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

| CS 140 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

18373117 吴朝旭 <1205672770@qq.com> 权重：1

主要负责任务2；

18373526 王勇 <672515431@qq.com > 权重：1

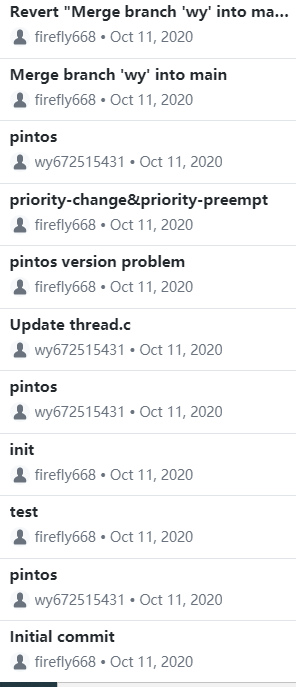
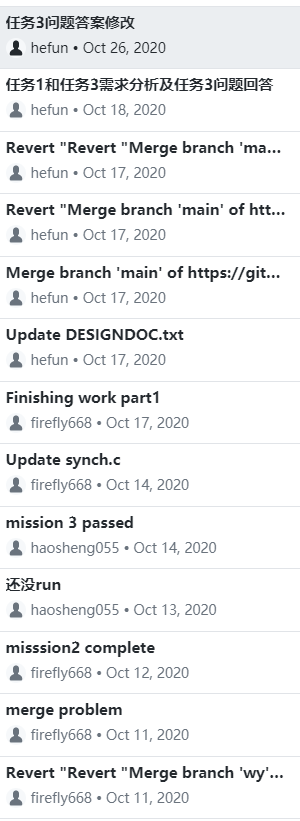
主要负责任务1；

18373554 田震 <1030010026@qq.com> 权重：1

主要负责文档撰写；

18373756 郝晟 <2431811460@qq.com > 权重：1

主要负责任务3。



（hefun—田震，firefly668—吴朝旭，haosheng055—郝晟，wy672515431—王勇）

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

>> 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.

ALARM CLOCK

===========

---- REQUIREMENT ANALYSIS ----

根据timer\_sleep函数的原本的实现我们可以看到，在while循环中，函数不断地对时间是否达到ticks进行判断，如果没有，就调用thread\_yield()，使得线程进入就绪队列。这样会使线程在CPU就绪队列和running队列之间来回切换，占用资源。

我们接下来要对测试用例进行分析。

1. alarm-negative测试：调用了timer\_sleep(-100),我们的程序不能崩溃，也不需要执行任何操作；

2. alarm-zero测试：调用了timer\_sleep(0), 我们的程序应该立即返回；

3. alarm-priority测试：要求当alarm clock唤醒多个线程时，高优先级的线程先运行；

4. alarm-simultaneous测试：多个线程，多次休眠同样的时间，每次应在同一tick下唤醒；

5. alarm-wait测试：多个线程，多次休眠不同的时间，应该能够正常唤醒。

因此我们由以下需求：

1. 避免“忙等待”

2. 对0和负值进行相应的处理

3. 对多个线程、多次休眠的情况能够正常处理

解决方案

首先对ticks进行判断，在其小于等于0时直接返回。

另外，我们维护一个blocked\_list, 将睡眠的线程按照预定醒来时间(unblocked\_tick)由小到大放入。当ticks增加时，我们从前往后将列表中的线程的unblocked\_tick与ticks做比较，如果unblocked\_tick≤ticks则说明此时线程应该被唤醒，如果出现unblocked\_tick>ticks的情况，说明列表后面的线程此时都无需唤醒。这样，我们就避免了“忙等待“，也实现了多个线程、多次休眠的情况的正常处理。

---- 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.

1.thread:

add:

int64\_t unblocked\_tick; /\*调用timer\_sleep的线程唤醒时间\*/

2.static struct list blocked\_list;

储存所有调用timer\_sleep()函数且未被unblock的线程,当线程"休眠"了n个tick之后，从该队列中移除。

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer\_sleep(),

>> including the effects of the timer interrupt handler.

当传入给timer\_sleep()函数的参数小于等于0时,函数直接返回。否则，记录线程应该在哪个tick苏醒.(利用全局变量ticks.)

然后将线程放入上述定义的blocked\_list里面，并调用thread\_block()阻塞线程。

每过一个tick便会触发一个时钟中断。在时钟中断里，我们获取当前的ticks数，然后将blocked\_list扫一遍。

如果当前的ticks数大于等于线程苏醒的ticks，我们便把线程从blocked\_list中移除，

并且调用thread\_unblock()函数解除线程的阻塞态。

>> A3: What steps are taken to minimize the amount of time spent in

>> the timer interrupt handler?

我们在将线程插入blcoked\_list时是利用list\_insert\_ordered函数有序插入的。这样保证

blcoked\_list中的线程是按照苏醒ticks的从小到大排序的。当我们在timer interrupt handler

中遍历blocked\_list时，我们不用完整的遍历，当我们一旦遍历到苏醒时间比当前ticks大的线程时，即可break。

---- SYNCHRONIZATION ----

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

>> timer\_sleep() simultaneously?

当某一个线程进入timer\_sleep函数进入timer\_sleep()函数之后，会调用intr\_disable()函数屏蔽中断，

其他线程调用该函数必须是要在Intr\_on的情况下调用，故避免了资源竞争的问题。

>> A5: How are race conditions avoided when a timer interrupt occurs

>> during a call to timer\_sleep()?

当某一个线程进入timer\_sleep()函数之后，会调用intr\_disable()函数屏蔽中断，保证操作的原子性。

---- RATIONALE ----

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

>> another design you considered?

我们考虑过只加入unblocked\_tick，然后调用系统原本写好的thread\_foreach()函数对all\_list中的线程进行检查，

unblocked\_tick为0就唤醒该线程，但考虑到all\_list可能很长，每个tick都循环一遍消耗时间很长，所以添加了一个比all\_list短

的list：blocked\_list，每次时间中断只用循环检测blocked\_list就行了，且利用了list\_insert\_ordered函数有序插入，不需要完整循环。

PRIORITY SCHEDULING

===================

---- REQUIREMENT ANALYSIS ----

分析题目需求，将需求分成两部分：

1. 无捐赠部分

1.高优先级线程要比低优先级先获得锁（在信号量中完成“P”操作）；

2.执行信号量中的“V”操作后可能唤醒了高优先级线程，需要立即切换；

3.在等待条件变量中高优先级线程比低优先级线程先被唤醒；

4.修改线程优先级后，需要立即按照修改后优先级执行高优先级线程。

1. 捐赠部分

1.符合捐赠条件时，低优先级线程需要能够被捐赠为高优先级线程，且释放锁后恢复到原本优先级；

2.能够递归捐赠；

3.多高优先级线程捐赠给一个低优先级线程时，维持最大值；

4.线程占用多个锁时且都发生捐赠时，在释放其中一个锁时，其优先级应当为剩下捐赠的最大值；

5.捐赠期间修改优先级应在捐赠结束后起作用；

首先看第一部分。

1.原本的代码中sema\_down中是用list\_push\_back将等待线程加入waiters，这种是无序的，我们将其改为list\_insert\_ordered即可（还需编写比较函数）；同时在sema\_up中加入list\_sort函数。

2.原本的代码中sema\_up在unblock一个线程并且sema->value++后就没有了，不能达到要求。只需在return前添加对thread\_yield的调用即可。

3.条件变量情况与信号量类似，但条件变量中的waiters是一个个信号量，只有在执行sema\_down后才能知道线程的优先级，所以我们在cond\_signal中对waiters中第一个信号量执行sema\_up之前执行list\_sort，即可将条件变量的waiters改写成优先队列。

4.原本的代码中thread\_set\_priority只有一句话：thread\_current()->prority = new\_priority;即对线程优先级的修改，这样做不能满足抢占式调度的要求，在最后加上thread\_yield()即可。

第二部分：

该部分较为复杂，且5个要求中，前4个要求对应的代码块是一样的，不能逐一分析、完成编写，需要综合5个要求进行分析，得出修改方案。

1.捐赠应发生在lock\_accquire函数中，执行sema\_down之前完成捐赠，在lock\_release中收回捐赠，确定了修改位置；

2.捐赠过程中要想递归捐赠，我们必须知道lock->holder所等待的锁，以便递归调用；收回捐赠时，线程优先级应为剩下捐赠的最大值（线程可能被多次捐赠，如若只有一次捐赠，应该恢复原优先级），因此需要知道该线程拥有哪些锁，同时在锁上标注等待该锁的最大优先级；同时还需要记录原优先级，以便恢复。

3.避免捐赠期间修改优先级导致捐赠被打断，应改为修改原优先级，在加上判断来确定当前的优先级。

因此，我们得出了需要添加的数据结构，在上诉分析中的位置加入对数据结构的操作即可完成要求。

---- 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:

add:

int original\_priority; /\* Priority before donation\*/

struct lock\* lock\_waiting\_for; /\*该线程所等待的锁\*/

struct list locks; /\*该线程所拥有的锁\*/

struct lock:

add:

int max\_priority; /\* the max priority of thead try to acquire\*/

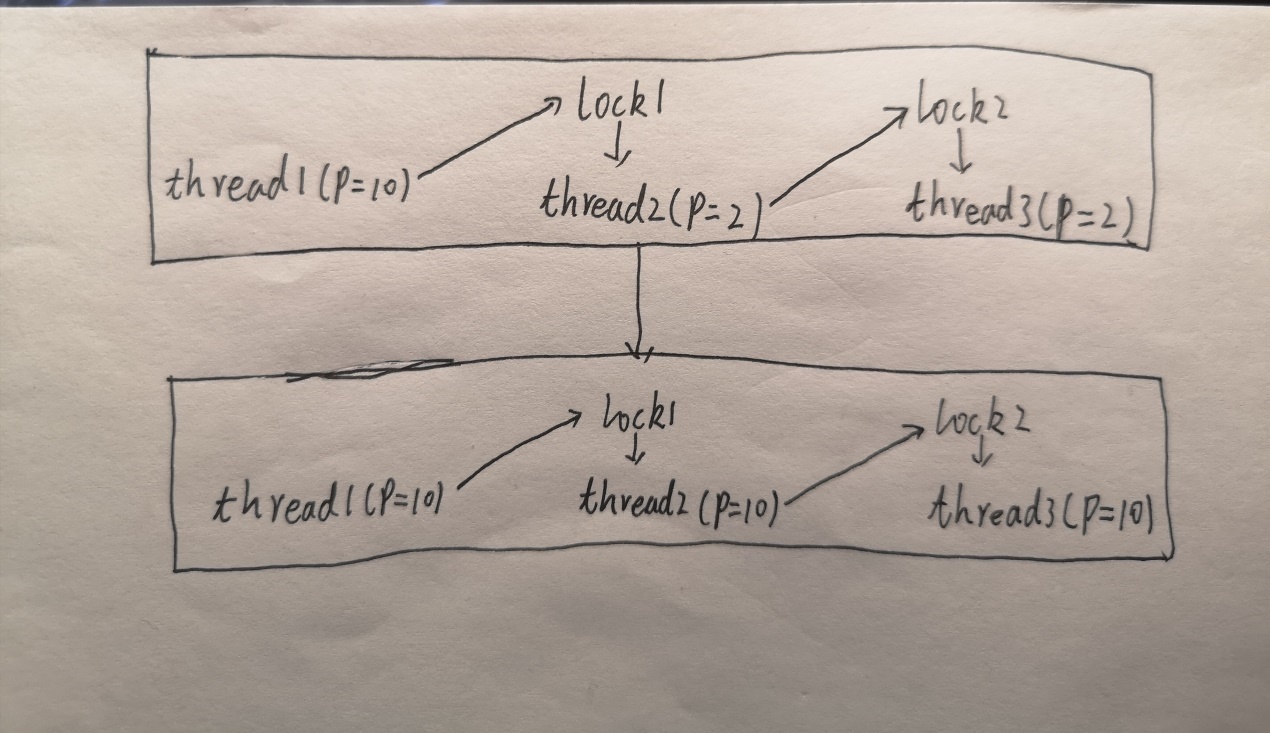
struct list\_elem elem; /\* List element for priority donation. \*/

>> B2: Explain the data structure used to track priority donation.

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

>> .png file.)

解释在B1的注释中；



---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

锁是用信号量实现的，所以与信号量类似；通过将信号量和条件变量的等待队列（waiters）

修改为按优先级排列的优先队列，保证每次唤醒等待队列中最大的一个。

>> B4: Describe the sequence of events when a call to lock\_acquire()

>> causes a priority donation. How is nested donation handled?

首先判断函数执行的前提，即参数lock非空、非外中断、参数lock没有被当前线程占用；

之后关闭中断响应，检查捐赠条件，然后更新参数lock的max\_priority，保证其为所有等待获取该锁的线程的优先级的最大值，

然后将参数lock的拥有线程的优先级改为最大值。如果lock的拥有者在等待其他锁，递归调用该过程，解决嵌套捐赠问题。

之后执行sema\_down()，等待自己获得锁；

sema\_down()执行完毕后，需要更新变量，将lock加入到线程的locks中，lock的所有者改为当前线程，

同时将lock->max\_priority设为该线程的priority。

>> B5: Describe the sequence of events when lock\_release() is called

>> on a lock that a higher-priority thread is waiting for.

首先判断函数执行的前提，即参数lock非空、参数lock被当前线程占用；

之后在释放锁之前，将参数lock从该线程的locks列表中去除，同时设置该线程的优先级为locks中的max\_priority最大值。

---- 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?

在thread中添加了original\_priority属性,thread\_set\_priority()只能修改original\_priority，

priority不能比捐赠优先级更低；

如果使用锁lock\_set来实现，应该在thread\_set\_priority()中调用lock\_accquire()获取该锁，

lock\_set应在捐赠过程中被占用，待捐赠过程完成后再解锁，使得thread\_set\_priority()获得lock\_set后能够继续执行。

---- RATIONALE ----

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

>> another design you considered?

一开始认为也应该在struct lock中加一个等待该锁的thread\_list来确保能够找到等待该锁的最大优先级线程，但其实只用取其最大优先级，而且可以在线程获得锁后修改该值，保证其为等待线程中的最大值（或更大）。因此修改为只添加一个max\_priority属性即可。

ADVANCED SCHEDULER

==================

---- REQUIREMENT ANALYSIS ----

在之前的优先级调度中，线程的优先级是不会改变的，对于一些需要很长CPU时间的线程，只要其优先级够高，就能够一直执行到完成，这样显然不够灵活。我们需要能够根据线程所使用过的cpu时间对线程的优先级进行灵活的调整。

分析题目需求，我们需要实现一个多级反馈调度机制，这包括：

1. 使每个线程有一个介于-20到20之间的nice值和recent\_cpu值（用以度量线程“最近“收到的CPU时间）

2. 一个系统范围的load\_avg值，用以度量过去一分钟内准备运行和正在运行的线程的平均数量

3. 每四个tick对所有线程的优先级根据以下公式进行更新：

4. 每一个tick对正在运行的线程的recent\_cpu加1

5. 每一秒按以下公式与recent\_cpu和load\_avg进行一次更新：

6. 为实现数据的更新，需要实现浮点数的运算

7. 在更新权重后，能够按照任务二中的优先级调度（不包含优先级捐赠）对线程进行调度。

---- 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:

add:

int nice; /\* 用于计算更新线程的优先级\*/

fixed\_t recent\_cpu; /\* 计量线程最近使用的CPU\*/

---- 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 B

12 8 4 0 61 60 59 A

16 12 4 0 60 60 59 B

20 16 8 0 60 59 59 A

24 20 8 0 59 59 59 C

28 20 8 4 59 59 58 B

32 20 12 4 59 58 58 A

36 24 12 4 58 58 58 C

>> 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?

1.如果优先级最高的线程的优先级降低后与别的线程优先级相等，那么接下来的线程如何选择，这在文档中没有描述。

在我们的程序中，如果优先级最高的线程优先级更改后会yield，加入ready\_list中时我们使用list\_insert\_ordered函数，如果有线程优先级相等的情况，新加入的线程会排在已有的与其优先级相等的线程的后面。

>> C4: How is the way you divided the cost of scheduling between code

>> inside and outside interrupt context likely to affect performance?

由于任务3中priority、load\_avg、recent\_cpu的更新操作均与ticks有密切的关系，我们将priority、load\_avg、recent\_cpu的更新操作放在了timer\_interrput函数中。

根据代码分析，timer\_interrput函数是时间中断的中断处理函数，我们在这个函数中消耗的时间会占用线程得到的时间，也就是说，线程使用的时间要少于我们认为它得到的CPU时间，但是我们依然会根据我们认为它得到的CPU时间去降低线程的优先级，这样会导致调度决定的改变。所以我们需要尽可能减少在timer\_interrput函数中的操作。

---- 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?

我们复用了任务2中的队列，而没有按照文档中所提到的使用64个不同优先级的队列。缺点是在线程更新优先级后的插入时间复杂度为O(n)，而使用64个不同优先级的队列插入时的时间复杂度为O(1)；优点是节省了空间。

>> 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?

我们使用了宏定义来进行fixed-point数字的计算，这样做的原因是：

1. fixed-point数字的计算操作用到的地方比较多，每次都直接进行计算会使代码繁琐，逻辑也不清晰。

2. fixed-point数字的计算操作需要完成的操作比较少，如果使用函数，在函数调用时要保存调用函数的现场，执行完后又要恢复调用函数的现场，频繁的计算操作带来的开销就比较大，如果采用宏定义，在预处理阶段即进行了展开，在执行时不需要转换。

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?