* thread listener 1 (#24) @ p: 1 ep: 1 listening

\*\*\* thread listener 2 (#25) @ p: 1 ep: 1 listening

\*\*\* thread listener 3 (#26) @ p: 1 ep: 1 listening

\*\*\* thread speaker 2 (#28) @ p: 1 ep: 1 speaking 200

\*\*\* thread speaker 3 (#29) @ p: 1 ep: 1 speaking 300

\*\*\* thread speaker 1 (#27) @ p: 1 ep: 1 speaking 100

\*\*\* thread listener 1 (#24) @ p: 1 ep: 1 heard 200

\*\*\* thread speaker 2 (#28) @ p: 1 ep: 1 spoken 200

\*\*\* thread listener 2 (#25) @ p: 1 ep: 1 heard 300

\*\*\* thread speaker 3 (#29) @ p: 1 ep: 1 spoken 300

\*\*\* thread listener 3 (#26) @ p: 1 ep: 1 heard 100

\*\*\* thread speaker 1 (#27) @ p: 1 ep: 1 spoken 100

\*\*\* 3. mixed situation

\*\*\* thread speaker 2 (#31) @ p: 1 ep: 1 speaking 200

\*\*\* thread listener 1 (#33) @ p: 1 ep: 1 listening

\*\*\* thread listener 1 (#33) @ p: 1 ep: 1 heard 200

\*\*\* thread speaker 3 (#32) @ p: 1 ep: 1 speaking 300

\*\*\* thread listener 2 (#34) @ p: 1 ep: 1 listening

\*\*\* thread listener 2 (#34) @ p: 1 ep: 1 heard 300

\*\*\* thread listener 3 (#35) @ p: 1 ep: 1 listening

\*\*\* thread speaker 1 (#30) @ p: 1 ep: 1 speaking 100

\*\*\* thread speaker 2 (#31) @ p: 1 ep: 1 spoken 200

\*\*\* thread speaker 3 (#32) @ p: 1 ep: 1 spoken 300

\*\*\* thread listener 3 (#35) @ p: 1 ep: 1 heard 100

\*\*\* thread speaker 1 (#30) @ p: 1 ep: 1 spoken 100

评价: 从时序上(speaking -> listening -> heard -> spoken)和消息的正确性上都没有问题。

5. 这道题主要添加了priority scheduler中的代码。首先，实现优先级调度，最初用了一个LinkedList的数组，大小是priorityMaximum –priorityMinimum + 1, 每个priority各占用一个LinkedList. 但如果使用LinkedList, 当thread优先级改变的时候从队列找到它并删除将会变得非常慢。因此最终方案中在ThreadState内添加了next和prev两个field, 从而让ThreadState可以接成链表。以上做法使得一个ThreadState只能在一个ThreadQueue内，而我在文档中找到了支持这个前提的说明:

(ThreadQueue.waitForAccess()的文档) A thread must not simultaneously wait for access to multiple resources.

这样，每个ThreadQueue需要维护上述数目的ThreadState数组，而当某个thread需要从队列的内部删除的时候只需要操作prev, next, 并在必要的时候修改数组即可。这里用prev, next构造了一个循环双链表。

另外，维护一个maxGroup变量，记录当前ThreadQueue里非空的组中优先级最大的是哪个，并提供函数searchMaxPriority()来更新这个变量。如果没找到非空组，maxGroup指向priorityMinimum.

有了上述数据结构，很容易写出waitForAccess()和nextThread()等函数, 此处不赘述。下面介绍priority donating的实现。

首先，每个ThreadState中维护一个effectivePriority. 它最初被设置为与其优先级相等。然后，当priority改变的时候（通过setPriority()），首先检查是否超过当前的有效优先级，如果超过才需要调整其在队列中的位置。

其次，在需要transferPriority的队列中，记录当前资源持有者。最初的持有者是绕过waitForAccess(), 而直接调用acquire()确定的，因而在每一次nextThread()被调用时，资源持有者holder必不为空。而每一次nextThread()的调用也都将改变资源持有者。这时候，首先收回原来holder的effectivePriority, 然后将effectivePriority交给新的holder. 本队列的effectivePriority即上述maxGroup. 但是，一个thread可能会持有好几个资源，于是就需要有一个LinkedList来记录目前该线程持有哪些队列所保护的资源，而上述nextThread()中需要作的就是从这个LinkedList中删除本队列，然后要求其重新计算effectivePriority(队列的maxGroup以及线程本身的优先级中的最大值)。赋予线程优先级则是在LinkedList中添加当前队列，然后重新计算effectivePriority. 重新计算后的effectivePriority如果与原先不同，还需要调整其在队列中的位置(如果它在某队列中等待资源的话).

注意上述措施必须在transferPriority == true时才实施，否则仅实施之前的策略就足够了。

（代码: PriorityScheduler.PriorityQueue, PriorityScheduler.ThreadState）

注: 修改了PriorityScheduler.increasePriority()和decreasePriority()的默认实现。默认实现中，如果这个调用不合法(比如increase的时候已经是最高优先级了), 则会直接返回而不恢复中断状态。

测试: 首先测试优先级调度, fork 3个线程，其中2个优先级为2，另一个优先级为1, 在运行时，每个线程会在循环5次后提升自己的优先级。观察运行时序判断对错。

然后测试priority donating, 这一部分在join的测试中已经包含，此处没有更多的测试，但如上所述，在join中的测试似乎不够充分。

（测试代码: PriorityScheduler.selfTest(), KThread.selfTest()）

测试结果:

PriorityScheduler.selfTest()

\*\*\* thread ping thread (#2) @ p: 2 ep: 2 looped 0 times

\*\*\* thread ping2 thread (#3) @ p: 2 ep: 2 looped 0 times

\*\*\* thread ping thread (#2) @ p: 2 ep: 2 looped 1 times

\*\*\* thread ping2 thread (#3) @ p: 2 ep: 2 looped 1 times

\*\*\* thread ping thread (#2) @ p: 2 ep: 2 looped 2 times

\*\*\* thread ping2 thread (#3) @ p: 2 ep: 2 looped 2 times

\*\*\* thread ping thread (#2) @ p: 2 ep: 2 looped 3 times

\*\*\* thread ping2 thread (#3) @ p: 2 ep: 2 looped 3 times

\*\*\* thread ping thread (#2) @ p: 2 ep: 2 looped 4 times

\*\*\* thread ping2 thread (#3) @ p: 2 ep: 2 looped 4 times

\*\*\* thread ping thread (#2) @ p: 2 ep: 2 looped 5 times

\*\*\* thread ping2 thread (#3) @ p: 2 ep: 2 looped 5 times

\*\*\* thread ping thread (#2) @ p: 3 ep: 3 looped 6 times

\*\*\* thread ping thread (#2) @ p: 3 ep: 3 looped 7 times

\*\*\* thread ping thread (#2) @ p: 3 ep: 3 looped 8 times

\*\*\* thread ping thread (#2) @ p: 3 ep: 3 looped 9 times

\*\*\* thread ping thread (#2) @ p: 3 ep: 3 finished

\*\*\* thread ping2 thread (#3) @ p: 3 ep: 3 looped 6 times

\*\*\* thread ping2 thread (#3) @ p: 3 ep: 3 looped 7 times

\*\*\* thread ping2 thread (#3) @ p: 3 ep: 3 looped 8 times

\*\*\* thread ping2 thread (#3) @ p: 3 ep: 3 looped 9 times

\*\*\* thread ping2 thread (#3) @ p: 3 ep: 3 finished

\*\*\* thread pong thread (#4) @ p: 1 ep: 1 looped 0 times

\*\*\* thread pong thread (#4) @ p: 1 ep: 1 looped 1 times

\*\*\* thread pong thread (#4) @ p: 1 ep: 1 looped 2 times

\*\*\* thread pong thread (#4) @ p: 1 ep: 1 looped 3 times

\*\*\* thread pong thread (#4) @ p: 1 ep: 1 looped 4 times

\*\*\* thread pong thread (#4) @ p: 1 ep: 1 looped 5 times

\*\*\* thread pong thread (#4) @ p: 2 ep: 2 looped 6 times

\*\*\* thread pong thread (#4) @ p: 2 ep: 2 looped 7 times

\*\*\* thread pong thread (#4) @ p: 2 ep: 2 looped 8 times

\*\*\* thread pong thread (#4) @ p: 2 ep: 2 looped 9 times

\*\*\* thread pong thread (#4) @ p: 2 ep: 2 finished

评价: 符合优先级调度的特征。刚开始2个有相同优先级的threads交替运行，然后当其中一个率先增加自己的优先级时，另一个还没来得及变更优先级的thread就没有机会运行了，直到优先级高的thread运行结束。最后，那个优先级最低的thread运行。

6. 小船问题只要设计好每个thread的策略既可。策略简述如下:

成人的策略:

如果我在Oahu并且小船到达，

而且 船上回来的小孩告诉我Molokai还有小孩，

尝试上船，如果成功则划倒Molokai去。

小孩的策略:

如果我在Oahu并且小船到达，

而且 这里没有大人 或者 船上回来的小孩告诉我Molokai没有其他小孩了，

尝试上船，如果成功并且自己是第1个，

如果这里的小孩至少有2个(包括自己)，等其他小孩。

其他小孩到达以后划到Molokai去，然后叫醒另一个小孩。

如果自己已经是第2个，告诉那个正在等待的孩子可以开船了，自己睡觉。

如果我在Molokai并且小船到达，

尝试上船，如果成功划到Oahu并告诉其他人Molokai是否还有小孩。

从上面的策略容易看出，最后到达Molokai的总是小孩，所以小孩滑向Oahu前，有必要检视一下自己是否已经是最后的乘客了，如果是这样，到达Molokai后通知主线程传送完毕，并告诉其他人(他们都在Molokai, 所以通讯是可能的)结束线程。由于成人在划到Molokai后就没有什么动作了，所以成人线程会在到达Molokai后立刻结束。而小孩线程则需要有人通知了才会结束。

在实现中，小孩线程在一个大的while循环中，每次船到达一个地方时他们都会被船上的人唤醒(wakeAll), 然后执行自己的策略，最后再次等待船靠岸。所以在最后一次等待后他们会得知Molokai已经没有人，线程便会在此时跳出循环正常结束。

（代码: Boat）

测试: 默认代码，把后面2组测试的注释删除了。

（测试代码: Boat.selfTest()）

测试结果:

\*\*\*Testing Boats with only 2 children\*\*\*

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*\* transfer finished

\*\*\*Testing Boats with 2 children, 3 adults\*\*\*

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Adult rowing to Molokai.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Adult rowing to Molokai.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Adult rowing to Molokai.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*\* transfer finished

\*\*\*Testing Boats with 4 children, 4 adults\*\*\*

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Adult rowing to Molokai.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Adult rowing to Molokai.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Adult rowing to Molokai.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Adult rowing to Molokai.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*Child rowing to Oahu.

\*\*Child rowing to Molokai.

\*\*Child arrived on Molokai as a passenger.

\*\*\* transfer finished

评价: 结果很符合逻辑和要求。