+--------------------+

| CS 140 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+--------------------+

---- Info ----

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

[**阮明康<860797144@qq.com>**](mailto:阮明康%3c860797144@qq.com%3e)

[**吴宏昱<hapda@qq.com>**](mailto:吴宏昱%3chapda@qq.com%3e)

[**安显琴<1244905436@qq.com>**](mailto:安显琴%3c1244905436@qq.com%3e)

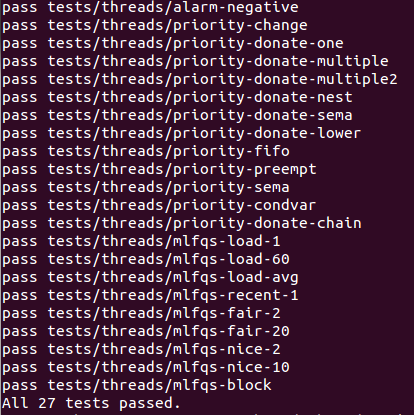
---- PRELIMINARIES ----

>> 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.



（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](#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()](#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](#timer_sleep) (int64\_t ticks)

[timer\_interrupt](#timer_interrupt) (struct intr\_frame \*args UNUSED)

添加函数：

@file: /threads/thread.c

[void checkInvoke](#checkInvoke)(struct thread\* t,void\* aux UNUSED)

1. 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

}

}

1. 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](#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**](#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_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

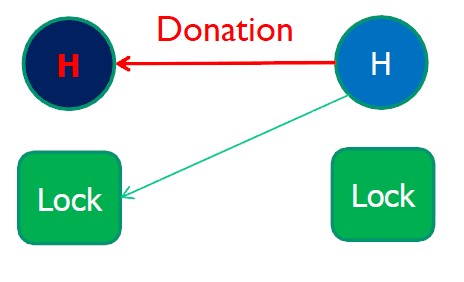
•H 和 L 需要相同的资源 (例如. 某个锁) 而 L 正持有该资源

–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)

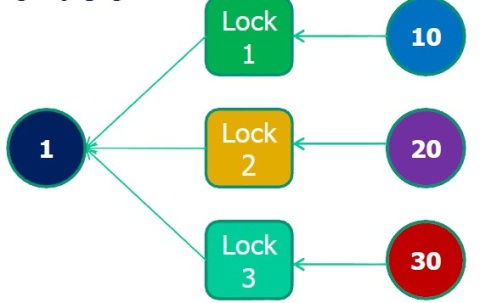
而 H 被阻断，所以实际执行顺序是 M L H



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](#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

1. [struct thread](#struct_thread) 中的list locks， 还有 [struct locks](#struct_lock) 中的 lock\_priority

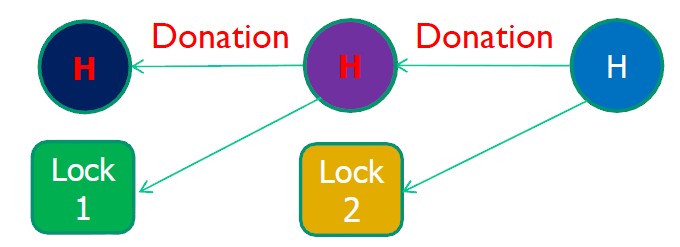


两者配合使用，应对 multiple-donate的情况。由于每次donate的时候都是因为优先级高的一个进程需要申请一个握在优先级比较低的线程手中的，因此锁在涉及到priority-donate的时候维护一个lock\_priority，记录获得这个锁的线程此时的优先级，因为存在multiple-donate，线程可能会接受几个不同的优先级，因此需要在锁中，而不是在线程的结构中维护这样一个信息，以在释放锁，undonate的时候能够将线程优先级恢复到正确的值。

>> 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](#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

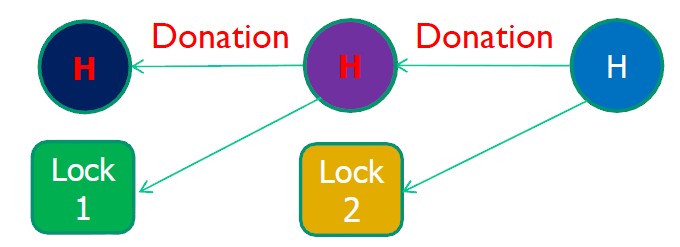
While(正在申请的锁被优先级较低的线程thrd持有){

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\_priority2,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()](#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](#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. \*/

};