Project 1: Threads

> Description: Use the existing thread system to extend the functionality of this system to gain a better understanding of synchronization problems.

The details of the project is located at: Here.

Base Information

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

---- GROUP ----

> Fill in the names and email addresses of your group members.

姓名 学号 权重

刘禹辰 18373223 1

陈新钰 18373198 1

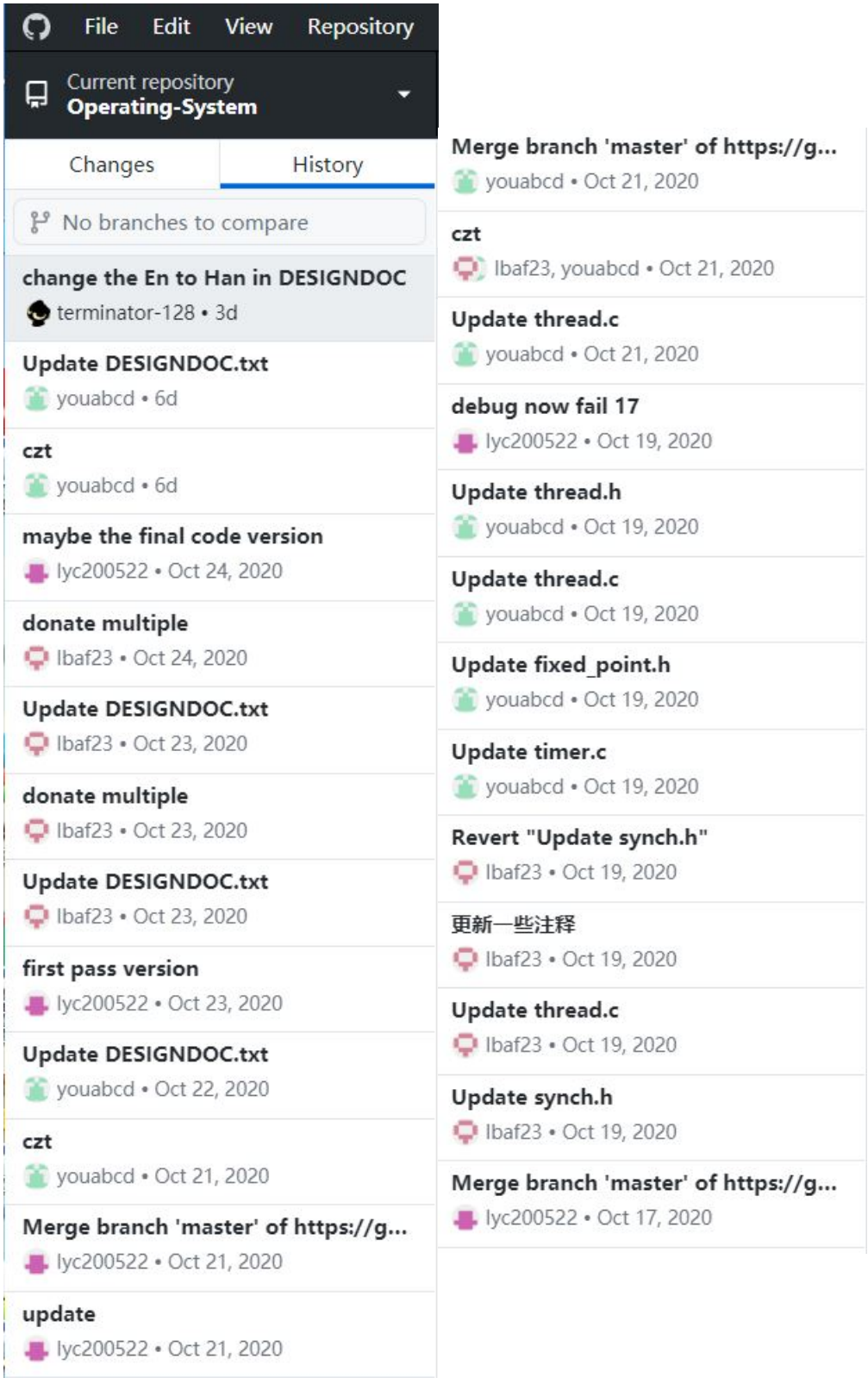
李柯凡 18373238 1

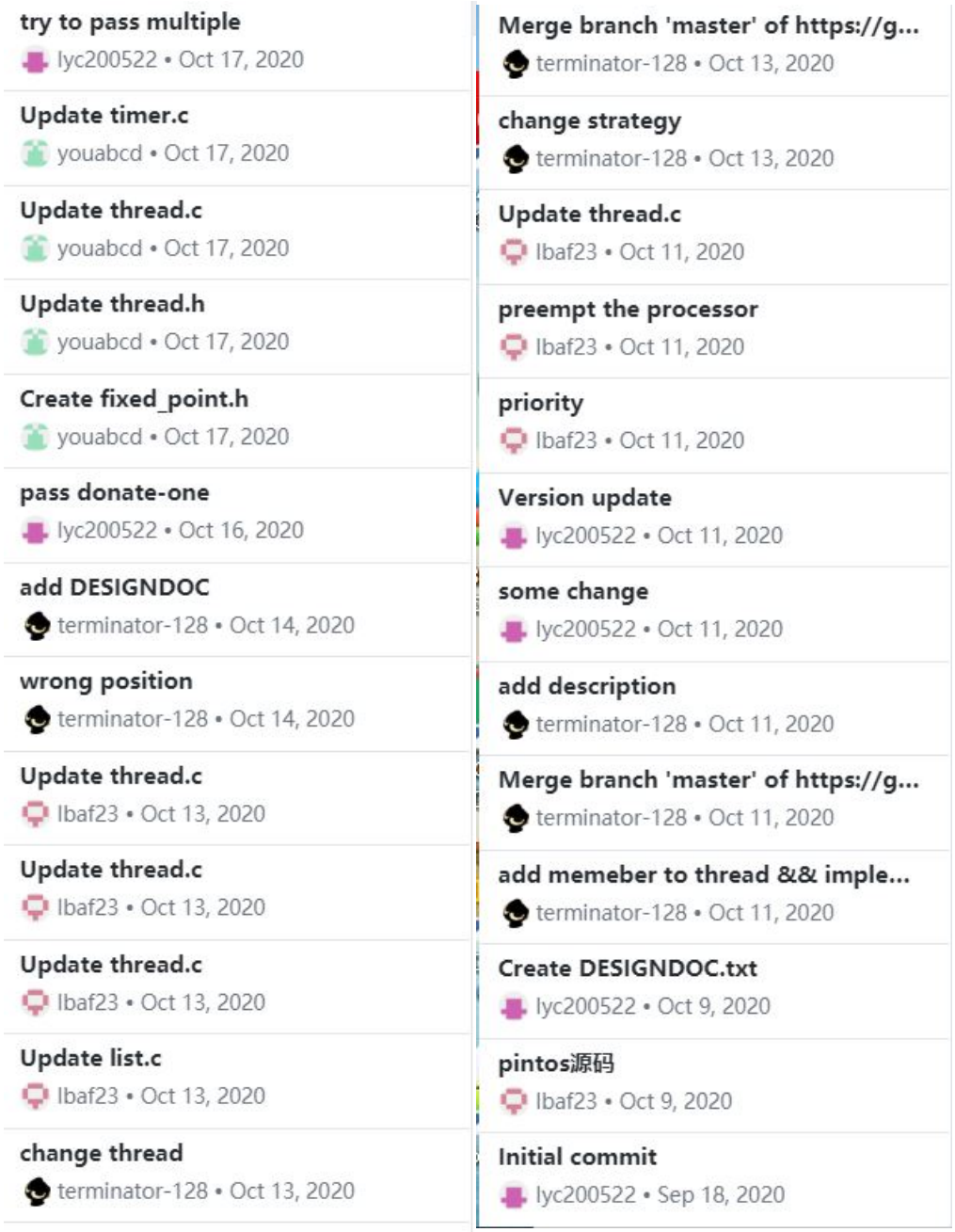
陈振涛 18373285 1

---- PRELIMINARIES ----

> 无

---- Upload records ----





---- Pass Situation ----

Requirements Document

1. Alarm Clock

> First assignment.

1.1 Content

- **Task**: 重新实现 timer_sleep() 来消除忙等待。
- **Requirement**: 拥有同样优先级的多线程在同时唤醒时，遵循FCFS原则；具有不同优先级的多线程在同时唤醒时，遵循优先级调度，之后才是Round Robin调度。 Test Case: alarm-simultaneous, alarm-priority 如果timer_sleep 传入参数为零，则在睡眠之后立即唤醒；如果传入参数为负数，则会直接退出timer_sleep()函数。 Test Case: alarm-negative, alarm_zero. 除非当前线程处于空闲状态，当前线程无需在正好传入ticks扫零后唤醒；只要至少线程前进至少 ticks 个时间，且保持线程间的同步关系。 Test Case: alarm-wait 接口 重新实现timer_sleep() /* devices/timer.c */ Function: void timer_sleep(int64_t ticks)挂起调用线程util的执行时间至少增加了 timer ticks的计时次数。除非系统是空闲的，否则线程不需要在恰好 ticks之后唤醒。只要等待了准确的时间之后，把它放在准备队列后。

2. Priority Scheduling

> Second assignment.

- **任务**：在Pintos中实现优先级调度（优先级捐赠）。
- **需求**:
    1. 当一个线程被添加到***就绪列表***中，该线程的优先级***高于***当前运行的线程，***则当前线程应该***立即将处理器让给新线程。

> 测试用例: **priority-preempt**, **priority-change**.

2. 当一个线程被添加到具有***相同优先级***的就绪列表中时，当这个当前正在运行的线程产生时，它应该将*放在与其具有相同优先级的所有线程之***后**。

   > 测试用例: **priority-fifo**.

3. 当线程正在等待***锁***、***信号量***或***条件变量***时，应首先唤醒***优先级最高的***等待线程。

   > 测试用例: **priority-donate-one**.

4. 一个线程可以在任何时候***提高或降低***它自己的优先级，但是降低它的优先级使它*不再具有最高优先级必须使它****立即yield***。

   > 测试用例: **priority-change**, ***priority-donate-lower***.

5. 分别考虑高、中、低优先级线程***H***、***M***、***L***。如果H需要等待L（在lock、sema或cond的等待列表中），并且M在就绪列表中，那么H将永远无法获得CPU，因为无法获得CPU，但是**H应该在M**之前获得CPU。

   > 测试用例: **priority-donate-multiple2**, **priority-donate-sema**.

6. 当要处理多个捐赠，其中***多个优先级被捐赠给一个线程***的时候，必须在释放锁、信号量或条件之后***恢复***下一个优先级。

   > 测试用例: **priority-donate-multiple**.

7. 处理嵌套捐赠：如果***H***正在等待***M***持有的锁，***M***正在等待***L***持有的锁，则***M***和***L***都应该提升到***H***的优先级。

   > 测试用例: **priority-donate-nest**, **priority-donate-chain**.

8. 需要为锁实现优先级捐赠，而不必为其他Pintos同步构造实现优先级捐赠。***在所有情况下都需要实施优先级调度***。

   > 测试用例: **priority-sema**, **priority-condvar**.

- **Interface**:

  Implement the following functions that allow a thread to examine and modify its own priority.

  ```
  /* threads/thread.c */
  Function: void thread_set_priority(int new_priority)
  ```

  > Sets the current thread's priority to new_priority. If the current thread no longer has the highest priority, yields.

  ```
  /* threads/thread.c */
  Function: int thread_get_priority(void)
  ```

  > Returns the current thread's priority. In the presence of priority donation, return the higher (donated) priority (**runtime priority**).

- **Attention**:
  - You need not provide any interface to allow a thread to directly modify other threads' priorities.
  - The priority scheduler is not used in anay later projects.
- **FAQ**:

There some common questions: Click Here.

> More question, please go to the forum webset: [Here]

3. Advanced Scheduler

> Third assignment.

3.1 Content

- **Task**: Implement a multilevel feedback queue scheduler similar to the BSD scheduler to reduce the average response time for running jobs on your system. See section BSD scheduler , for detailed requirements.

  > More Information about Scheduler: Click Here.

- **Requirement**:

  1. Like the priority scheduler, the advanced scheduler chooses the thread to run ***based on priorities***. However, the advanced scheduler ***does not do priority donation***.

     >

  2. You must write your code to allow us to choose a scheduling algorithm policy at Pintos startup time. By default, ***the priority scheduler must be active***, but we must be able to choose the 4.4BSD scheduler with the "***-mlfqs***" kernel option.

     >

  3.

- **Interface**:

  Implement the functions described below, which are for use by test programs.

  ```
  /* threads/thread.c */
  Function: int thread_get_nice(void)
  ```

  > Returns the current thread's nice value.

  ```
  /* threads/thread.c */
  Function: void thread_set_nice(int new_nice)
  ```

  > Sets the current thread's nice value to ***new_nice*** and ***recalculates the thread's priority based on the new value*** (see section B.2 Calculating Priority). If the running thread no longer has the highest priority, ***yield***s.

  ```
  /* threads/thread.c */
  Function: int thread_get_load_avg(void)
  ```

  > Returns **100 times** the current system load average, rounded to the **nearest integer**.

3.2 多级队列反馈调度

3.2.1 Niceness

- 线程优先级是由调度器使用下面给出的公式动态确定的。见3.2.3

- 然而，每个线程也有一个整数的nice值（尼斯），它决定了该线程对其他线程的友好程度。尼斯值越高，优先级越低。

> nice的值为0，不会影响线程的优先级。正的nice，会降低线程的优先级，让线程放弃一些本来可以获得的CPU时间。nice的取值范围为-20~20。

3.2.2 浮点数的计算

- 这些运算都涉及到浮点数，所以我们需要自己写出基本的浮点运算的宏，具体算法如下。

```
   /* 表示变动的位数*/
   #define FP_Q 14

   /*Convert n to fixed point*/
   #define FP_FP(n) ((int)(n<<FP_Q))

   /*Convert x to integer (rounding toward zero)*/
   #define FP_INT(x) (x>>FP_Q)

   /*Add x and y*/
   #define FP_ADD(x,y) (x+y)

   /*Subtract y from x*/
   #define FP_SUB(x,y) (x-y)

   /*Add x and n*/
   #define FP_ADDN(x,n) (x+(n<<FP_Q))

   /*Subtract n from x*/
   #define FP_SUBN(x,n) (x-(n<<FP_Q))

   /*Multiply x by y*/
   #define FP_MUL(x,y) ((int)(((int64_t) x)*y>>FP_Q))

   /*Multiply x by n*/
   #define FP_MULN(x,n) (x*n)

   /*Divide x by y*/
   #define FP_DIV(x,y) ((int)((((int64_t) x) << FP_Q)/y))

   /*Divide x by n*/
   #define FP_DIVN(x,n) (x/n)

   /* Get rounded integer of a fixed-point value. */
   #define FP_ROUND(x) (x>=0?((x+(1<<(FP_Q-1)))>>FP_Q):((x-(1<<(FP_Q-
1)))>>FP_Q))
```

3.2.3 计算优先级

- 我们的调度器有64个优先级，因此有64个就绪队列，编号为0(PRI_MIN)到63(PRI_MAX)。 线程优先级是在线程初始化时计算的，也可以每隔4个时钟刻度重新计算每个线程的优先级。无论在哪种情况下，它都由以下公式决定

  priority=PRI_MAX - (recent_cpu/4)-(nice*2)

> 其中recent_cpu是线程最近使用的CPU时间的估计值（见下文3.2.4），nice是线程的尼斯值。

结果应四舍五入到最接近的整数（截断）。

> 最近_cpu和nice上的系数分别为1/4和2，在实践中发现效果不错，但缺乏深层含义。计算出的优先级总是被调整为位于PTI_MIN到PRI_MAX的有效范围内。

3.2.4 计算recent_cpu

- 我们希望recent_cpu能够衡量每个进程 "最近 "获得的CPU时间。此外，作为一种改进，最近的CPU时间越大，获得cpu的权重应该越高。

> 一种方法是使用一个n个元素的数组来跟踪过去n秒中每一秒收到的CPU时间。然而，这种方法需要每个线程的O(n)空间和每次计算新的加权平均值的O(n)时间。

- 取而代之的是，我们使用指数加权移动平均线，它采用这种一般形式:

  x(0)=f(0)

  x(t)=a*x(t-1)+f(t)

  a=k/(k+1)

> 其中x(t)是整数时间t>=0时的移动平均数，f(t)是被平均的函数，k>0控制衰减的速度。

  我们可以按照以下几个步骤对公式进行迭代:

  x(1)=f(1)

  x(2)=a*f(1)+f(2)

  ...

  x(5)=a^4f(1)+a^3f(2)+a^2f(3)+af(4)+f(5)

  ...

  x(t)=a^{t-1}f(1)+a^{t-2}f(2)+···+af(t-1)+f(t)

> f(t)的值在时间t上的权重为1，在时间t+1上的权重为a，在时间t+2上的权重为a^2，以此类推。我们还可以关系到x(t)在时间t+k时约为1/e，在时间t+2*k时约为1/e^2，以此类推。从相反的方向看，f(t)在时间t+ln(w)/ln(a)时衰减为权重w。

3.2.5 计算 load_avg

- 最后是load_avg，通常被称为系统负载平均值，作用是估计过去一分钟内准备运行的平均线程数。

> 和recent_cpu一样，它是一个指数加权移动平均数。与priority和recent_cpu不同，load_avg是全系统的，而不是针对线程的。

- 在系统启动时，它被初始化为0，此后每秒钟（tick）更新一次，它按照以下公式更新:

  load_avg = (59/60)*load_avg+(1/60)*ready_threads

> 其中 ready_threads 是更新时正在运行或准备运行的线程数（不包括空闲线程）。

- 由于一些测试所做的假设，load_avg必须在系统tick计数器达到秒的倍数时准确更新，也就是说，当timer_ticks()%TIMER_FREQ==0时才对其进行一次更新，而不是在任何其他时间。

3.2.6 总结

- 以下公式总结了实现调度器所需的计算。它们并不是对调度器要求的完整描述。
- 每个线程都有一个介于-20和20之间的尼斯值（nice），它可以直接控制此线程的优先级。每个线程还有一个优先级，在0(PRI_MIN)到63(PRI_MAX)之间，每隔4个刻度就会用下面的公式重新计算一次。

  priority=PRI_MAX-(recent_cpu/4)-(nice*2)

- 每一个定时器tick一次，正在运行的线程的recent_cpu就会增加1，每秒一次，每个线程的recent_cpu都会这样更新。

  recent_cpu=(2*load_avg)/(2*load_avg+1)*recent_cpu+nice

- 在上面的公式中，需要实数，PintOS还需要解决实数运算的方法。具体方法已在3.2.2中说明。
- 查看更多细节请点击这: https://web.stanford.edu/class/cs140/projects/pintos/pintos_7.html#SEC137

DESIGNDOC

ALARM CLOCK

==========

---- DATA STRUCTURES ----

> A1: 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.

Add to struct thread:

```
/* Member to implementing thread_sleep(). */
int64_t ticks_to_wakeup;            /* Ticks taken to be waken */
struct list_elem sleepelem;         /* List element for sleeping blocked threads
List */
```

---- ALGORITHMS ----

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

thread_sleep() 将获取的睡眠时间赋值给当前线程的ticks_to_wakeup成员变量，之后将当前线程状态设置为 THREAD_BLOCKED 并推入睡眠线程队列。在时钟中断触发之后，会触发子程序检查更新睡眠线程池中的线程剩余睡眠时间并且唤醒已经应该苏醒的线程。

> A3: What steps are taken to minimize the amount of time spent in the timer interrupt handler?

我们创建了一个睡眠线程池，切当时钟中断处理程序触发时，将直接从睡眠线程池中操作与更新，从而减少了判断是否是sleeping thread的时间。

---- SYNCHRONIZATION ----

> A4: How are race conditions avoided when multiple threads call timer_sleep() simultaneously?

当timer_sleep()被当前线程调用时，由于没有采用排序队列，插入仅O(1)复杂度。所以我们采用的措施是先关闭中断，使得其中的操作作为原子指令而不能被打断进行，从而直接避免调用timer_sleep()函数后产生的竞速问题。

> A5: How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?

同样，由于关闭中断，timer_interrupt 在timer_sleep()期间不会发生。

---- RATIONALE ----

> A6: Why did you choose this design? In what ways is it superior to another design you considered?

睡眠线程队列的实现简单，且减少判断花费。实现睡眠线程的数据结构和相关函数在pintos中已经设计好，而且相关函数齐全。当判断是否生成排序队列时，由于采用的所有线程同步更新或唤醒，所以花费并不会显著减少。 相对于其他直接在就绪中搜索的操作，我们显然在效率上高很多。 当然，我们对排序的情况有所考虑，由于tick的int_32可容纳几亿年（以秒为单位），而且FAQ中说明不用担心溢出问题，所以只要在成员变量上稍作修改（需要被唤醒的系统ticks），可以在时钟中断唤醒时降低一定时间复杂度。但即便如此，我们仍然认为，在设计的过程中多余的假设仍然是危险的，在时间精度高（远小于一秒）的情况下难以保证不会溢出。如果维护睡眠线程池为按照ticks_to_wakeup的递增队列，同样需要全部更新，排序队列并不能显著降低时间复杂度。 所以由简洁性、可靠性、有效性，我们选择当前的实现策略。

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.

添加到 struct lock struct list_elem lockelem; /* 锁的元素，为了加入线程的hold_locks列表中 */ *int max_thread_priority;*
/* 等待锁的线程的最大优先级，根据这个值在hold_locks列表中查找捐赠的最大优先级 */

添加到 struct thread struct list hold_locks; /* 存储此线程拥有的锁 */ struct lock *waiting_lock; /* 此线程等待的锁，实现递归捐赠 */ *int original_priority; /* 此线程被捐赠前的优先级，为了恢复线程的原本优先级 */*

> B2: Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation. (Alternately, submit a .png file.)

struct lock 中，lockelem是一个list_elem类型的变量，如果锁被某个线程拥有，则加入 struct thread 的 hold_locks 列表中；max_thread_priority存储等待锁的线程的最大优先级。

struct thread 中，hold_locks列表存储线程拥有的锁，当有更高优先级的线程申请这些锁的时候， 可以提高此线程的优先级；waiting_lock存储此线程等待的锁，当有线程等待此线程拥有的锁，且此 线程同时等待另一个线程拥有的锁的时候，实现递归捐赠优先级；original_priority存储此线程原 本的优先级，如果此线程拥有的锁的max_thread_priority的值没有比original_priority大的，则 恢复此线程的原本优先级。

1. 初始状态，Thread 1 持有锁 lock 1, Thread 2 持有锁 lock 2, 等待锁 lock 1, 已经发生了 Thread 2 对Thread 1的优先级捐赠，Thread 1 的当前优先级变成了 33。

Thread 1	**当前优先级 (priority)**	**原本优先级 (original_priority)**	**持有的锁 (hold_locks)**	**等待的锁 (waiting_lock)**
	33	32	lock 1	NULL
Thread 2	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)
	33	33	lock 2	lock 1
Thread 3	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)

Thread 1	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)

lock 1	最大优先级(max_thread_priority)	持有锁的线程(holder)
	33	Thread 1

lock 2	最大优先级(max_thread_priority)	持有锁的线程(holder)
	33	Thread 2

2. Thread 3 申请 lock 2，产生递归捐赠

(1) 首先向Thread 2捐赠优先级

Thread 1	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)
	33	32	lock 1	NULL
Thread 2	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)
	34	33	lock 2	lock 1
Thread 3	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)
	34	34	NULL	lock 2

lock 1	最大优先级(max_thread_priority)	持有锁的线程(holder)
	33	Thread 1

lock 2	最大优先级(max_thread_priority)	持有锁的线程(holder)
	34	Thread 2

(2) 发现Thread 2 等待锁 lock 1，再次捐赠

Thread 1	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)
	34	32	lock 1	NULL
Thread 2	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)
	34	33	lock 2	lock 1
Thread 3	当前优先级 (priority)	原本优先级 (original_priority)	持有的锁 (hold_locks)	等待的锁 (waiting_lock)
	34	34	NULL	lock 2

lock 1	最大优先级(max_thread_priority)	持有锁的线程(holder)
	34	Thread 1

lock 2	最大优先级(max_thread_priority)	持有锁的线程(holder)
	34	Thread 2

(3) Thread 1没有等待的锁，递归捐赠结束。

---- ALGORITHMS ----

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

等待锁和信号量的线程存放在一个根据线程的优先级排序的优先队列中，按照优先级大小依次唤醒。

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

当前线程申请某个锁的时候，首先判断是否使用循环调度，如果使用，判断锁是否被其它线程拥有。 如果有，假设此线程为线程A。判断当前线程的优先级是否大于锁的最大优先级，如果大于则更改锁 的最大优先级，进行优先级捐赠部分。

捐赠优先级部分：首先提高锁的最大优先级，如果线程A的优先级小于锁的最大优先级则提高线程A的优先级，然后调用thread_yield函数，重新把线程插入优先队列，让优先级最高的线程先运行。

如果线程A等待的锁，被另一个线程拥有，则发生递归捐赠nested donation。此处需要用一个循环来实现，循环条件为：线程等待的锁不为空且阻塞的线程的优先级大于拥有锁的线程的优先级。循环中 设置锁为当前线程等待的锁，并查找拥有线程A等待的锁的线程B，实现递归捐赠优先级，直到线程没有等待的锁为止，或者当循环次数超过八次时，自动停止。

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

在循环调度的情况下，首先从当前线程拥有的锁的队列中移除锁，然后在当前线程的锁队列中找到优先级 最高的锁，将max(锁的优先级，当前线程的原本优先级)设置为当前线程的优先级，最后释放锁，并且调用 thread_yield函数，如果线程释放锁之后不是优先级最高的线程，那么让优先级最高的线程先运行。

---- 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?

当设置优先级时如果此线程的优先级已经被捐赠过，则需要判断捐赠的优先级和设置的优先级谁更高，如果 新设置的优先级高于线程此时的优先级（无论线程有没有被捐赠过），都可以直接把新的优先级赋值给线程 现在的优先级，如果线程的优先级低于线程此时的优先级，则继续判断线程有没有被捐献过，如果线程没有 被优先级捐赠过，那么可以直接降低线程现在的优先级，如果线程被优先级捐赠过，则不能改变线程现在的 优先级。 不可以。

---- RATIONALE ----

> B7: Why did you choose this design? In what ways is it superior to another design you considered?

在处理锁的hold_lock属性时，我们最开始考虑将其实现为一个按照锁的最大优先级排序的优先队列，这样 在取值的时候方便找到最大值，但是由于这个list经常会修改，锁的最大优先级也会修改，所以会导致非常 麻烦，并且效率低下，我们改成了普通队列，并且在需要最大值时循环查找，这样避免了多次排序，降低了 时间复杂度。

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.

changed 'struct' member: int nice; int recent_cpu; nice:nice的值用于影响线程优先级，值为正数降低此线程的优先级，负数则相反。(-20<=nice<=20) recent_cpu:recent_cpu的值用于衡量一个线程最近获得CPU的时间。 changed global variable: static int load_avg; load_avg:load_avg的值用于估计过去一分钟内准备运行的平均线程数。

---- 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:

timer recent_cpu priority thread ticks A B C A B C to run

---

0 0 0 0 63 61 59 A 4 4 0 0 62 61 59 A 8 8 0 0 61 61 59 A 12 12 0 0 60 61 59 B 16 12 4 0 60 60 59 A 20 16 4 0 59 60 59 B 24 16 8 0 59 59 59 A 28 20 8 0 58 59 59 B 32 20 12 0 58 58 59 C 36 20 12 4 58 58 58 A

> 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?

`recent_cpu`的累计是有歧义的。在本次的设计中并没有考虑到CPU计算4次所花

费的时间。处理方法是每四次tick，才会将recent_cpu添加4次。将模糊的时间具体化为每 个tick。 在计算priority，recent_cpu，load_avg这三个数值的公式中都涉及了浮点数的运算， 这些计算过程中会出现数值上的舍入，可能会导致数值不确定。使用定点数学的算理。调度 与之相匹配。

> C4: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

通过调度程序平衡线程的不同调度需求，兼容不同类型的线程，比如需要快速响应的

I/O线程，需要大量CPU时间的复杂计算线程。在调度过程中，CPU需要计算recent_cpu等 数据，导致实际运行的线程会需要占用更久的CPU时间。因此，调度成本在内外中断中上升， 会降低性能。

---- 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?

优点： 1.使用了宏实现了fixe point算法，轻巧方便。 2.断言使用完整，保证出错时能及时终止。 缺点： 1.在线程优先级相同时，选择的线程执行方式为轮换执行，并不能确保一定的顺序。 2.只有一个就绪队列存放就绪的线程，在进行优先级调度时增大了复杂度。 改进： 1.在遇到线程优先级相同的情况下，多加一个判断，比如哪个线程的recent_cpu小，哪个 先运行，可以继续细化比较。 2.创建64个就绪队列，分别存放优先级不同的就绪队列，便于直接调度。

> 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?

这次实验中采用的是新建一个`fixed_point.h`文件，在其中写好一组实现各种定点数

算法的宏。采用此种方式首先调用起来会很方便。其次是能够使用的范围广泛，只要include 此文件就可以在任意地方调用。宏可以直接在源代码中打开，运行时不需再分配内存。所以 比内联函数快，而此算法并不复杂，所以直接使用宏优势比函数明显。

```
            SURVEY QUESTIONS
            ================
```

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

> In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time?

> Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

> Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

> Any other comments?