## Pintos project 1

小组成员：

赵宇 16281026

周雨豪 16281027

分工各占50%

Pintos:基于80x86架构的一个简单操作系统框架，支持内核线程，加载和运行用户程序，以及文件系统，但是所有这些都是以较为简单的方式实现。

在Ubuntu 16.04 LTS上运行。

（在Ubuntu 16.04 Kylin ，Ubuntu 18.04上boch安装失败）

通过QEMU仿真器加载pintos操作系统。命令如下：

sudo apt-get install qemu // install QEMU

将'pintos.gdb'文件中的变量GDBMACROS复制至HOME/os-

pg/pintos/src/misc/gdb-macros //Set GDBMACROS

cd $HOME/os-pg/pintos/src/utils ; make ; // Compile the Utilities

cd $HOME/os-pg/pintos/src/threads/ // Compile Pintos Kernel

export PATH=$HOME/os-pg/pintos/src/utils:$PATH // Add to Path variable

https://pintosiiith.wordpress.com/2012/09/13/install-pintos-with-qemu/

**1.题目1：进程繁忙等待**

Reimplement timer\_sleep(), defined in devices/timer.c.

Although a working implementation is provided, it "busy waits," that is,it spins in a loop checking the current time and calling thread\_yield() until enough time has gone by. Reimplement it to avoid busy waiting.

初始是使用“busy wait”方式实现的。

即线程不断循环，检查当前时刻并调用thread\_yield()函数，直到时间片耗尽。

这种方式占用了大量的CPU资源！

重新实现的任务是避免“busy wait”，即繁忙等待现象的出现。

“busy wait”原理分析

void timer\_sleep (int64\_t ticks)

Suspends execution of the calling thread until time has advanced by at least x timer ticks. Unless the system is otherwise idle, the thread need not wake up after exactly x ticks. Just put it on the ready queue after they have waited for the right amount of time.

/\* Number of timer ticks since OS booted. \*/

static int64\_t ticks;

ticks 指的是pintos OS自启动以来，系统执行时间的单位量

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

void timer\_sleep (int64\_t ticks)

{

int64\_t start = timer\_ticks (); //获取ticks的当前值，即时间位置

ASSERT (intr\_get\_level () == INTR\_ON); //断言可以被软中断，否则死循环

while (timer\_elapsed (start) < ticks) //在ticks的时间内不断执行

thread\_yield();

}

下面研究thread\_yield()函数。

/\* Yields the CPU. The current thread is not put to sleep and may be scheduled again immediately at the scheduler's whim. \*/

void thread\_yield (void){

struct thread \*cur = thread\_current (); //获取当前线程的起始指针位置

enum intr\_level old\_level;

ASSERT (!intr\_context ()); //断言为软中断

old\_level = intr\_disable ();

if (cur != idle\_thread) //若当前线程并不空闲

list\_push\_back (&ready\_list, &cur->elem);//把当前线程中的元素移至ready\_list就绪队列

cur->status = THREAD\_READY //线程状态改成THREAD\_READY就绪状态

schedule ();

intr\_set\_level (old\_level);

}

下面研究schedule()函数。

/\* Schedules a new process. At entry, interrupts must be off and the running process's state must have been changed from running to some other state. This function finds another thread to run and switches to it. It's not safe to call printf() until thread\_schedule\_tail() has completed. \*/

static void schedule (void)

{

struct thread \*cur = running\_thread (); //获取当前线程cur

struct thread \*next = next\_thread\_to\_run (); //获取下一个要run的线程

struct thread \*prev = NULL;

ASSERT (intr\_get\_level () == INTR\_OFF); //同下

ASSERT (cur->status != THREAD\_RUNNING); //原语操作，确保过程不被中断

ASSERT (is\_thread (next)); //同上

if (cur != next) //若当前线程和下一个要run的线程不是同一个

prev = switch\_threads (cur, next); //切换为cur当前线程

thread\_schedule\_tail (prev); //为下一个要run的线程分配资源等

}

综上所述，timer\_sleep()的功能可以总结为;

在ticks时间内， 若当前线程cur处于running状态，就将其移动到ready\_list中，阻止其继续执行。

后果：当前线程不断在就绪队列和执行状态中切换，即繁忙等待。很浪费系统资源！

解决方案：

在thread结构体中添加一个int成员blocked\_ticks，以记录当前线程处于sleep状态的单位时间长度。

当执行timer\_sleep()时，block当前进程。在每次ticks计数时，将ticks\_blocked减一；

遍历检查所有已有线程，当ticks\_blocked == 0时，调用thread\_unlock()函数唤醒这一进程。



**2.题目2：实现优先级调度alarm-priority**

本题的要求是实现优先级调度priority scheduling。通过对timer\_sleep函数的分析可以得到，我们可以通过维持ready\_list就绪队列为一个优先级队列，即插入线程到ready\_list时保证这个thread是一个优先级队列，来完成题目要求。

在thread.h中可以看到thread结构体中有优先级成员priority：

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. \*/

#ifdef USERPROG

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

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

#endif

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

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

/\* Record the time the thread has been blocked. \*/

int64\_t ticks\_blocked;

};

且对于成员priority有：

/\* Thread priorities. \*/

#define PRI\_MIN 0 /\* Lowest priority. \*/

#define PRI\_DEFAULT 31 /\* Default priority. \*/

#define PRI\_MAX 63 /\* Highest priority. \*/

在/lib/kernel/list.h的源码中找到了一个叫做list\_insert\_ordered的函数，感觉和题目有点关系，看了一下之后真的有点关系：

/\* Inserts ELEM in the proper position in LIST, which must be

sorted according to LESS given auxiliary data AUX.

Runs in O(n) average case in the number of elements in LIST. \*/

Void list\_insert\_ordered (struct list \*list, struct list\_elem \*elem,

list\_less\_func \*less, void \*aux)

{

struct list\_elem \*e;

ASSERT (list != NULL);

ASSERT (elem != NULL);

ASSERT (less != NULL);

for (e = list\_begin (list); e != list\_end (list); e = list\_next (e))

if (less (elem, e, aux))

break;

return list\_insert (e, elem);

}

这个函数能够实现在插入ready list时维持插入的顺序，所以我们还需要加入一个比较功能函数thread\_cmp\_priority（在thread.h中首先声明）。

/\* priority compare function. \*/

Bool thread\_cmp\_priority (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

{

return list\_entry(a, struct thread, elem)->priority > list\_entry(b, struct thread, elem)->priority;

}

然后对init\_thread和thread\_unblock中的list\_push\_back函数替换为list\_insert\_ordered，并作类似修改即可。

关于优先级变化priority-change和抢占调度priority-preempt，通过对源码的分析我们得到，pintos在改变一个thread的优先级时，所有其他的thread的执行顺序都会发生变化。所以我们需要在线程的优先级发生变化时就将其插入ready list中，才能对所有thread重新排序，即：在设置一个线程优先级时，要立即重新考虑所有线程的执行顺序，重新安排所有现场的执行顺序。所以我们直接在线程设置优先级时调用上面讲过的thread\_yield函数即可。

/\* Sets the current thread's priority to NEW\_PRIORITY. \*/

void

thread\_set\_priority (int new\_priority)

{

thread\_current ()->priority = new\_priority;

thread\_yield ();

}

同时在thread\_create函数的最后，需要根据新线程的优先级判断是否要将当前线程插入ready list就绪队列。

/\*Add to run queue. \*/

thread\_unblock (t);

/\* Solution Code \*/

/\* Preempt the current thread \*/

if (thread\_current()->priority < priority)

thread\_yield();

return tid;

}

Failed的还有优先级捐赠的几个test。包括：

priority-donate-one, priority-donate-multiple,

priority-donate-multiple2,priority-donate-nest, priority-donate-sema,priority-donate-lower,

priority-sema,priority-condvar,priority-donate-chain共9个。

我们分别对这9个测试的代码进行分析。由于代码较长，这里略去了代码部分，只阐述分析的结果。

Priority-donate-one中出现了新的函数lock\_acquire,lock\_release，含义分别是锁的获取与释放。线程A获取锁时如果发现拥有锁的线程B优先级比自己低，则提升线程B的优先级；在线程B释放锁后，还要恢复线程B原来的优先级。

Priority-donate-multiple与donate-multiple2中出现两个锁，要求在恢复优先级时，考虑其它线程对该线程的优先级捐赠。这就需要记录给线程捐赠了优先级的所有线程。

Priority-donate-nest中有优先级分别为高中低的三个线程H,M,L。当M在等待L的锁时，L的优先级会被提升为M。而当H来获取M的锁时，M和L的优先级都会提升为H。这等于要求优先级的捐赠是递归捐赠的。因此我们还需要记录线程正在等待哪个线程释放锁。

Priority-donate-sema中出现了sema\_up和sema\_down，对应着V操作和P操作。加入了信号量的调整，但其实没有使问题更复杂，因为锁的本质也是信号量的变化。

Priority-donate-lower则修改了一个优先级被捐赠的线程的优先级。在修改时线程优先级依然是被捐赠的优先级，但释放锁后线程的优先级变成了修改后的优先级。

Priority-sema和Priority-condvar两个测试逻辑要求信号量的等待队列和condition的waiters队列都是优先队列。

donate-chain则要求实现链式优先级捐赠，释放锁后如果线程没有被捐赠，则需立即恢复原来的优先级。

总结一下这些测试的逻辑大概为：

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）。

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。

 · 经过以上分析，我们用代码实现这些逻辑。

首先在thread.h中加入如下内容:

int base\_priority; /\* Base priority. \*/

struct list locks; /\* Locks that the thread is holding. \*/

struct lock \*lock\_waiting; /\* The lock that the thread is waiting for. \*/

在synch.h中的lock struct中加入成员：

struct list\_elem elem; /\* List element for priority donation. \*/

int max\_priority; /\* Max priority among the threads acquiring the lock. \*/

修改synch.c中的lock\_acquire函数：

/\* Acquires LOCK, sleeping until it becomes available if

necessary. The lock must not already be held by the current

thread.

This function may sleep, so it must not be called within an

interrupt handler. This function may be called with

interrupts disabled, but interrupts will be turned back on if

we need to sleep. \*/

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 ();

\*/

/\* Solution Code \*/

struct thread \*cur = thread\_current();

struct lock \*tmp;

/\* Lock is currently held by somethread \*/

if (!thread\_mlfqs && lock->holder != NULL)

{

cur->lock\_waiting4 = lock;

tmp = lock;

while (tmp != NULL && tmp->max\_priority < cur->priority)

{

/\* Update the max priority \*/

tmp->max\_priority = cur->priority;

/\* Donate priority to its holder thread \*/

thread\_donate\_priority(tmp->holder);

/\* Continue donation to threads the holder is waiting for \*/

tmp = tmp->holder->lock\_waiting4;

}

}

sema\_down(&lock->semaphore);

enum intr\_level old\_level = intr\_disable();

cur = thread\_current();

if (!thread\_mlfqs)

{

/\* Now that I've got the lock, I'm not waiting for anylock. \*/

cur->lock\_waiting4 = NULL;

/\* Besides, the max\_priority of this lock must be my priority. \*/

lock->max\_priority = cur->priority;

thread\_hold\_lock(lock);

}

lock->holder = cur;

intr\_set\_level(old\_level);

}

在P操作前递归地实现优先级捐赠，然后再被唤醒后该线程成为这个锁的owner。

while (tmp != NULL && tmp->max\_priority < cur->priority)

{

/\* Update the max priority \*/

tmp->max\_priority = cur->priority;

/\* Donate priority to its holder thread \*/

thread\_donate\_priority(tmp->holder);

/\* Continue donation to threads the holder is waiting for \*/

tmp = tmp->holder->lock\_waiting4;

}

在thread.h中声明：

void thread\_donate\_priority(struct thread \*t);

void thread\_hold\_lock(struct lock \*lock);

void thread\_remove\_lock(struct lock \*lock);

bool lock\_cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED);

void thread\_update\_priority(struct thread \*t);

实现thread\_donate\_priority和thread\_hold\_lock函数：

/\* Donate the priority of current thread to thread t. \*/

void

thread\_donate\_priority(struct thread \*t)

{

enum intr\_level old\_level = intr\_disable();

thread\_update\_priority(t);

/\* Remove the old t and insert the new one in order \*/

if (t->status == THREAD\_READY)

{

list\_remove(&t->elem);

list\_insert\_ordered(&ready\_list, &t->elem, thread\_cmp\_priority, NULL);

}

intr\_set\_level(old\_level);

}

实现lock\_cmp\_priority函数：

/\* Function for lock max priority comparison. \*/

Bool lock\_cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

{

return list\_entry(a, struct lock, elem)->max\_priority > list\_entry(b, struct lock, elem)->max\_priority;

}

之后需要改变释放锁时系统的行为，修改synch.h中的lock\_release函数：

/\* Releases LOCK, which must be owned by the current thread.

An interrupt handler cannot acquire a lock, so it does not

make sense to try to release a lock within an interrupt

handler. \*/

Void lock\_release (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (lock\_held\_by\_current\_thread (lock));

/\* Solution Code \*/

if (!thread\_mlfqs)

thread\_remove\_lock(lock);

lock->holder = NULL;

sema\_up (&lock->semaphore);

}

实现thread\_remove\_lock函数：

/\* Remove the lock for the thread. \*/

void

thread\_remove\_lock(struct lock \*lock)

{

enum intr\_level old\_level = intr\_disable();

list\_remove(&lock->elem);

thread\_update\_priority(thread\_current());

intr\_set\_level(old\_level);

}

别忘了thread\_update\_priority函数，这个函数处理了释放锁时线程优先级的变化，即如果当前线程还有锁，则获取其锁的max\_priority，若其大于base\_priority，则update被捐赠的priority。

/\* Update the thread's priority. \*/

void

thread\_update\_priority(struct thread \*t)

{

enum intr\_level old\_level = intr\_disable();

int max\_pri = t->base\_priority;

int lock\_pri;

/\* If the thread is holding locks, pick the one with the highest max\_priority.

\* And if this priority is greater than the original base priority,

\* the real(donated) priority would be updated.\*/

if (!list\_empty(&t->locks\_holding))

{

list\_sort(&t->locks\_holding, lock\_cmp\_priority, NULL);

lock\_pri = list\_entry(list\_front(&t->locks\_holding), struct lock, elem)->max\_priority;

if (max\_pri < lock\_pri)

max\_pri = lock\_pri;

}

t->priority = max\_pri;

intr\_set\_level(old\_level);

}

最后修改thread\_set\_priority函数;

/\* Sets the current thread's priority to NEW\_PRIORITY. \*/

void

thread\_set\_priority (int new\_priority)

{

/\*

thread\_current ()->priority = new\_priority;

thread\_yield();

\*/

/\* Solution Code \*/

if (thread\_mlfqs)

return;

enum intr\_level old\_level = intr\_disable();

struct thread \*cur = thread\_current();

int old\_priority = cur->priority;

cur->base\_priority = new\_priority;

if (list\_empty(&cur->locks\_holding) || new\_priority > old\_priority)

{

cur->priority = new\_priority;

thread\_yield();

}

intr\_set\_level(old\_level);

}

就搞定了优先级捐赠priority\_donate的全部test。还剩下sema和condvar两个优先级队列。首先要把condition的队列修改为优先级队列，修改synch.c中的cond\_signal函数：

/\* If any threads are waiting on COND (protected by LOCK), then

this function signals one of them to wake up from its wait.

LOCK must be held before calling this function.

An interrupt handler cannot acquire a lock, so it does not

make sense to try to signal a condition variable within an

interrupt handler. \*/

void

cond\_signal (struct condition \*cond, struct lock \*lock UNUSED)

{

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (lock\_held\_by\_current\_thread (lock));

if (!list\_empty (&cond->waiters))

{

/\* Solution Code \*/

list\_sort(&cond->waiters, cond\_cmp\_priority, NULL);

sema\_up (&list\_entry (list\_pop\_front (&cond->waiters), struct semaphore\_elem, elem)->semaphore);

}

}

然后实现条件变量比较函数cond\_cmp\_priority:

/\* Solution Code \*/

/\* Function for condvar waiters priority comparison. \*/

bool

cond\_cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

{

struct semaphore\_elem \*sa = list\_entry(a, struct semaphore\_elem, elem);

struct semaphore\_elem \*sb = list\_entry(b, struct semaphore\_elem, elem);

return list\_entry(list\_front(&sa->semaphore.waiters), struct thread, elem)->priority > \

list\_entry(list\_front(&sb->semaphore.waiters), struct thread, elem)->priority;

}

类似地，分别修改sema\_up和sema\_down：

/\* Up or "V" operation on a semaphore. Increments SEMA's value

and wakes up one thread of those waiting for SEMA, if any.

This function may be called from an interrupt handler. \*/

void

sema\_up (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

if (!list\_empty (&sema->waiters))

{

/\* Solution Code \*/

list\_sort(&sema->waiters, thread\_cmp\_priority, NULL);

thread\_unblock(list\_entry(list\_pop\_front(&sema->waiters), struct thread, elem));

}

sema->value++;

/\* Solution Code \*/

thread\_yield();

intr\_set\_level (old\_level);

}

还有

/\* Down or "P" operation on a semaphore. Waits for SEMA's value

to become positive and then atomically decrements it.

This function may sleep, so it must not be called within an

interrupt handler. This function may be called with

interrupts disabled, but if it sleeps then the next scheduled

thread will probably turn interrupts back on. \*/

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)

{

/\* Solution Code \*/

list\_insert\_ordered(&sema->waiters, &thread\_current ()->elem, thread\_cmp\_priority, NULL);

thread\_block ();

}

sema->value--;

intr\_set\_level (old\_level);

}

Mission 2搞掂。

**3.题目三：多级反馈高级调度实现**

参考资料在这：

[http://www.ccs.neu.edu/home/amislove/teaching/cs5600/fall10/pintos/pintos\_7.html](http://www.ccs.neu.edu/home/amislove/teaching/cs5600/fall10/pintos/pintos_7.html" \t "https://blog.csdn.net/weixin_41705703/article/details/_blank)

要求实现64级调度队列，每个队列对应一个priority，从PRI\_MIN到PRI\_MAX。OS会从高优先级队列开始调度线程，且会随时间推移动态地更新线程的优先级。这个计算涉及到了pintos本身并没有实现的浮点数运算。

The following table summarizes how fixed-point arithmetic operations can be implemented in C. In the table, x and y are fixed-point numbers, nis an integer, fixed-point numbers are in signed p.q format where p + q = 31, and f is 1 << q：

|  |  |
| --- | --- |
| Convert n to fixed point: | n \* f |
| Convert x to integer (rounding toward zero): | x / f |
| Convert x to integer (rounding to nearest): | (x + f / 2) / f if x >= 0,  (x - f / 2) / f if x <= 0. |
| Add x and y: | x + y |
| Subtract y from x: | x - y |
| Add x and n: | x + n \* f |
| Subtract n from x: | x - n \* f |
| Multiply x by y: | ((int64\_t) x) \* y / f |
| Multiply x by n: | x \* n |
| Divide x by y: | ((int64\_t) x) \* f / y |
| Divide x by n: | x / n |

在fixed\_point.h中实现浮点运算逻辑：

#ifndef \_\_THREAD\_FIXED\_POINT\_H

#define \_\_THREAD\_FIXED\_POINT\_H

/\* Basic definitions of fixed point. \*/

typedef int fixed\_t;

/\* 16 LSB used for fractional part. \*/

#define FP\_SHIFT\_AMOUNT 16

/\* Convert a value to a fixed-point value. \*/

#define FP\_CONST(A) ((fixed\_t)(A << FP\_SHIFT\_AMOUNT))

/\* Add two fixed-point value. \*/

#define FP\_ADD(A,B) (A + B)

/\* Add a fixed-point value A and an int value B. \*/

#define FP\_ADD\_MIX(A,B) (A + (B << FP\_SHIFT\_AMOUNT))

/\* Subtract two fixed-point value. \*/

#define FP\_SUB(A,B) (A - B)

/\* Subtract an int value B from a fixed-point value A. \*/

#define FP\_SUB\_MIX(A,B) (A - (B << FP\_SHIFT\_AMOUNT))

/\* Multiply a fixed-point value A by an int value B. \*/

#define FP\_MULT\_MIX(A,B) (A \* B)

/\* Divide a fixed-point value A by an int value B. \*/

#define FP\_DIV\_MIX(A,B) (A / B)

/\* Multiply two fixed-point value. \*/

#define FP\_MULT(A,B) ((fixed\_t)(((int64\_t) A) \* B >> FP\_SHIFT\_AMOUNT))

/\* Divide two fixed-point value. \*/

#define FP\_DIV(A,B) ((fixed\_t)((((int64\_t) A) << FP\_SHIFT\_AMOUNT) / B))

/\* Get the integer part of a fixed-point value. \*/

#define FP\_INT\_PART(A) (A >> FP\_SHIFT\_AMOUNT)

/\* Get the rounded integer of a fixed-point value. \*/

#define FP\_ROUND(A) (A >= 0 ? ((A + (1 << (FP\_SHIFT\_AMOUNT - 1))) >> FP\_SHIFT\_AMOUNT) \

: ((A - (1 << (FP\_SHIFT\_AMOUNT - 1))) >> FP\_SHIFT\_AMOUNT))

#endif /\* threads/fixed-point.h \*/

这里用了16位数表示小数部分，因此对整数的运算从第17位开始。

首先实现timer\_interrupt函数：

if (thread\_mlfqs)

{

thread\_mlfqs\_increase\_recent\_cpu\_by\_one ();

if (ticks % TIMER\_FREQ == 0)

thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu ();

else if (ticks % 4 == 0)

thread\_mlfqs\_update\_priority (thread\_current ());

}

加入以下内容：

/\* Sets the current thread's nice value to NICE. \*/

void

thread\_set\_nice (int nice)

{

thread\_current ()->nice = nice;

thread\_mlfqs\_update\_priority (thread\_current ());

thread\_yield ();

}

/\* Returns the current thread's nice value. \*/

int

thread\_get\_nice (void)

{

return thread\_current ()->nice;

}

/\* Returns 100 times the system load average. \*/

int

thread\_get\_load\_avg (void)

{

return FP\_ROUND (FP\_MULT\_MIX (load\_avg, 100));

}

/\* Returns 100 times the current thread's recent\_cpu value. \*/

int

thread\_get\_recent\_cpu (void)

{

return FP\_ROUND (FP\_MULT\_MIX (thread\_current ()->recent\_cpu, 100));

}

然后在thread.c中声明全局变量load\_avg。

最终测试结果如下，所有test均通过：

### 

### 实验分析：

     本实验做起来还是比较有难度的，初始在安装pintos时就出现了诸多问题，环境的配置比较麻烦，之后在做的过程中发现优先级调度是比较难的部分，一开始不明所以，通过和其他组同学的讨论和实践，慢慢理解其源代码，从而能够在本来基础上进行改进，结合前人的工作进行我们自己的分析逐步去完成这个实验。