```
+--------------------+
|        CS 140      |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT  |
+--------------------+
```

---- Info ----

>> Fill in the names and email addresses of you.

阮明康<860797144@qq.com>

吴宏昱<hapda@qq.com>

安显琴<1244905436@qq.com>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

（1） 运行结果：

Project 1 All 27 tests pass.

```
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
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```

（2） 关于本文档的阅读

由于实际实现代码较长，我们小组在文档后面的附录中附上了代码的完整版本，而在解析每个 task 的实现的时候并没有把完整的 struct 结构体定义或者是函数定义写出来，而是只是注明了与该 task 有关的一些信息。 然而，点击相应的 struct 名或者函数名能够跳到附录的具体实现中去。

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.

参考资料：

Pintos project 1 ppt :http://wenku.baidu.com/view/719281d5c1c708a1284a44f2.html

网上流传的温帅哥系列视频

Pintos 4p group 小组的文档 pintosof4p.googlecode.com/files/project1docpre1.pdf

ALARM CLOCK

==========

---- DATA STRUCTURES ----

需要更改 struct thread，添加字段 int ticks_block(初始化为 0),记录当前线程还需要休眠的 ticks。

```
struct thread
  {
    /* Owned by thread.c. */
    /*
    一些无关的数据成员
    */
        int ticks_block;                //记录已经经历的 ticks，以确定是否需要唤醒
    /*
    一些无关的数据成员.....
    */
  }
```

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),
>> including the effects of the timer interrupt handler.

timer_sleep()函数原本的实现是使线程进入自旋状态，在自旋了指定的 ticks 之后，自旋结束并执行线程余下的指令。

优点：实现简单

缺点：浪费 CPU 时间，在线程进入 sleep 的时候实际上并没有释放 CPU，是一种伪休眠

A3.解决方式：
1）修改的函数（点击函数名跳到函数实现）
@file : /device/timer.c
timer_sleep (int64_t ticks)
timer_interrupt (struct intr_frame *args UNUSED)
添加函数：
@file: /threads/thread.c
void checkInvoke(struct thread* t,void* aux UNUSED)

2）完成方式：
```
timer_sleep(ticks){
  设置本线程的 ticks_block 为 ticks
  将本线程状态设置为 THREAD_BLOCKED
  Schedule();
}
```
被阻断的线程不再进入 ready_list，但是仍然存在于 all_list 中
在每次 timer_interrupt 时，系统遍历整个 all_list 检查是否需要将符合条件的被阻断的线程重新加入到 ready_list 中运行
```
timer_interrupt(){
        本线程 ticks 自增 1
        foreach thread a in all_list{
                检查是否需要唤醒
        }
}
```
检查线程是否需要唤醒的函数： void checkInvoke(struct thread* t,void* aux UNUSED)
```
checkInvoke(thread* a){
        if(线程 a 处于 THREAD_BLOCK 状态 且 线程 a->ticks_block 非零)
                a->ticks_block --;
        else
                unlock 线程 a
}


}
```
3）效果：
由于忙等待变为真正的休眠，提高了系统实际的并发程度和吞吐量。

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer_sleep() simultaneously?
调用 timer_sleep()的时候，会先关闭中断，因此不会有线程抢占当前 CPU，保证当前只有一个线程调用 timer_sleep 函数。即使很不幸存在这样的一种情况，也是线程安全的，因为 timer_sleep 是由线程本身调用的，且不存在共享的关键数据（例如 ticks_block 等）。另外，系统在初始化的时候（init.c 中），进入多线程之前会开启信号量和锁的机制，同一时刻只能有一个线程调用 timer_sleep 且至少一直运行至函数返回。
>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
Timer_sleep 执行过程中是在 block 掉当前线程时是关闭中断的。
---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to
>> another design you considered?
尝试过使用 block_list 来储存正在休眠的线程，这样子就不需要遍历长度更大的 all_list 就能够检查线程是否需要被唤醒了。 由于时间问题，并未改进。这是因为设计的缺陷问题导致的，会在后期进行改进。

PRIORITY SCHEDULING

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

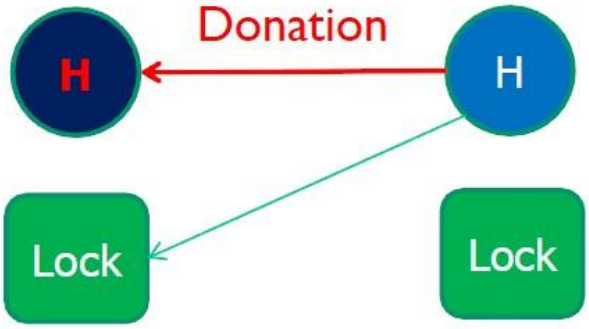
```
struct thread
  {
    /*
    .......
    一些无关的数据成员
    .......
    */
        int priority;                   /* Priority. */
        int old_priority;           //原始的优先级，和 donated 一起用
        struct list locks;          //当前线程所持有的所有的锁
        bool donated;               //表示当前的线程是否被捐赠过优先级
        struct lock* blocked;       //当前线程正在被什么锁所 block
    /*
    .......
```

```
    一些无关的数据成员

    .....
    */

  };
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;                 /* Current value. */

    struct list waiters;            /* 正在等待当前信号量的线程的链表 */

    struct list_elem holder_elem;

    int lock_priority;              /* 当前获得信号量的线程的优先级*/

  };

/* Lock. */
struct lock
  {
    struct thread *holder;          /* 当前获得锁的线程 */

    struct semaphore semaphore; /* 当前锁的信号量，一般初始化为 1 */

    struct list_elem holder_elem; /* 用于加入到某个获得当前信号量的线程的 locks 队
列中*/

    int lock_priority;              /*当前获得锁的线程的优先级*/

  };
```

>> B2: Explain the data structure used to track priority donation.
使用到
1）bool donated, int old_priority
两者配合使用。 donate 表示当前进程的优先级 priority 是原始的优先级还是 donate 得来的。 而若是 donate 得来的，那么 old_priority 存有线程最原始的优先级，old_priority 在线程创建时创建，并且在线程的生命周期中只会被非 donate 的优先级改变而改变。 Donation 中的优先级改变不会涉及到 old_priority 的改变。

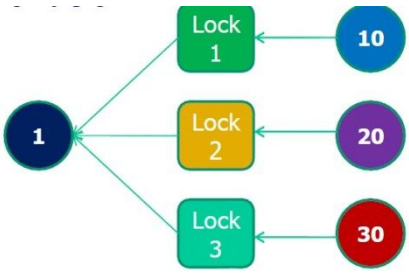
考虑下面的情况：
有三个不同优先级的进程, H, M and L

- 优先级: H > M > L
•H 和 L 需要相同的资源（例如. 某个锁）而 L 正持有该资源
- 预期的执行顺序是 L, H, M
•实际上, H 被 block 而 M 首先执行，然后是 L 和 H 执行.
- P(M) > P(L)
而 H 被阻断，所以实际执行顺序是 M L H



解决办法： priority-donation 将优先级较高的线程的优先级 donate 给持有资源的优先级的进程，让其先执行。执行完之后，在 lock_release（）函数中，若出现嵌套 donate 的情况，也能通过这两个变量递归

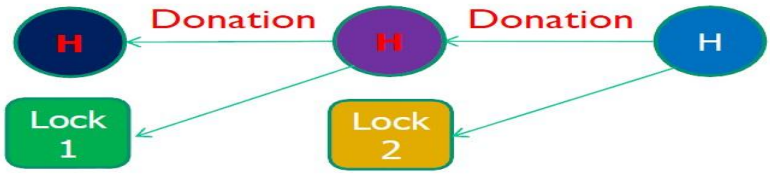
2）struct thread 中的 list locks， 还有 struct locks 中的 lock_priority



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，undonate 的时候能够将线程优先级恢复到正确的值。

>> Use ASCII art to diagram a nested donation. (Alternately, submit a
>> .png file.)

Nested-donation



---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?

每个锁对应一个信号量（初始化为 1），每个 condtion 在等待时会维护一个 叫 waiter 的信号量。每个信号量，锁都维护着 一个 lock_priority 的变量，能够记录当前信号量或者是锁的优先级，信号量维护一个 waiters 的队列， 在释

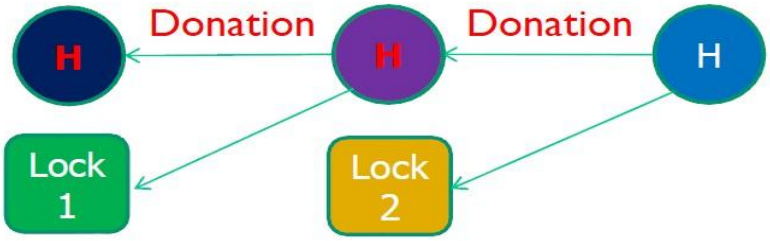
放锁，的时候，会首先按照 waiters 中的优先级对队列进行排序，然后取出其中优先级较高的执行。

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?

在 lock_acquire()函数中有一段关键代码：

伪代码为：

Assume: thrd ： 持有锁但是优先级较低的线程
curr: 正在申请使用锁的线程
Another: 当前被 thrd 持有，curr 正在申请的锁
While(正在申请的锁被优先级较低的线程 thrd 持有){
提高 thrd 的优先级，并同时更新所持有的锁 another 的优先级
若 thrd 也被某个线程 block ， 那么 thrd = block 掉 thrd 的线程
another = 使 thrd 被 block 掉的锁
}最终，使得下面的一种情况出现：



具体实现为：

```
while(!thread_mlfqs && thrd != NULL && thrd->priority < curr->priority)
  {
   thrd->donated = true;
   thread_set_priority_mlfq(thrd,curr->priority,true);
   if(another -> lock_priority < curr->priority)
   {
        another->lock_priority = curr->priority;
        list_remove(&another->holder_elem);
        list_insert_ordered(&thrd->locks,&another->holder_elem,cmp_p
riority2,NULL);
    }
   if(thrd->status == THREAD_BLOCKED && thrd->blocked != NULL)
   {
       another = thrd->blocked;
       thrd = thrd->blocked->holder;
   }
   else
       break;
```

```
  }
```

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.

分析：

在释放锁的时候，若这是进程所持有的最后一个锁，那么进程将恢复到最原始的优先级，且将等待该锁的线程通过 sema_up(lock->semaphore)唤醒。若不是，那么将处理所还持有的锁中优先级最高的那个，并将自己的优先级恢复到该锁相应的优先级，以处理 multiple-donation 的情况，lock_release()

关键部分为：

```
if(list_empty(&curr->locks))
{
//情况 1：当前是最后一个锁
          curr->donated = false;
          thread_set_priority(curr->old_priority);
  }else{
//情况 2：还持有其他锁
          l = list_front(&curr->locks);
          another = list_entry(l,struct lock,holder_elem);
          if(another->lock_priority != PRI_MIN - 1){
        //another 是当前的 curr 拿到的锁的优先级最高的一个，那个锁记录
        //线程 curr 拿到锁的时候的优先级
          thread_set_priority_mlfq(curr,another->lock_priority,false);
          }else{
     //没有锁了，直接设置为最最最初的优先级就好，就是没有 donate 那一个
                thread_set_priority(curr->old_priority);
          }
```

---- SYNCHRONIZATION ----

>> B6: Describe a potential race in thread_set_priority() and explain
>> how your implementation avoids it. Can you use a lock to avoid
>> this race?

thread_set_priority 只能由当前线程调用。可能的出现的 race condition 是两个线程同时需要将自身优先级设置为最高级，因为设置优先级可能会抢占当前线程 CPU。导致两个线程不断抢占对方的 CPU，造成死循环。

解决办法： 只有新设置的优先级比当前进程的优先级高（不能相等）时才抢占 CPU。

不能使用锁来解决这个问题，原因：
锁机制存在着 priority-donation 的情况，

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to
>> another design you considered?
我曾尝试过使用多个优先级队列，而不是一个 all_list，那样子的话就不需要对 all_list 进行频繁的排序调整了，但是这样的话实现难度较大，需要同时维护 64 个队列,还有一个 block_list.于是我选择了当前的实现。在一个队列中，并且通过线程的优先级对 list 进行排序。

ADVANCED SCHEDULER
==================

---- DATA STRUCTURES ----

>> C1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

```
struct thread
  {
    /* Owned by thread.c. */
    /*
    一些无关的数据成员
    */
    int32_t nice;                   //当前线程 nice 值
    int64_t recent_cpu;             //当前线程 recent_cpu 值
    /*
    一些无关的数据成员.....
    */
   }
```

---- ALGORITHMS ----

>> C2: Suppose threads A, B, and C have nice values 0, 1, and 2. Each
>> has a recent_cpu value of 0. Fill in the table below showing the
>> scheduling decision and the priority and recent_cpu values for each
>> thread after each given number of timer ticks:
根据我们小组的实现，优先级一样时按照 FIFO 方式实现。

timer	recent_cpu			priority			thread
ticks	A	B	C	A	B	C	to run
0	0	0	0	63	61	59	ABC

4	4	0	0	62	61	59	ABC
8	8	0	0	61	61	59	ABC
12	12	0	0	60	61	59	BAC
16	12	4	0	60	60	59	BAC
20	12	8	0	60	59	59	ACB
24	16	8	0	59	59	59	ACB
28	20	8	0	58	59	59	CBA
32	20	8	4	58	59	58	BAC
36	20	12	4	58	58	58	BAC

>> C3: Did any ambiguities in the scheduler specification make values
>> in the table uncertain? If so, what rule did you use to resolve
>> them? Does this match the behavior of your scheduler?
有。
（1）当一个线程优先级下降到和 ready_list 中的最高优先级一样时，是执行当前线程，还是执行 ready_list 中的线程。
解决办法：执行当前线程，不交出 CPU，如上表中 ticks 为 4 到 为 8 的时候，线程 A 的优先级下降到 61，但是并不交出 CPU。
（2） 当一个线程优先级下降到和 ready_list 中的非最高优先级一样时，是先执行当前线程，还是执行 ready_list 中已经存在的且为该优先级的线程。
解决办法： 先执行已经存在于 ready_list 中的线程。 例如：
上表中，ticks 为 28 到 32 的时候， 线程 C 优先级下降到 58，等到线程 B 执行完之后应该首先执行线程 A
>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?
在中断过程中经历的时间会 记录于 kernel_ticks 中（或者 user_ticks 中），而 当前线程经历的时间则 记录在 thread_ticks 中，通过比较两者可以得出这中断内外所使用的时间代价。

---- RATIONALE ----

>> C5: Briefly critique your design, pointing out advantages and
>> disadvantages in your design choices. If you were to have extra
>> time to work on this part of the project, how might you choose to
>> refine or improve your design?
本次设计由于刚刚接触 Pintos，并没有完整的设计就开始进入代码，导致浪费时间过多，后来参考温帅哥的实现方法后终于有了头绪，学到的一点是，在没有对整个项目有整体的把握之前不要贸然开始实际的 coding

>> C6: The assignment explains arithmetic for fixed-point math in
>> detail, but it leaves it open to you to implement it. Why did you
>> decide to implement it the way you did? If you created an
>> abstraction layer for fixed-point math, that is, an abstract data
>> type and/or a set of functions or macros to manipulate fixed-point

>> numbers, why did you do so? If not, why not?
fixed-point.h 功能实现：定点实数运算宏定义。由于 pintos 本身没有浮点数类型，而在内核中支持浮点数运算将会大大降低内核的运算速度，事实上，很多商业操作系统内核都是不支持浮点数运算的。而 load_avg, recent_cpu 并不是整形，所以我们需要用整数来表示实数，采用的办法就是一个 int_32 型整数的低十四位默认为小数点后的位数，然后对转换后的数进行操作。 采用宏定义即可完成，不需要使用函数，以加快速度。

源代码摘要：

@file timer.c

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
   enum intr_level old_level = intr_disable();
   ticks++;
   thread_foreach(checkInvoke,NULL);
   intr_set_level(old_level);
   thread_tick ();
}

/* Sleeps for approximately TICKS timer ticks.    Interrupts must
     be turned on. */
void
timer_sleep (int64_t ticks)
{
   if(ticks <= 0){
         return;
   }
   struct thread* curThread = thread_current();
   curThread->ticks_block = ticks;
   enum intr_level old = intr_disable();
   thread_block();
   intr_set_level(old);
}
```

@file synch.h

```
struct semaphore
  {
    unsigned value;                    /* Current value. */
    struct list waiters;               /* List of waiting threads. */
    struct list_elem holder_elem;
    int lock_priority;
  };

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

@file synch.c

```
void
lock_release (struct lock *lock)
{
  //My code
  struct thread * curr;
  struct list_elem *l;
  struct lock* another;
  enum intr_level old_level;


  curr = thread_current();
    ASSERT(curr->blocked == NULL)
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));
  old_level = intr_disable();
  //当前锁暂时被设置为 NULL，在下一个 sema_down 的时候，就是在等待程序会重新设置
holder
  lock->holder = NULL;
  //将当前 lock 从线程的 locklist 中去掉
  list_remove(&lock->holder_elem);
  lock->lock_priority = PRI_MIN - 1;
  //signal，并且准备交出 cpu
  sema_up (&lock->semaphore);
  //处理完了这个锁，开始处理本线程其他的锁
  if(list_empty(&curr->locks))
```

```
    {
        curr->donated = false;
        //不设置 old_priority,因为 donated 已经为 false 了，只要设置 priority 就行
        thread_set_priority(curr->old_priority);
    }else{
        l = list_front(&curr->locks);
        another = list_entry(l,struct lock,holder_elem);
        if(another->lock_priority != PRI_MIN - 1){
            //another 是当前的 curr 拿到的锁的最后一个，那个锁记录着线程 curr 拿到锁的
时候的优先级
            thread_set_priority_mlfq(curr,another->lock_priority,false);

        }else{
            //没有锁了，直接设置为最最最初的优先级就好，就是没有 donate 那一个
           thread_set_priority(curr->old_priority);
        }
    }
    intr_set_level(old_level);
}

void
lock_acquire (struct lock *lock)
{
    ASSERT (lock != NULL);
    ASSERT (!intr_context ());
    ASSERT (!lock_held_by_current_thread (lock));
    //My code
    //curr 是当前正在运行的，也是想要获取锁的进程
    //thrd 则是获得锁的进程，注意下面有一个过程，使得 thrd 成为最终让 curr 停止的根本
原因，
    //因为存在需要嵌套 donate 的情况

    struct thread *curr,*thrd;
    //当前进程正在争的锁
    struct lock *another;

    enum intr_level old_level;
    old_level = intr_disable();
    curr = thread_current();
    thrd = lock->holder;
    //先设置 curr->blocked,若能够成功取得锁就会去掉
    curr->blocked = another = lock;

    //下面将 curr 的 priority 给 donate 给 thrd，会出现嵌套的情况，具体看文档
```

```
  //若是 mlfqs，则不会出现 donate 的情况，所有都是根据 recent_cpu 和 nice 算的，就可以跳过了
  while(!thread_mlfqs && thrd != NULL && thrd->priority < curr->priority)
  {
  //My code
    thrd->donated = true;
    thread_set_priority_mlfq(thrd,curr->priority,true);
    if(another -> lock_priority < curr->priority)
          {
            another->lock_priority = curr->priority;
            list_remove(&another->holder_elem);
            list_insert_ordered(&thrd->locks,&another->holder_elem,cmp_priority2,NULL);
          }
    if(thrd->status == THREAD_BLOCKED && thrd->blocked != NULL)
    {
        another = thrd->blocked;
        thrd = thrd->blocked->holder;
    }
    else
      break;
  }
  //这时候明显 sema_down 是不可能有结果的，所以线程会一直等待直到能够拿到锁
  sema_down (&lock->semaphore);
  lock->holder = curr;
  lock->lock_priority = curr->priority;
  //printf("\nlock3456:                    %x                    holder:%s, priority: %d    %s\n",lock,curr->name,curr->priority,thread_current()->name);
  curr->blocked = NULL;
  //更新当前线程获得的锁的链表
  list_insert_ordered(&lock->holder->locks,&lock->holder_elem,cmp_priority2,NULL);
  intr_set_level(old_level);
}

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

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

  old_level = intr_disable ();
  //线程一直等待另外的线程 signal，否则不断入列，注意
  while (sema->value == 0)
```

```
      {
      //能够运行这段代码的线程肯定不是在 ready_list 里面的。所以不用担心重复插入
         list_insert_ordered(&sema->waiters,&thread_current()->elem,cmp_priority,NULL);
         thread_block ();
      }
   sema->value--;
   intr_set_level (old_level);
}
void
sema_up (struct semaphore *sema)
{
   struct thread *wake_up,*curr;

   enum intr_level old_level;

   ASSERT (sema != NULL);
   //wake 是信号量 signal 后将会唤醒的线程
   wake_up = NULL;
   curr = thread_current();

   old_level = intr_disable ();

   //当前信号量
   //由于一个线程可能会同时获得几个 locks，因此，可能在处理另外的锁的时候会得到别的线程
   //donate 到的优先级，这时候 sema->waiters 里面的线程可能不是有序的了，当然，在将线程插入到
   //waiters 之前还是要用 insert_order 来尽量减少 sort 时间（毕竟有些不会获得几个锁）
   //注意，约定 thread_unblock 不会马上抢占 CPU，只是会改变线程状态而已，下次 schedule 生效
   if (!list_empty (&sema->waiters))
   {
         list_sort(&sema->waiters,cmp_priority,NULL);
         wake_up = list_entry(list_pop_front(&sema->waiters),struct thread,elem);
         thread_unblock(wake_up);
   }
   sema->value++;
   //当前线程应该是将其 unblock 后，有可能当前执行 sema_up 的线程优先级是比较低的，就要放弃 CPU 了
   //如果只有一个锁，这个情况是不存在的，因为当前的肯定是拿到了最高优先级的（donate 来的），但是，
   //由于有多个锁，可能别的进程会被 donate 更高的 priority(然后被另外的锁 block 掉了)
   if(wake_up != NULL && wake_up->priority > curr->priority){
      thread_yield();
```

```
    }
    intr_set_level (old_level);
}
```

@file thread.c

```
void checkInvoke(struct thread* t,void* aux UNUSED){
      //因为 checkInvoke 直接在 alllist 中取的，所以要确保当前处理的进程是不在 readylist 中
的
      //其实可以改进为另外增加一个 block_list,30 号交时间紧迫，就没有增加
    if(t->status == THREAD_BLOCKED && t->ticks_block > 0){
        t->ticks_block --;
        //如果可以唤醒的话，就 unblock
        if(t->ticks_block == 0){
            thread_unblock(t);
        }
    }
}
```

@file thread.h

```
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. */
      //My code
```

```
    int ticks_block;
    int old_priority;                   //原始的优先级，和 donated 一起用
    struct list locks;                  //当前线程所持有的所有的锁
    bool donated;                       //表示当前的线程是否被捐赠过优先级
    struct lock* blocked;               //当前线程正在被什么锁所 block

    //mlfq

    /*nice*/
    int32_t nice;

    int64_t recent_cpu;
    //mlfq
    //My code
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```