**实验二 Alarm-Clock**

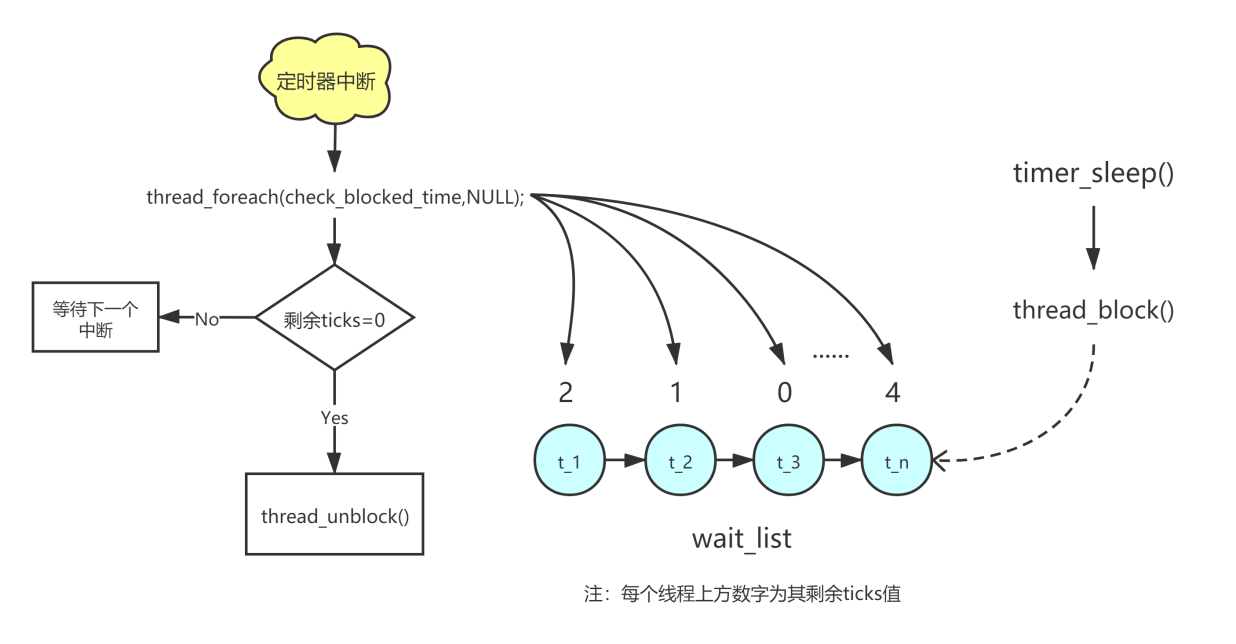
**一、实验目的**

重新实现timer\_sleep()函数，避免“忙等待”的发生。通过阅读Pintos部分源码，初步了解Pintos操作系统内核的关键函数和底层实现；通过重新设计并实现函数，加深对操作系统中线程、中断等理论知识的理解。

**二、实验方案**

为了实现让线程“睡眠”一段时间，我在timer\_sleep()函数中将原来的while循环改为了thread\_block()函数，目的是让线程暂时进入阻塞态。此后每当一次中断到来时，就遍历wait\_list中的所有线程检查它们剩余的ticks数，若ticks数已为0，则调用thread\_unblock()函数唤醒该线程，若ticks数不为0，则进行空操作，相当于继续让该线程“睡眠”。

**总体设计思路可用下图表示：**

****

**三、详细实现**

1. 修改thread.h,在struct\_thread结构体中添加变量block\_ticks\_counter，用于记录当前线程被阻塞的ticks数。

struct thread

{

/\* Owned by thread.c. \*/

tid\_t tid; /\* Thread identifier. \*/

enum thread\_status status; /\* Thread state. \*/

char name[16]; /\* Name (for debugging purposes). \*/

uint8\_t \*stack; /\* Saved stack pointer. \*/

int priority; /\* Priority. \*/

struct list\_elem allelem; /\* List element for all threads list. \*/

/\* Shared between thread.c and synch.c. \*/

struct list\_elem elem; /\* List element. \*/

int block\_ticks\_counter; /\*Define a variate to record ticks remained.\*/

#ifdef USERPROG

/\* Owned by userprog/process.c. \*/

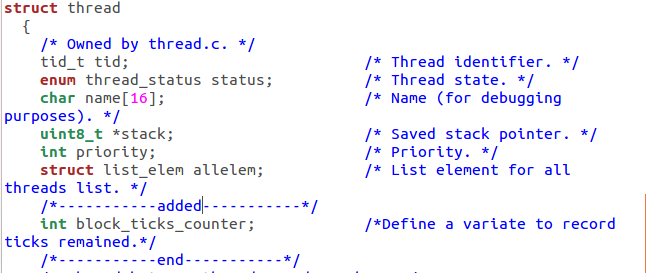
uint32\_t \*pagedir; /\* Page directory. \*/

#endif

/\* Owned by thread.c. \*/

unsigned magic; /\* Detects stack overflow. \*/

};



1. 修改thread.c中的thread\_create()函数，初始化线程被阻塞的ticks数为0（创建一个线程时，其初始化状态应为非阻塞态）。

thread\_create (const char \*name, int priority,

thread\_func \*function, void \*aux)

{

struct thread \*t;

struct kernel\_thread\_frame \*kf;

struct switch\_entry\_frame \*ef;

struct switch\_threads\_frame \*sf;

tid\_t tid;

enum intr\_level old\_level;

ASSERT (function != NULL);

/\* Allocate thread. \*/

t = palloc\_get\_page (PAL\_ZERO);

if (t == NULL)

return TID\_ERROR;

/\* Initialize thread. \*/

init\_thread (t, name, priority);

tid = t->tid = allocate\_tid ();

/\* Prepare thread for first run by initializing its stack.

Do this atomically so intermediate values for the 'stack'

member cannot be observed. \*/

old\_level = intr\_disable ();

/\* Stack frame for kernel\_thread(). \*/

kf = alloc\_frame (t, sizeof \*kf);

kf->eip = NULL;

kf->function = function;

kf->aux = aux;

/\* Stack frame for switch\_entry(). \*/

ef = alloc\_frame (t, sizeof \*ef);

ef->eip = (void (\*) (void)) kernel\_thread;

/\* Stack frame for switch\_threads(). \*/

sf = alloc\_frame (t, sizeof \*sf);

sf->eip = switch\_entry;

sf->ebp = 0;

intr\_set\_level (old\_level);

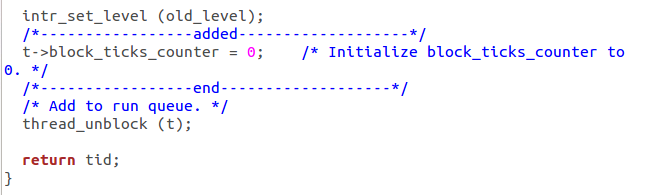
t->block\_ticks\_counter = 0; /\* Initialize block\_ticks\_counter to 0. \*/

/\* Add to run queue. \*/

thread\_unblock (t);

return tid;

}



1. 优化timer.c中的timer\_sleep()函数，不再使用原来的while循环，而是调用thread\_block()消除了原先的“忙等待”情况。

/\* Sleeps for approximately TICKS timer ticks. Interrupts must

be turned on. \*/

void

timer\_sleep (int64\_t ticks)

{

/\*int64\_t start = timer\_ticks ();

ASSERT (intr\_get\_level () == INTR\_ON);

while (timer\_elapsed (start) < ticks)

thread\_yield ();\*/

if(ticks > 0)

{

struct thread\* current\_t = thread\_current(); //获取当前线程的指针

current\_t->block\_ticks\_counter = ticks; //将等待中断的次数设置为ticks次

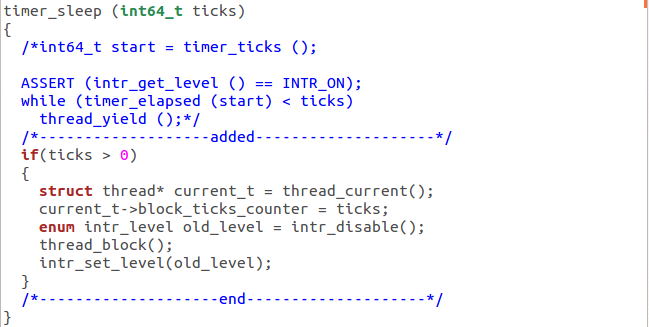
enum intr\_level old\_level = intr\_disable(); //使用thread\_block()要先关中断

thread\_block(); //阻塞该进程

intr\_set\_level(old\_level); //执行完上面的操作后恢复原来的中断状态

}

}



1. 修改thread.c，在其中加入函数check\_blocked\_time()，用于检测线程的阻塞时间并且根据阻塞时间进行相应的操作。若线程t处于阻塞状态并且其阻塞时间尚未为0，则将其剩余的阻塞ticks数自减1，此时若剩余的阻塞ticks数为0，则说明该线程的阻塞时长已到，所以我们应将其唤醒并插入到ready\_list中。

void check\_blocked\_time(struct thread \*t,void \*aux UNUSED)

{

//若线程t处于阻塞状态并且其阻塞时间尚未为0

if(t->status==THREAD\_BLOCKED && t->block\_ticks\_counter>0)

{

//将其剩余的阻塞ticks数自减1

t->block\_ticks\_counter--;

//若剩余的阻塞ticks数为0

if(t->block\_ticks\_counter==0)

{

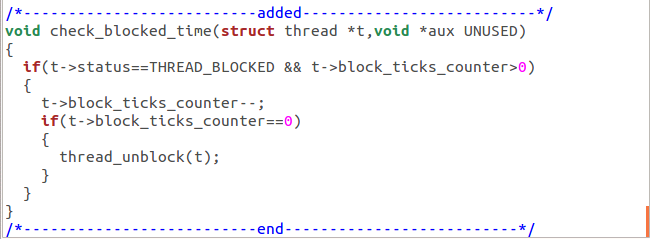
//唤醒该线程

thread\_unblock(t);

}

}

}



1. 修改函数timer\_interrupt()，实现功能：遍历整个线程链表，对每个线程使用check\_blocked\_time()函数进行剩余阻塞ticks数的检测及根据检测结果进行相应操作。

/\* Timer interrupt handler. \*/

static void

timer\_interrupt (struct intr\_frame \*args UNUSED)

{

ticks++;

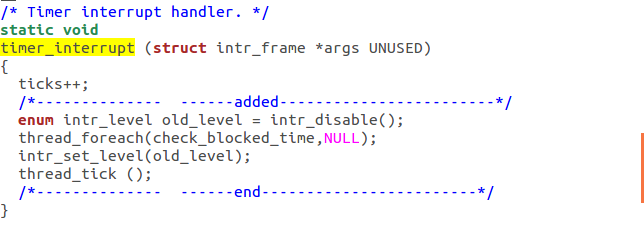
enum intr\_level old\_level = intr\_disable(); //记录原来的中断状态

thread\_foreach(check\_blocked\_time,NULL); //使用thread\_foreach()要先关中断

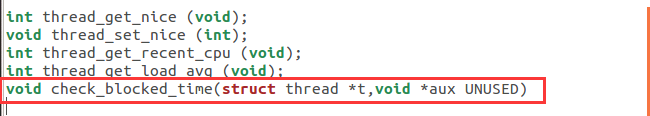
intr\_set\_level(old\_level); //恢复原来的中断状态

thread\_tick ();

}

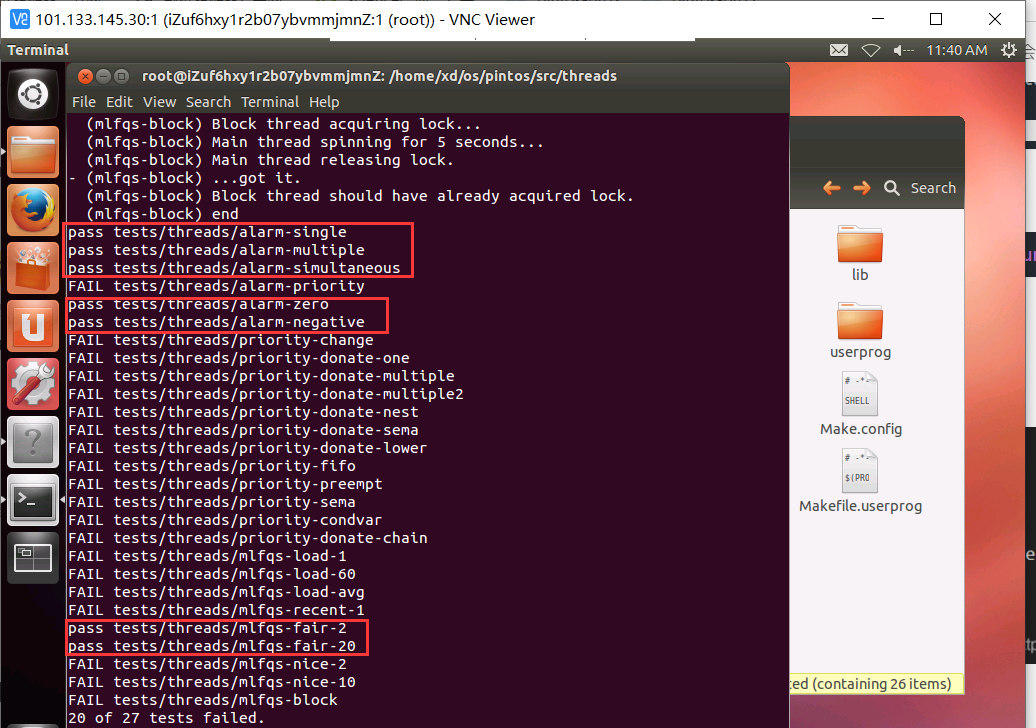


1. 最后在thread.h中添加check\_blocked\_time()函数声明。



**四、实验结果**

完成代码优化后，进入/home/xd/os/pintos/src/threads/目录，运行make check命令进行自动检查，最终结果是7成功，20失败，符合预期。



**五、心得体会**

通过本次实验，我初步接触到了Pintos操作系统内核的底层代码实现，通过阅读源码和老师写的实验指导文档，我对于Pintos实现线程的基本阻塞和唤醒以及配合中断进行线程调度的概念有了一定的理解。在本次实验中，我重新实现了timer\_sleep()函数，避免“忙等待”的发生，提高了系统CPU的运行效率。在实验中我在make check后遇到了“Old-style parameter declarations in prototyped function definition.”的问题，经过仔细排查发现是thread.h和thread.c中在对函数check\_blocked\_time()进行声明时没加分号，修改完后在make check即可正常编译执行出结果。

**实验三 Priority Scheduling**

**一、实验目的**

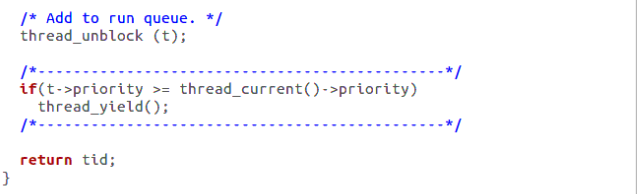
原始Pintos系统中对于线程的调度，没有考虑优先级问题，采用的是最为简单的FCFS策略。本实验要求为Pintos建立优先级调度机制，并确保任何时刻CPU上运行的都是最高优先级线程。

**二、实验方案**

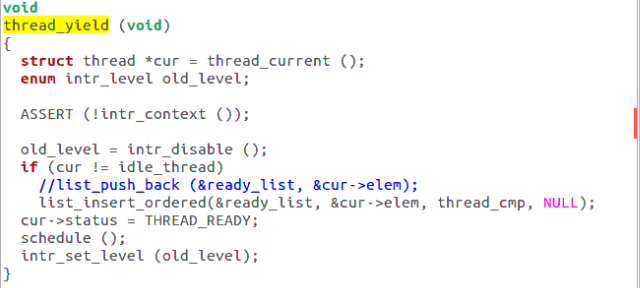
本实验的要求是为Pintos操作系统建立优先级调度机制，因此我们首先要考虑清楚在哪些情况下会发生线程的优先级调度，再考虑对于这些情况分别应采用怎样的调度策略。

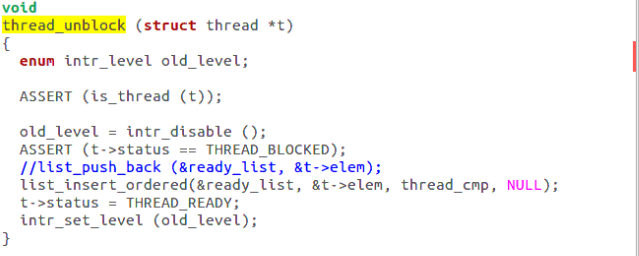
1. 新线程被建立后，若该线程的优先级高于正在占据CPU的线程t，则t应让出CPU。对于该情况，处理方式如下：在thread\_create()函数中比较新线程与线程t的优先级，若线程t优先级较低，则调用thread\_yield()将线程t加入ready-list中。为了实现ready-list中的线程按优先级降序排列，需修改thread\_yield()函数，在其中调用list\_insert\_ordered()函数。同理，对于thread\_unblock()函数，也需进行同样的修改，使被唤醒线程在插入ready-list时仍能以优先级降序排列。
2. 若一个正在运行的线程t的优先级被突然降低，以至于比ready-list中的某些线程优先级还低，则此时t必须让出CPU。对于该情况，处理方式如下：修改thread\_set\_priority()函数，在修改完t的优先级后，将该新优先级与ready\_list队首线程的优先级进行比较，若新优先级低于队首线程优先级，则调用thread\_yield()函数将线程t加入到ready-list中。
3. 如果有多个线程因为等待一把锁、一个信号量或者一个条件变量而被阻塞，当条件满足时，拥有最高优先级的线程应被优先唤醒进入ready-list。当其进入ready-list时，会引发1中情况发生。**首先**，我们应修改sema\_up()函数，每次执行V操作唤醒一个线程后将其按优先级顺序插入到ready-list中，再比较ready-list队首线程和当前正在运行线程t的优先级，若t的优先级低，则应让出CPU。**其次**，对于P操作对应的sema\_down()函数，每次阻塞线程后应将其按照优先级顺序有序插入该信号量的wait-list中。**最后**，由于一个条件变量可能正在同时被多个线程等待，所以应将条件变量的wait-list中的线程也按优先级降序排列，当条件变量满足运行要求后，wait-list中优先级最高的线程应该优先获得锁。

**三、详细实现**

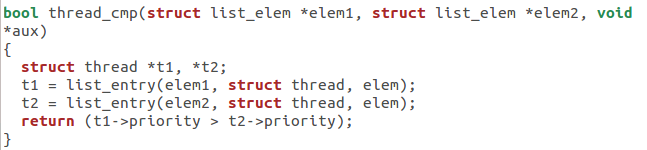
1. 修改thread.c中的thread\_create()函数，在结尾处加上比较语句，若该新建的线程的优先级高于正在执行的线程的优先级，则用thread\_yield()函数把正在运行的线程送入ready\_list。

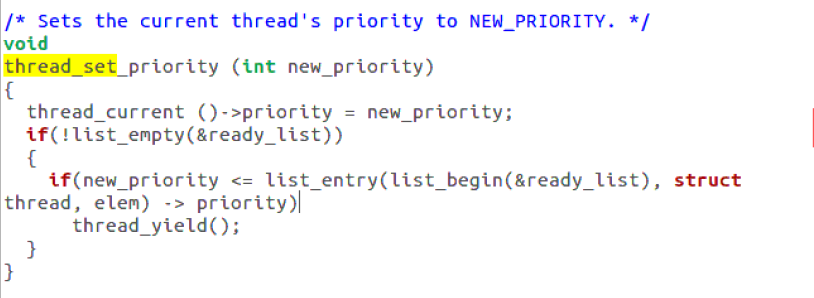
2. 修改thread\_yield()函数，使其具有将ready\_list中的线程按优先级降序排列的功能。

3. 修改thread\_unblock()函数，同样地，使其具有将ready\_list中的线程按优先级降序排列的功能。

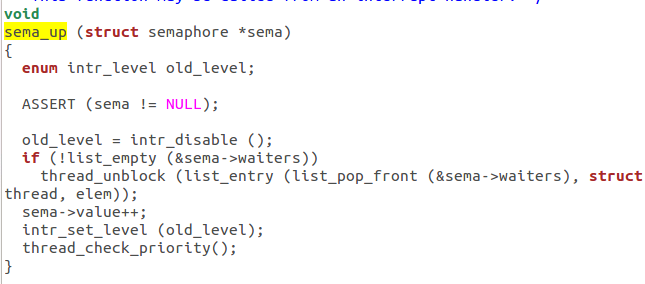


4. 在thread.c中增加优先级比较函数thread\_cmp()，以便被插入排序函数list\_insert\_ordered()作为参数调用。此外，还需在thread.h中声明该函数。

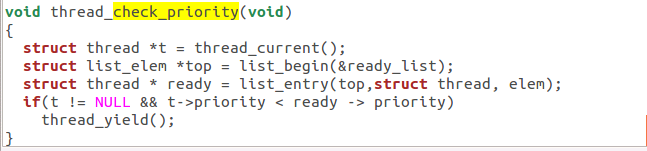
5. 修改thread\_set\_priority()函数，使其在为当前运行的线程t赋新优先级后，比较新优先级与ready\_list中队首线程的优先级，若t的新优先级低于队首线程优先级，则用thread\_yield()将t送入ready\_list中。

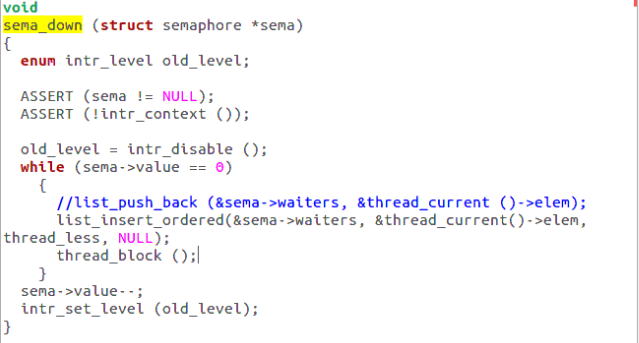
说明：第二个if语句中的list\_entry()函数用于获取ready\_list列表的头线程对应的结构体。

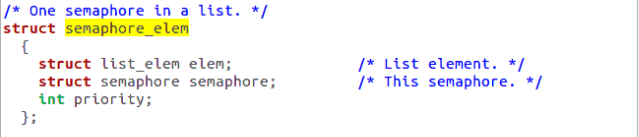
6. 在synch.c中修改sema\_up()函数，在末尾添加thread\_priority\_check()函数，使其在执行V操作唤醒一个线程后，将当前运行的线程的优先级和ready\_list队首线程的优先级进行比较。

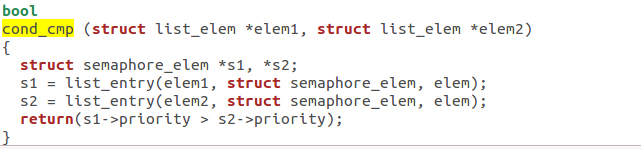


7. 在thread.c中添加thread\_check\_priority()函数以实现步骤6中提到的优先级判断功能。然后再在thread.h中对该函数进行声明。

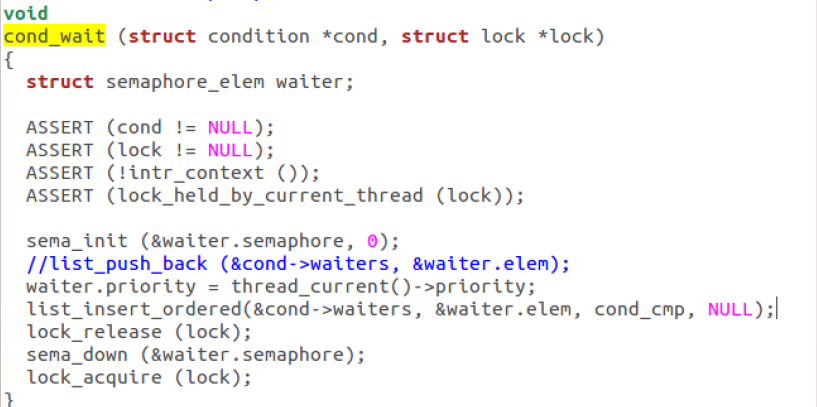
8. 在synch.c中修改sema\_down()函数，使其在执行P操作阻塞一个线程然后将该线程插入该信号量的wait\_list后，对wait\_list中的线程按优先级降序排列。

9. 在synch.c中修改结构体semaphore\_elem，添加priority属性，便于后续处理条件变量的wait\_list中的线程时需要进行优先级比较。

10. 在synch.c中定义cond\_cmp()函数，作为条件变量的wait\_list对线程进行插入排序时需要的参数。

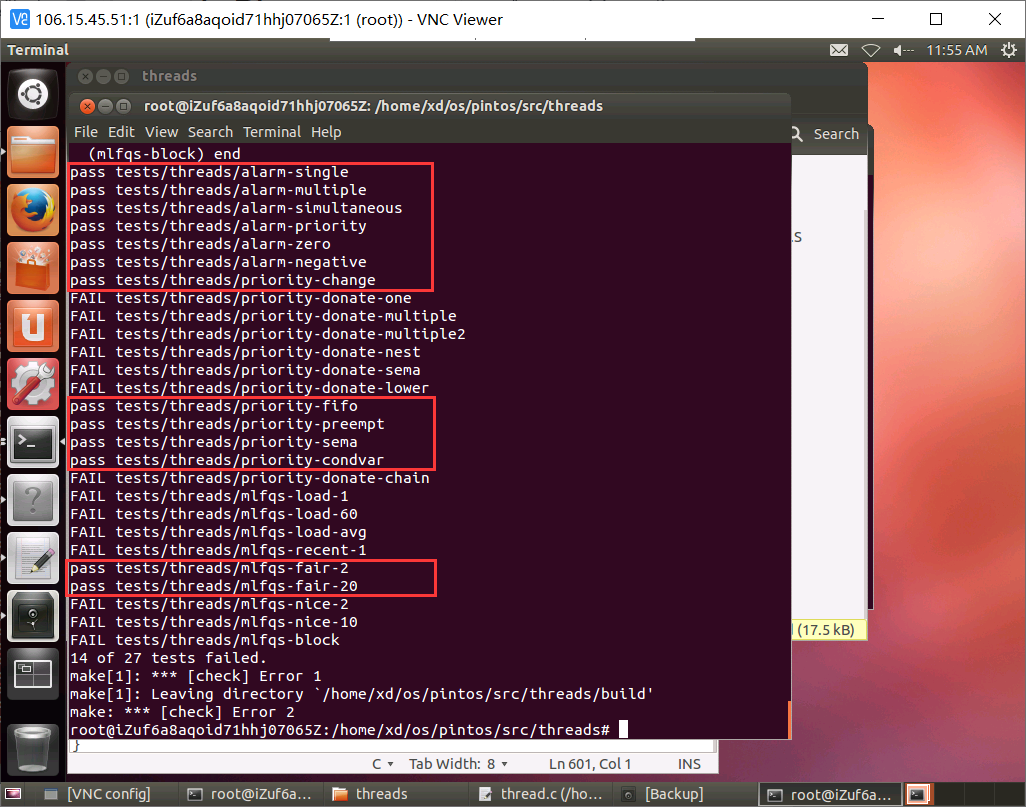


11. 修改cond\_wait()函数，利用插入排序函数对条件变量的wait\_list中的线程按照优先级降序排列。



**四、实验结果**

在/home/xd/os/pintos/src/threads/下执行make check命令，最终通过13个检测，其中有7个是实验二中通过的，剩下6个是本实验通过的，符合预期，实验成功。



**五、心得体会**

本次实验的要求是为Pintos操作系统实现线程的优先级调度功能。要完成本次实验，需要我们首先分析清楚在什么情况下需要进行线程间的优先级调度，然后针对不同的情况去修改Pintos的源码，包括在原函数末尾增加优先级判断语句和定义新函数thread\_cmp()和cond\_cmp()作为插入排序的传入参数等等。修改源代码的过程涉及了thread.c和synch.c中的多个函数，因此要耐心和细致，同时要注意区分各个变量及函数返回值的数据类型。一开始我在定义thread\_cmp()和cond\_cmp()时定义为了int型，后来make check提示该函数类型应为bool型，修改后正确。

可以说，本次实验的中心任务就是围绕优先级比较和排序进行的，主要包括正在运行线程与新线程优先级的比较，与ready\_list队首线程的比较，以及在使用thread\_yield() / thread\_unblock()，sema\_up() / sema\_down()之后应利用插入排序对ready\_list中的线程按优先级降序排列。通过本次实验，加深了我对操作系统实现线程优先级调度这一过程的理解，同时对Pintos的底层实现原理也有了进一步的认识和掌握。

**实验四 Priority Scheduling—Donation for Locks**

**一、实验目的**

解决由lock造成的优先级反转问题。

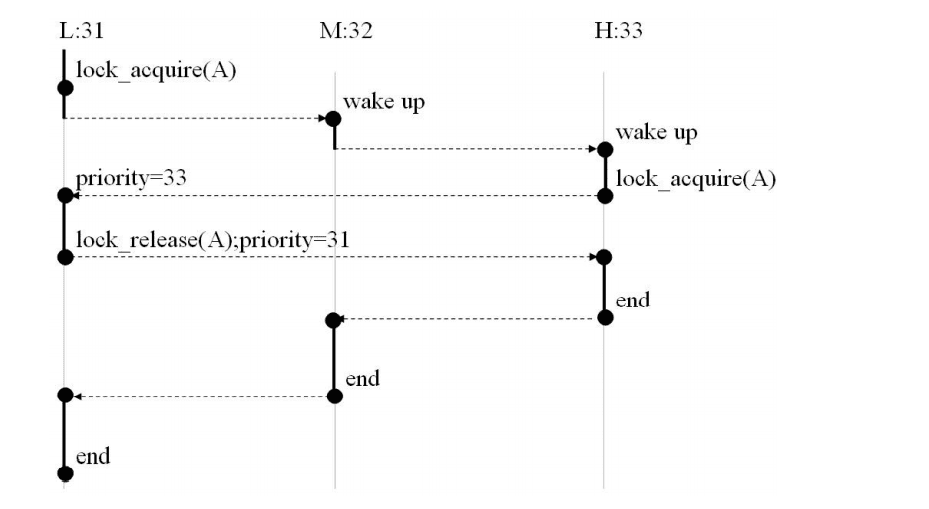
**二、实验方案**

使用优先级捐赠（Priority Donation）来解决由lock造成的优先级反转问题。

优先级捐赠是指高优先级的线程将自身优先级捐给与其竞争资源的低优先级线程，以促使其尽快释放资源，进而保证自身能尽快得到运行所需资源的策略。

优先级捐赠主要分以下几种情况：

1. **简单捐赠**，这是最为平常和简单的一种捐赠情况。lock、优先级流动及线程执行时间顺序如下图所示：



考虑线程H、M、L，优先级H > M > L，以时间为序各线程的执行情况如下：

Step1: H、M等待事件发生，处于挂起状态；L运行，持有锁A；

Step2: M等待的时间发生，就绪，抢占 CPU；

Step3: H等待的时间发生，就绪，抢占 CPU，并请求锁 A；

Step4: 由于A已被L获得，H挂起，并将优先级捐赠给 L；

Step5: L的优先级提升，开始运行，并释放锁A，同时恢复原始优先级；

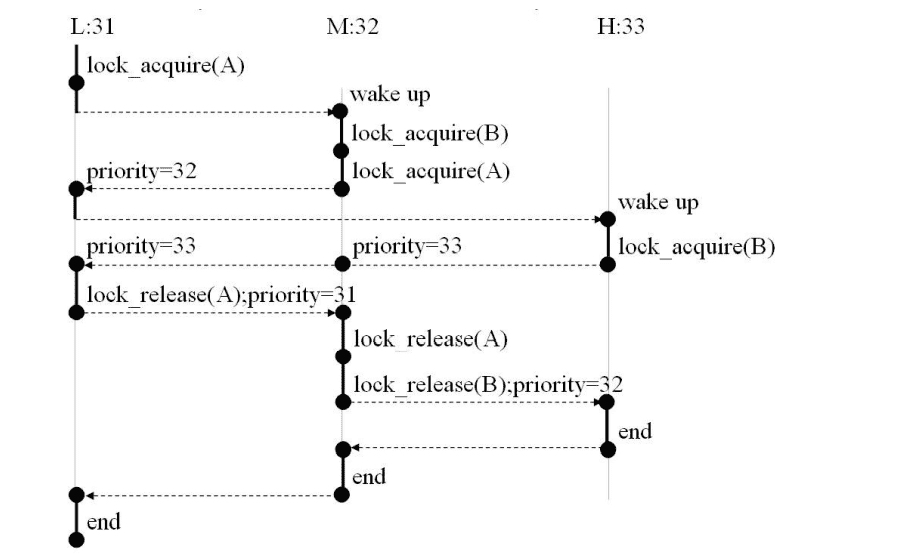
Step6: H被唤醒，运行；

Step7: M运行；

Step8: L运行。

这样可以保证最终线程的执行顺序为H->M->L。

1. **递归捐赠**，这是一种较为复杂的情况。lock、优先级流动及线程执行时间顺序如下图所示：



设有线程H、M、L，优先级H > M > L，以时间为序各线程的执行情况如下：

Step1: H、M等待事件发生，处于挂起状态；L运行，持有锁A；

Step2: M等待的事件到达，就绪，抢占CPU，请求锁B成功，并继续请求锁A，由于A已被L持有，M挂起，并捐赠自己优先级32给L；

Step3: L优先级获得提升，开始执行；

Step4: H等待的事件到达，就绪，抢占CPU，并请求锁B，由于B被M持有，H挂起，并捐赠自己优先级33给M，由于M因L而被阻塞，故优先级33也捐赠给L；

Step5: L优先级再次提升，开始运行；

Step6: L释放锁A，M被唤醒，L的优先级恢复到31；

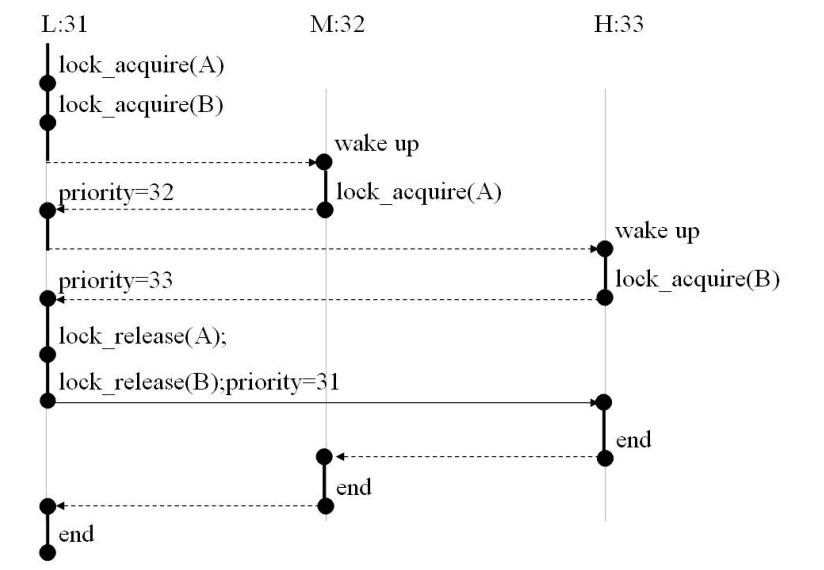
Step7: M开始运行，释放锁B，H被唤醒，M的优先级恢复到32；

Step8: H运行；

Step9: M运行；

Step10: L运行。

1. **多重捐赠**，这种情况也较为复杂。lock、优先级流动及线程执行时间顺序如下图所示：



设有线程 H、M、L，优先级 H > M > L：

Step1: H、M等待事件发生，处于挂起状态；L运行，先后持有锁A、B；

Step2: M等待的事件发生，就绪，抢占CPU，并请求锁A；

Step3: 由于A被L占有，故M挂起，并将优先级32捐赠给L；

Step4: L运行；

Step5: H等待的事件发生，就绪，抢占CPU，并请求锁B；

Step6: 由于B已被L占有，H挂起，并捐献优先级33给L；

Step7: L优先级提升，运行，先后释放A、B，L优先级恢复到31；

Step8: H被唤醒，运行；

Step9: M运行；

Step10: L运行。

**解决思路：**

1. 对于简单捐赠的情况，当高优先级线程H因为低优先级线程L占用lock而被阻塞时，H应将自己的优先级暂时捐赠给L，以让L先占据CPU运行。当L执行完毕释放该lock后，恢复L的优先级为L的原始优先级，再将lock归还给较高优先级线程H。此后H获得资源可以继续运行。
2. 对于递归捐赠的情况，通过检测被捐赠的线程是否已经获得了所需要的全部锁来判断是否出现嵌套捐赠的情况，如是则设置好参数来进行下一轮的优先级捐赠。和情况一差不多，也就是多捐赠一次，知道被捐赠线程获得了所需要的全部的锁。
3. 对于多重捐赠的情况，struct thread中的list locks，还有struct locks中的lock\_priority，两者配合使用应对multiple-donate的情况。由于每次donate的时候都是因为优先级高的一个进程需要申请一个握在优先级比较低的线程手中的，因此锁在涉及到priority-donate的时候维护一个lock\_priority，记录获得这个锁的线程此时的优先级，因为存在multiple-donate，线程可能会接受几个不同的优先级，因此需要在锁中，而不是在线程的结构中维护这样一个信息，以在释放锁的时候能够将线程优先级恢复到正确的值。

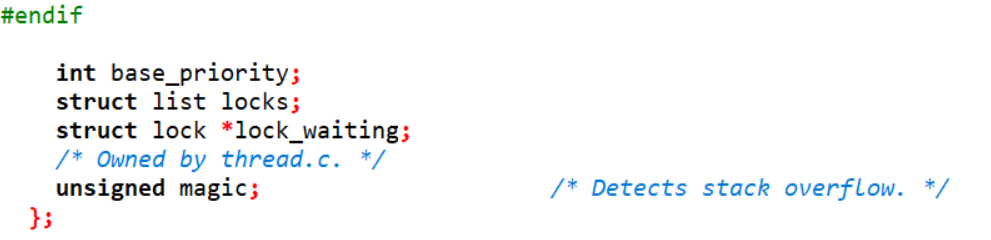
**在阅读过Pintos对优先级捐赠的测试点源码后，可总结出测试要点如下：**

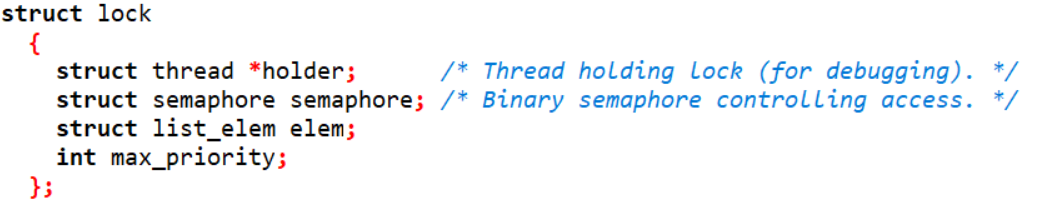
1. 在一个线程获取一个锁的时候，如果拥有这个锁的线程优先级比自己低就提高它的优先级，并且如果这个锁还被别的锁锁着，将会递归地捐赠优先级，然后在这个线程释放掉这个锁之后恢复未捐赠逻辑下的优先级。
2. 如果一个线程被多个线程捐赠，维持当前优先级为捐赠优先级中的最大值（acquire和release之时）。
3. 在对一个线程进行优先级设置的时候，如果这个线程处于被捐赠状态，则对original\_priority进行设置，然后如果设置的优先级大于当前优先级，则改变当前优先级，否则在捐赠状态取消的时候恢复original\_priority。
4. 在释放锁对一个锁优先级有改变的时候应考虑其余被捐赠优先级和当前优先级。
5. 将信号量的等待队列实现为优先级队列。
6. 将condition的waiters队列实现为优先级队列。
7. 释放锁的时候若优先级改变则可以发生抢占。

**三、详细实现**

**1. 修改数据结构**

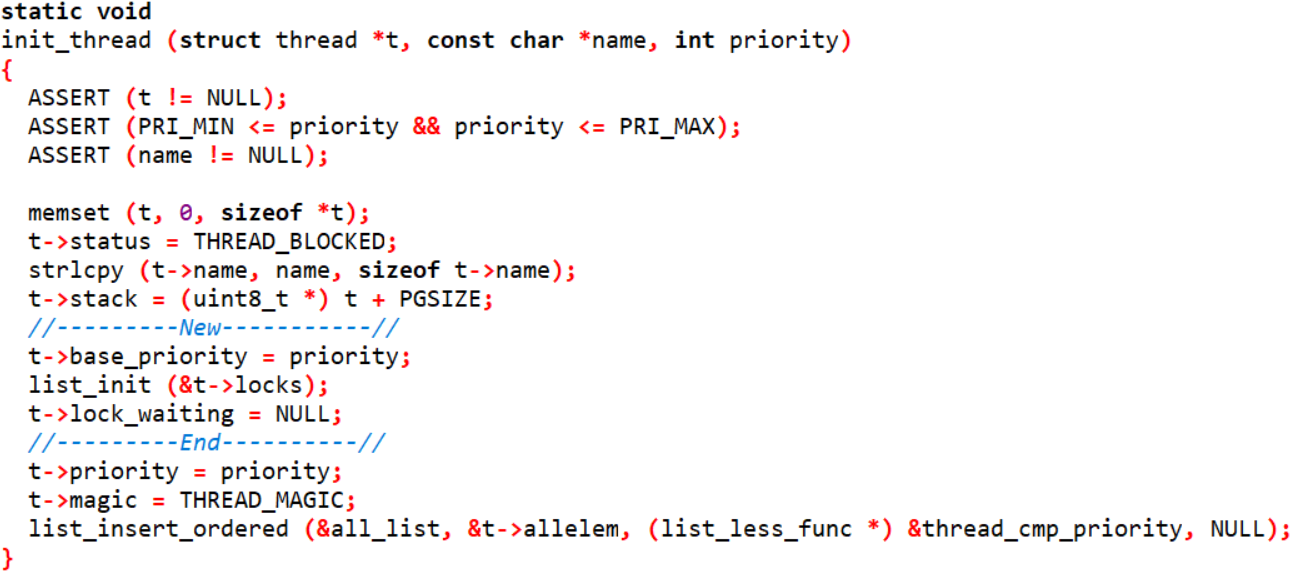
(1) 修改thread结构体，新增三个量，一个base\_priority记录线程的初始优先级，一个locks记录线程当前拥有的锁（多个），一个lock\_waiting用来记录线程等待被释放的锁。

(2) 修改lock结构体，新增两个量，一个elem记录优先级捐赠的列表元素，一个max\_priority记录请求锁的所有线程的最大优先级。

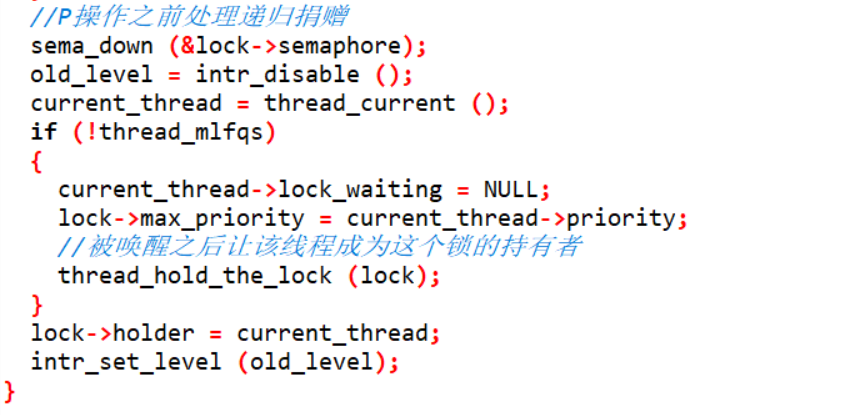
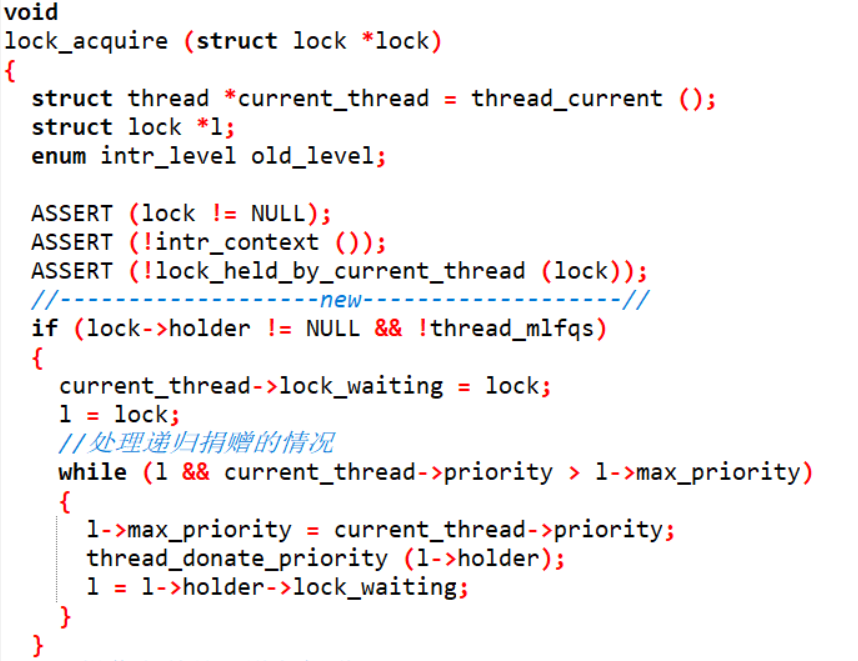


**2. 修改函数**

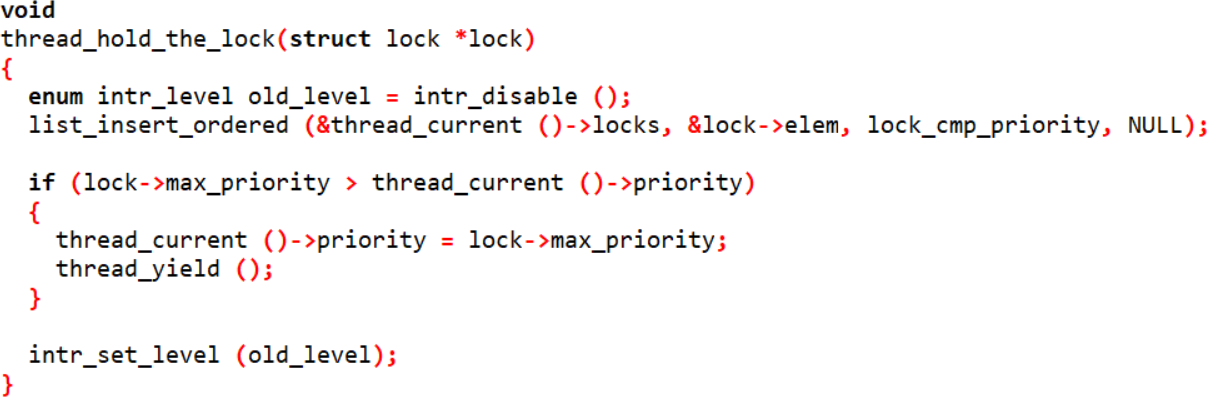
(1) 修改init\_thread()函数，实现对thread结构体中新增变量base\_priority和lock\_waiting的初始化。



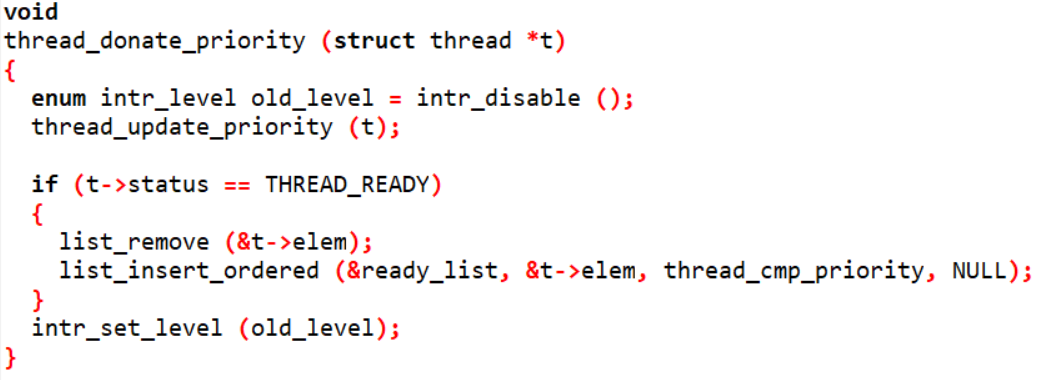
(2) 修改lock\_acquire()函数，在获取锁之前循环更新所有参与嵌套的线程的优先级，即在P操作之前解决递归捐赠的情况。然后在线程被唤醒之后让该线程持有该锁。



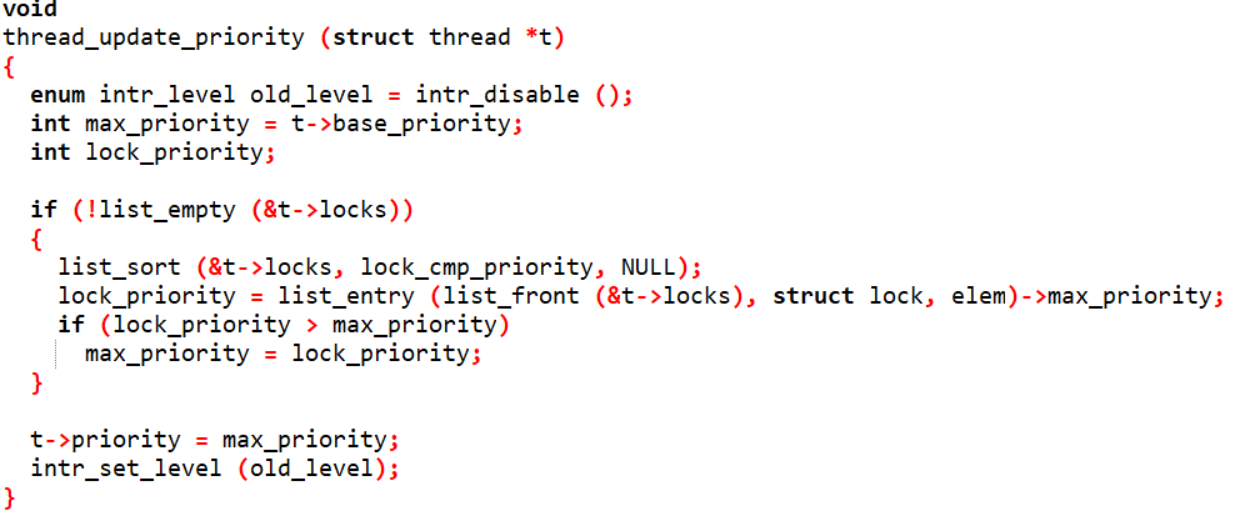
(3) 定义函数thread\_hold\_the\_lock实现将一个锁资源赋给线程。

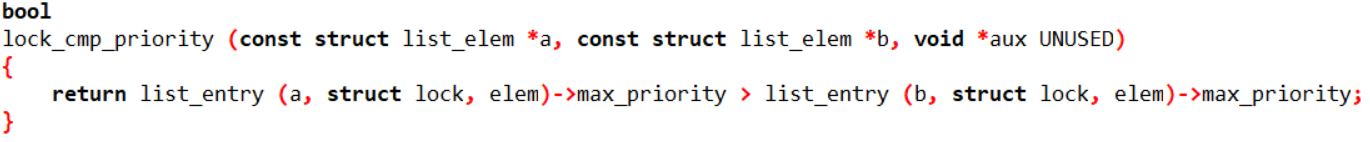


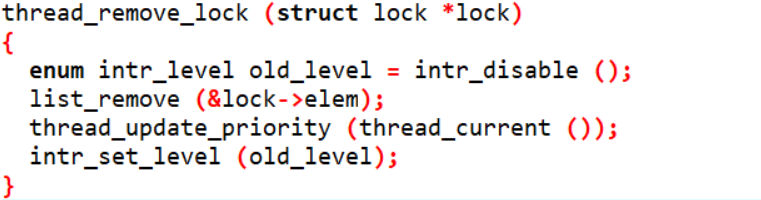
(4) 定义函数thread\_donate\_priority实现较高优先级线程将优先级捐赠给某一持有锁的较低优先级的线程。这里的优先级捐赠是通过修改锁的最高优先级， 然后调用thread\_update\_priority ()更新被捐赠线程的当前优先级实现的。



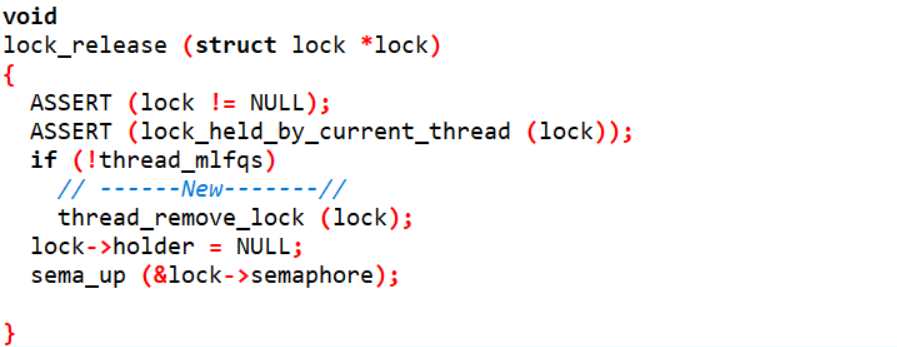
(5) 定义函数thread\_update\_priority ()实现线程优先级的更新。该函数一方面被thread\_donate\_priority()函数调用以实现优先级捐赠操作，另一方面考虑到当某线程释放锁之后，该线程的优先级可能发生变化，可用thread\_update\_priority()函数来处理这个逻辑。

(6) 定义函数lock\_cmp\_priority()实现锁队列的排序。

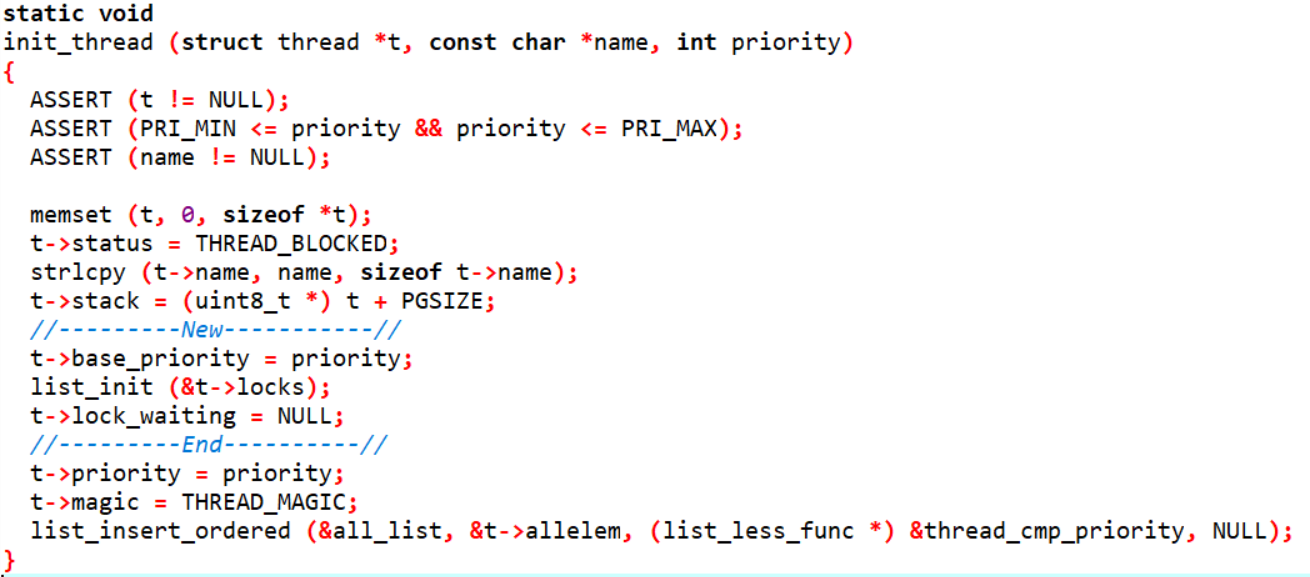
(7) 定义函数thread\_remove\_lock()实现线程释放锁时进行的操作，该函数将被synch.c中的lock\_release()函数调用。



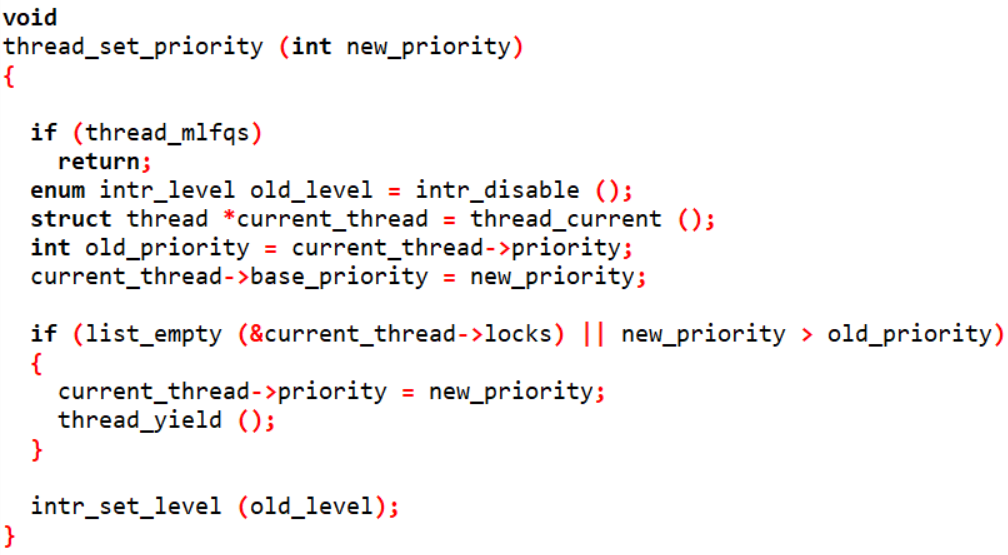
(8) 在lock\_release()函数中添加相应的语句调用(6)中的thread\_remove\_lock()函数。



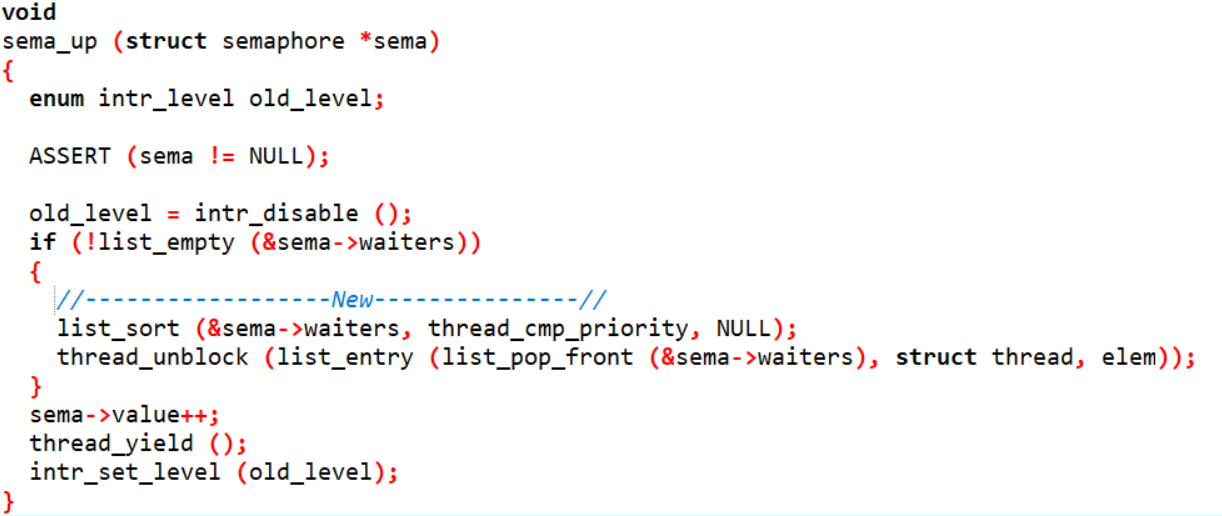
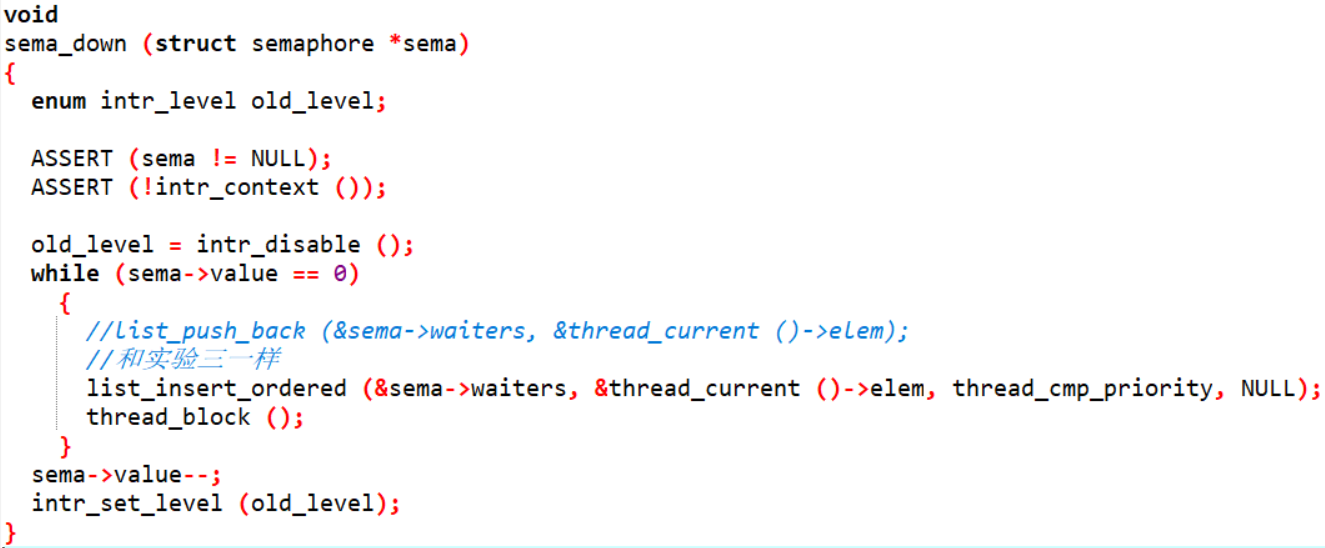
(9) 在lock\_release()函数中添加相应的语句调用(6)中的thread\_remove\_lock()函数。



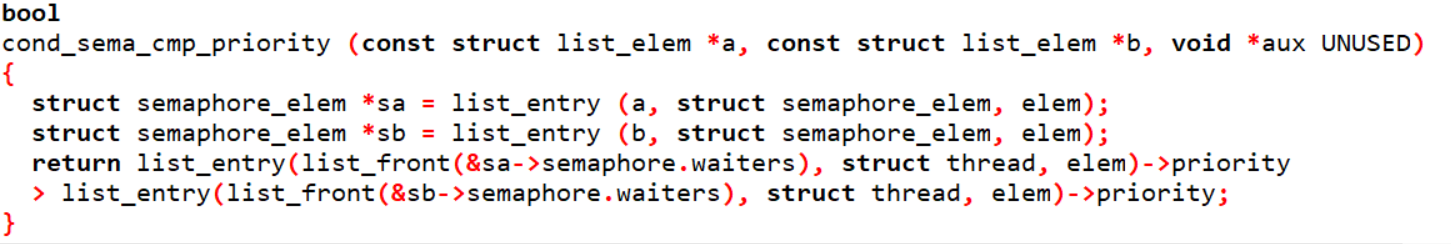
(10) 修改thread\_set\_priority()函数。如果没有锁，那么优先级捐赠的情况不用考虑，直接更新；如果更新的优先级大于当前线程的优先级，则更新当前线程优先级。总之，base\_priority一定要在该函数中得到更新。



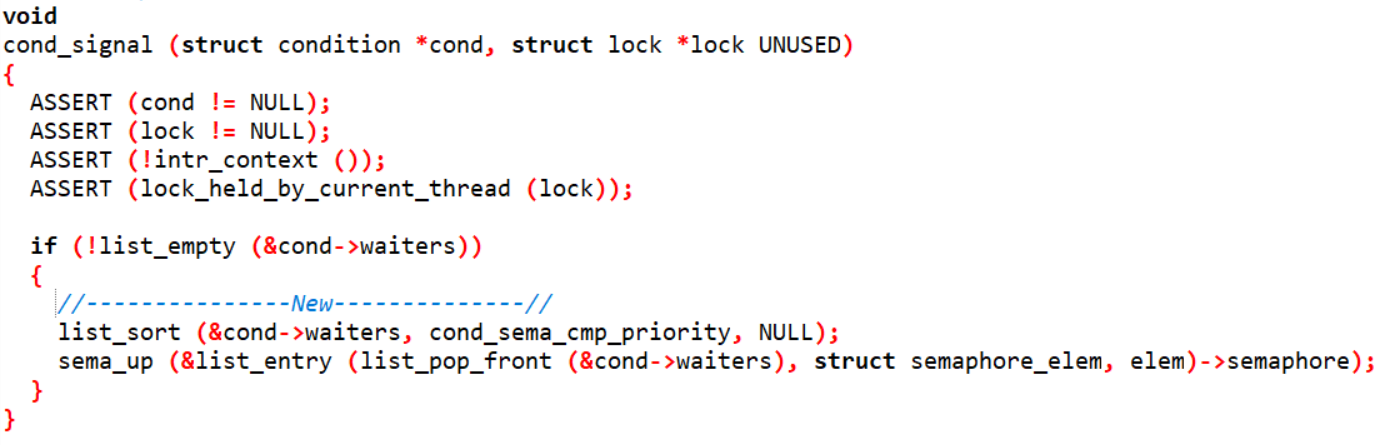
(11) 修改sema\_up()和seme\_down()函数，将信号量的wait\_list实现为优先级队列，即将wait\_list中的线程按优先级降序排列，该过程在实验三中已经完成过一遍。



(12) 定义cond\_sema\_cmp\_priority()函数实现条件变量wait\_list中线程的优先级比较，便于(12)中使用该函数作为list\_sort()排序函数的参数。

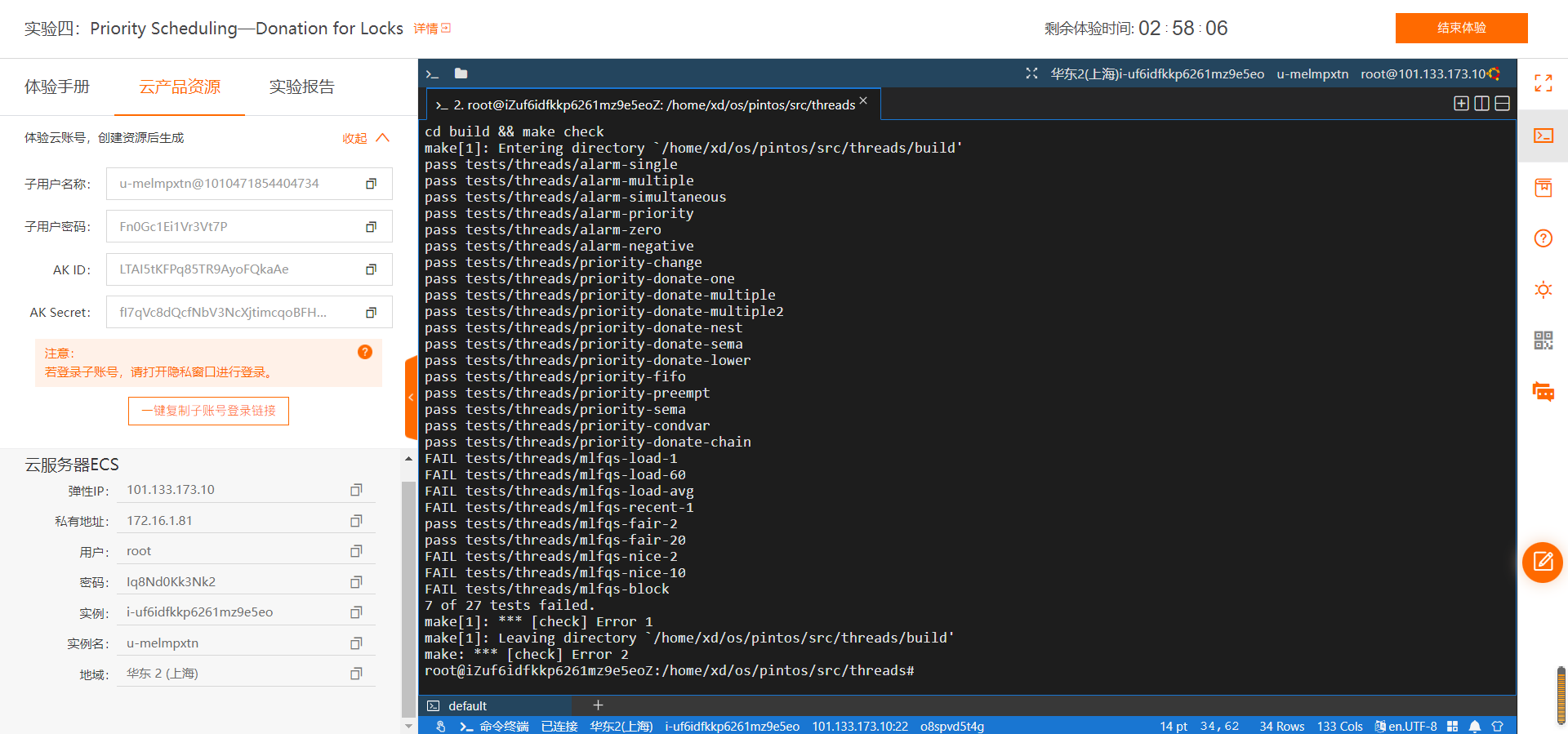
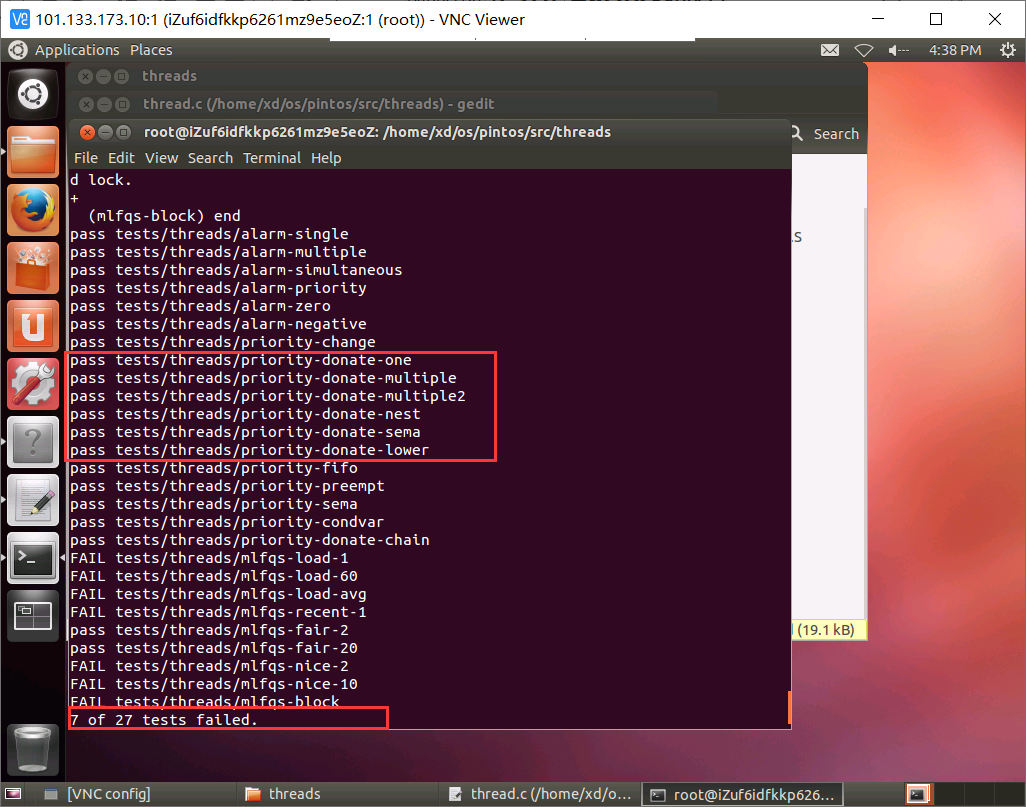


(13) 修改 cond\_signal()函数，将条件变量的wait\_list也改成优先级队列，其中list\_sort()排序函数使用(11)中的定义的比较函数作为参数。



**四、实验结果**

在/home/xd/os/pintos/src/threads/下执行make check命令，最终通过20个检测，其中有6个和priority-donate有关的测试点是本实验通过的，符合预期，实验成功。



1. **心得体会**

本次实验要求我们使用优先级捐赠（Priority Donation）来解决由lock造成的优先级反转问题。为了解决这一问题，我们首先要明白简单捐赠、嵌套捐赠和多重捐赠这三种方式实现的原理，然后再考虑每种情况该如何处理。在完成本次实验的过程中，为了加深对实验原理的理解，我查阅相关资料，阅读了Pintos有关优先级捐赠的测试代码，并由此获得了一些修改结构体与函数的启发。在具体实现时，我们还需要对Pintos的源码进行反复的阅读与体会，搞清楚各个变量尤其是指针的含义。在给函数传参时，要仔细注意各个参数的类型防止出错。在声明函数时，要注意所有函数均应在thread.h中声明，若仅在synch.h中声明而不在thread.h声明，make check后会报错。由于本次课设的代码涉及很多指针等变量的类型转换问题，所以在具体实现时也碰到了很多的问题，在与同学讨论，阅读Pintos源码、查阅相关资料后才得以实现。通过这次课设，我加深了对操作系统中线程优先级调度、优先级捐赠算法的理解，对于信号量、条件变量及其等待队列在优先级调度过程中涉及到的操作也有了更深的理解。