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

| CS 140 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

FirstName LastName <email@domain.example>

FirstName LastName <email@domain.example>

FirstName LastName <email@domain.example>

---- 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

===========

---- 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. 在threads中增加float.h，包含浮点运算，因为pintos没有浮点运算。
2. 修改synch.h中的struct lock，增加int max\_priority和struct list\_elem element两个属性来进行队列操作，前者用来记录阻塞队列中的最大优先级，后者用来找到链表中的lock结构体
3. 在float.c中增加static int load\_avg，记录当前的load average。
4. 在thread.h中修改struct thread，增加属性int64\_t ticks\_remain（记录剩余的睡眠ticks）、int old\_priority（记录之前的优先级）、struct lock\* waiting lock（线程阻塞的锁）、以及4.4bsd算法中要求的int nice和int recent\_cpu

---- ALGORITHMS ----

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

>> including the effects of the timer interrupt handler.

首先，更改线程的状态为blocked来阻塞线程，同时在线程结构体里加一个阻塞时间，每次时钟中断的时候递减。Handler在每个tick时都会对线程剩余的阻塞时间进行检测，当此线程的阻塞时间为0时通知该线程，将其唤醒。

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

>> the timer interrupt handler?

首先我们编写一个check\_remain()函数来检查一个阻塞线程剩余的阻塞时间，如果时间不为0，那么我们使它的阻塞时间减1，如果为0就将它转为就绪状态。然后通过thread\_foreach()来在每个tick时检查全部线程。

---- SYNCHRONIZATION ----

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

>> timer\_sleep() simultaneously?

Timer\_sleep()实现的方法是通过阻塞和唤醒机制，而不是通过yield和循环实现的busy waiting，这样我们可以在timer\_sleep()函数开始时关闭中断，来避免出现race condition的情况。

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

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

当一个线程调用timer\_sleep()时，enum intr\_level old\_level = intr\_disable ();这一句中，intr\_disable会将当前线程设置为关中断，同时返回当前状态让old\_level记录，这样可以禁止当前行为被中断。

---- RATIONALE ----

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

>> another design you considered?

这种设计是通过对线程的阻塞(timer\_sleep)与线程的唤醒(timer\_interrupt)来实现的，与通过循环来实现的的busy wait方法相比，这种设计不需要在等待中不断进行循环和检查，节省了CPU资源。

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.

1. 在threads中增加float.h，包含浮点运算，因为pintos没有浮点运算。

2. 修改synch.h中的struct lock，增加int max\_priority和struct list\_elem element两个属性来进行队列操作，前者用来记录阻塞队列中的最大优先级，后者用来找到链表中的lock结构体

1. 在float.c中增加static int load\_avg，记录当前的load average。
2. 在thread.h中修改struct thread，增加属性int64\_t ticks\_remain（记录剩余的睡眠ticks）、int old\_priority（记录之前的优先级）、struct lock\* waiting lock（线程阻塞的锁）、以及4.4bsd算法中要求的int nice和int recent\_cpu

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

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

>> .png file.)

（1）

1. 修改synch.h文件，struct lock增加的属性：

//用来记录阻塞队列中的最大优先级，是用来捐赠给持有锁的线程的

int max\_priority;

//用来找到链表中的lock结构体

struct list\_elem element;

1. 修改thread.h文件，struct thread增加的属性：

//用来记录之前的优先级

int old\_priority;

//用来记录线程阻塞的锁

struct lock\* waiting\_lock;

（2）以如下情况为例：

A 线程，优先级31，持有锁lock\_1.

B 线程，优先级32，持有锁lock\_2, 要获取锁lock\_1

C 线程，优先级33，要获取锁lock\_2

第一阶段：开始状态

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

.---------------------------------------------------.

| 线程 A (开始状态) |

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

| 成员 | 值 |

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

| priority | 31 |

| old\_priority | 31 |

| locks | {lock\_1 (max\_priority = -1)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

.---------------------------------------------------.

| 线程 B (开始状态) |

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

| 成员 | 值 |

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

| priority | 32 |

| old\_priority | 32 |

| locks | {lock\_2 (max\_priority = -1)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

.---------------------------.

| 线程 C (开始状态) |

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

| 成员 | 值 |

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

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | NULL |

'-------------------+-------'

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

第二阶段：B 获取 lock\_1：

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

.---------------------------------------------------.

| 线程 A (B 获取 L1) |

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

| 成员 | 值 |

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

| priority | 31 |

| old\_priority | 32 |

| locks | {lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

.---------------------------------------------------.

| 线程 B (B 获取 L1) |

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

| 成员 | 值 |

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

| priority | 32 |

| old\_priority | 32 |

| locks | {lock\_2 (max\_priority = -1)} |

| waiting\_lock | &lock1 |

'-------------------+-------------------------------'

.---------------------------.

| 线程 C (B 获取 L1) |

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

| 成员 | 值 |

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

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | NULL |

'-------------------+-------'

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

第3阶段（1）：C 获取 lock\_2：

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

.---------------------------------------------------.

| 线程 B (C 获取 L2, Step 1) |

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

| 成员 | 值 |

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

| priority | 32 |

| old\_priority | 33 |

| locks | {lock\_2 (max\_priority = 33)} |

| waiting\_lock | &lock1 |

'-------------------+-------------------------------'

.----------------------------------.

| 线程 C (C 获取 L2, Step 1) |

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

| 成员 | 值 |

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

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | &lock\_2 |

'----------------------+-----------'

.---------------------------------------------------.

| 线程 A (C 获取 L2, Step 1) |

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

| 成员 | 值 |

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

| priority | 31 |

| old\_priority | 32 |

| locks | {lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

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

第3阶段（2）：C 获取 lock\_2：

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

.---------------------------------------------------.

| 线程 B (C 获取 L2, Step 2) |

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

| 成员 | 值 |

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

| priority | 32 |

| old\_priority | 33 |

| locks | {lock\_2 (max\_priority = 33)} |

| waiting\_lock | &lock1 |

'-------------------+-------------------------------'

.----------------------------------.

| 线程 C (C 获取 L2, Step 2) |

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

| 成员 | 值 |

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

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | &lock\_2 |

'----------------------+-----------'

.---------------------------------------------------.

| 线程 A (C 获取 L2, Step 2) |

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

| 成员 | 值 |

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

| priority | 31 |

| old\_priority | 33 |

| locks | {lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

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

第4阶段：A 释放 lock\_1：

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

.-------------------------------.

| 线程 A (A 释放 lock\_1)) |

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

| 成员 | 值 |

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

| priority | 31 |

| old\_priority | 31 |

| locks | {} |

| waiting\_lock | NULL |

'---------------------+---------'

.----------------------------------------------------.

| 线程 B (A 释放 lock\_1) |

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

| 成员 | 值 |

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

| priority | 32 |

| old\_priority | 33 |

| locks | {&lock\_2 (max\_priority = 33), |

| | &lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+--------------------------------'

.------------------------------.

| 线程 C (A 释放 lock\_1) |

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

| 成员 | 值 |

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

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | &lock\_2 |

'--------------------+---------'

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

第5阶段：B 释放 lock\_2：

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

.-------------------------------.

| 线程 A (B 释放 lock\_2)) |

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

| 成员 | 值 |

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

| priority | 31 |

| oldpriority | 31 |

| locks | {} |

| waiting\_lock | NULL |

'---------------------+---------'

.----------------------------------------------------.

| 线程 B (B 释放 lock\_2) |

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

| 成员 | 值 |

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

| priority | 32 |

| old\_priority | 32 |

| locks | {&lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+--------------------------------'

.----------------------------------------------------.

| 线程 C (B 释放 lock\_2) |

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

| 成员 | 值 |

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

| priority | 33 |

| old\_priority | 33 |

| locks | {&lock\_2 (max\_priority = 33)} |

| waiting\_lock | NULL |

'-------------------+--------------------------------'

---- ALGORITHMS ----

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

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

答：将struct semaphore和struct condition中的属性waiters（正在等待的线程队列）改为按照优先级排序的队列。每次唤醒线程时，线程按照在该队列中的顺序依次放到ready\_list中，也就是优先级从最高到最低。这样保证了ready\_list中拥有最高优先级的线程是第一个等待被唤醒的。

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

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

答：

第一步：屏蔽中断

第二步：捐赠

2.1 如果lock的holder为NULL

2.1.1 sema\_down：如果信号量值为0，把所有需要这把锁的线程放到该信号量的等待队列里，直到信号量值变为正值。

2.1.2 设置当前线程为这个锁的持有者

2.2 否则比较lock的holder(L)的优先级和当前线程(C)的优先级

2.2.1如果线程L的优先级大于线程C的优先级

2.2.1.1 执行sema\_down直到信号量值变为正值，也就是直到锁释放

2.2.1.2 设置当前线程为这个锁的持有者

2.2.2 否则

2.2.2.1 [捐赠]将线程L的优先级赋值给线程C的优先级

2.2.2.2 执行sema\_down，直到锁被释放

2.2.2.3 当前线程成为这个锁的持有者

第三步：把中断恢复到被禁用之前的状态

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

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

---- 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?

---- RATIONALE ----

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

>> another design you considered?

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.

---- 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

4

8

12

16

20

24

28

32

36

>> 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?

占位

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

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

---- 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?

>> 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?

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?