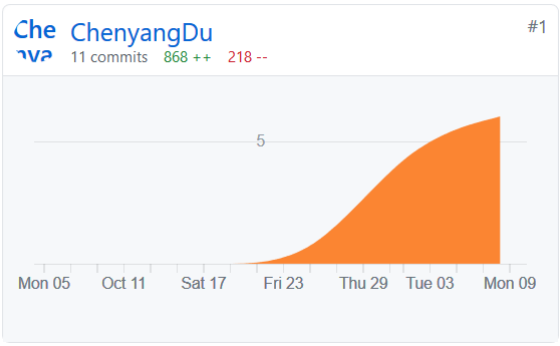
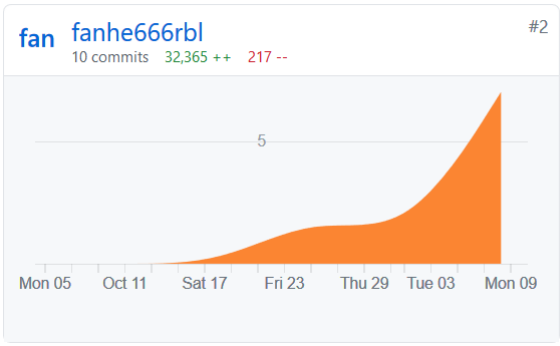
DESIGN DOCUMENT FOR PROJECT 1: THREADS

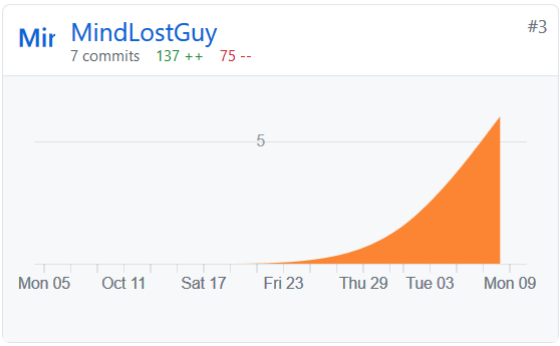
GROUP

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

姓名	学号	权重
杜宸洋	18373119	1.1
王熙林	18373491	1.1
任博林	18373691	1.1
高 锐	17351046	0.7







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.

```
添加到结构体thread(struct thread)
    /* 被原有函数 timer_sleep(),thread_create()和
     * 新建函数 blocked_time_check所引用
     */
    int64_t blocked_time /* 用于记录所设置的阻塞时间 */

添加静态队列list
    /* 用于存储 Mission 1中的睡眠队列，提高效率 */
    static struct list sleeping_list; /* 存储睡眠队列 */
```

ALGORITHMS

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

在对timer_sleep()的一次调用中：

1. 首先判断输入ticks是否为正，是否不处于外中断状态，是否可以进行中断，满足条件则继续，否则退出或者报错。
2. 获取当前中断状态并禁止中断，进行一个原子操作。
3. 调用函数sleeping_list_insert ()（睡眠队列插入），ticks作为参数导入。
4. 在上述函数中首先获取当前进程，之后将当前进程插入睡眠队列，并将当前进程的blocked_time设为ticks。
5. 阻塞当前进程，返回timer_sleep()。
6. 恢复中断状态，结束原子操作。

在时钟中断处理（timer interrupt handler）中：

1. 使得ticks加1，进行时间上的计数增加。
2. 调用blocked_time_check()函数，检测睡眠队列中的线程的阻塞时间。（阻塞时间检测函数）
3. 在阻塞时间检测函数中，对于睡眠队列中的每一个线程，先禁止中断，之后查看其blocked_time阻塞时间，并将其减1。
4. 如果blocked_time阻塞时间恰好为0，则直接唤醒线程加入等待队列，并将线程从睡眠队列中删除。
5. 如果blocked_time阻塞时间不为0，则准备进入下一个线程。
6. 每一个线程检测完之后，都恢复中断状态，结束原子操作。
7. 检测完所有睡眠队列中的线程之后，返回时钟中断处理函数。

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

1. 我们设了一个睡眠线程队列，在设置阻塞时间时，将当前线程直接加入到睡眠线程队列，以便于在处理睡眠线程时只需要在睡眠线程队列中进行处理即可。
2. 对于每个blocked_time==0的睡眠线程，在唤醒其之后，直接将其从睡眠队列中删除，防止二次检测拖慢时间。

SYNCHRONIZATION

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

我们在调用timer_sleep()时，函数内部禁止了中断，实际上是进行一个原子操作。

因此不可能被同时占用。

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

调用timer_sleep()时，函数内部禁止了中断，不可能响应时钟中断。

RATIONALE

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

初次选择时，我们实际上选择的时thread_foreach()函数对所有线程进行扫描，在之后的思考中，我们发现这种方法虽然是可行的但是会有很多冗余线程被扫描，尽管他们并未睡眠，因此，我们模仿all_list，参考thread_foreach()函数，在设置ticks时，就将当前线程加入到一个单独的队列sleeping_list之中，在扫描时，也只扫描这个睡眠线程队列，极大减少了冗余线程被扫描的情况，提高了系统效率。并且这种设计可以让我们在出现bug时，只需要考虑sleeping_list相关的函数即可，提高了可维护性和安全性。

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.

```
添加到结构体 struct lock 中：
    struct list_elem elem;      /* 用来构成链表 */
    int max_prioroty;           /* 拥有该锁的所有线程里面最高的优先级 */

添加到结构体 struct thread 中：
    int64_t blocked_time;           /* 线程等待时间 */
    int original_priority;              /* 线程原本的优先级 */
    struct list lock_list;              /* 线程拥有的锁 */
    struct lock *lock_waiting;      /* 正在等待的锁 */
```

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

在锁的结构体中添加锁当前的持有线程，在任意线程获得锁的时候，检查当前的锁，如果当前锁的最大优先级小于当前线程的优先级，则将当前锁的最大优先级更新，然后取得当前锁的持有线程（当前锁未被当前线程获取），如果该线程此时被其他锁阻塞，则将当前检查的锁更新为阻塞该线程的锁，继续检查，直到确保所有相关锁的最大优先级不小于当前线程的优先级。在每次检查的过程中，随着锁优先级的更新，也要对持有该锁的线程中的锁队列进行排序，确保线程中锁队列是一个优先队列，并且将线程的优先级保持为队列第一个锁的最大优先级（如果自身的初始优先级更大则保持自身优先级）。

ALGORITHMS

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

1. 对于信号量，在执行信号量的up操作时先进行等待队列的排序，再将线程unlock。
2. 对于condition，在执行cond_dignal操作时，先对等待队列排序，再执行信号量up操作。
3. 对于锁，每次获得锁后，先完成优先级捐赠，再对处在等待状态中的线程进行等待队列排序，确保唤醒的顺序是依据优先级的。

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

1. 将当前线程等待的锁置为该锁。
2. 进行优先级捐赠，确保所有相关的锁（即自身等待的锁被某线程持有，而某线程又在等待另一个被持有的锁，这样的锁都是相关的锁）的最大优先级高于当前线程的优先级，并维护其持有者的持有锁优先队列，更新其持有者的优先级，完成优先级的嵌套捐赠。
3. 获取锁
4. 当前线程的等待锁置空
5. 锁的最大优先级更新为当前线程的优先级
6. 维护线程持有的锁的优先队列
7. 更新优先级，重新调度 通过循环更新当前检查的锁为等待锁-持有锁链上的锁，确保此锁的最大优先级不小于当前线程的优先级，将所有等待锁-持有锁链上的线程的拥有锁队列维护为优先队列，

将每个线程的优先级更新为持有锁的最大优先级。这样最后能提升所有没有当前线程优先级高，但持有相关锁的线程的优先级，完成优先级的嵌套捐赠。

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

1. 将该锁从该线程的拥有锁队列中删除
2. 如果拥有锁队列不为空，因为拥有锁队列是优先队列，所以从拥有锁队列的头部取出下一个锁，更新当前线程的优先级为初始优先级和下一个锁的最大优先级二者中较大的那个。
3. 释放锁结构体中的信号量。 注：由于所有等待该锁的线程的优先级的最大值已经存储在拥有锁的最大优先级中，所以等待该锁的线程的优先级高低其实不会对当前线程释放锁的过程有影响，当前线程释放锁后，更新自己的优先级，再次调度的时候，根据更新后的优先级调度即可。

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?

当新设置的优先级比被捐赠的优先级低的时候，会产生线程优先级的竞争。我们的解决方法是，设置了线程的原始优先级，更改优先级的操作先更改原始优先级，如果该优先级低于线程当前的优先级，则不做进一步处理，因为此时线程处于被捐赠的状态。如果高于线程当前的优先级，则更改当前优先级，并进行抢占式调度，按照新优先级更新就绪队列。不能用锁解决，锁的操作中也涉及到操作线程的优先级，可能会互相调用，造成死锁。

RATIONALE

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

这个设计将线程的优先级的更新和捐赠实现为其拥有的锁按最大优先级维护的优先队列，不需要维护自身被捐赠的优先级队列，将优先级捐赠的逻辑实现为拥有锁的优先队列的维护，虽然逻辑比较复杂，但实现过程比较方便。

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.

```
1
2  typedef int fixed_t;                /* 用于浮点计算 */
3
4  struct thread 结构体中加入以下成员变量
5      int nice;                       /* 题目中要求的nice */
6      fixed_t recent_cpu;             /* 题目中要求的rcecent_cpu */
7
8  thread.c中加入全局变量
9      fixed_t load_avg;               /* 题目中要求的load_avg */
10
```

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 ticks	RA	RB	RC	PA	PB	PC	thread 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	59	59	59	C
24	16	8	0	59	59	59	B
28	16	8	4	59	59	58	A
32	16	12	4	59	58	58	A
36	20	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?

在多个线程的优先级相同的时候，选择优先级最大的线程是不确定的。为了解决该问题，我们修改了线程优先级的比较函数，以优先级为第一关键字，线程的`recent_cpu`为第二关键字，线程的`nice`为第三关键字，如果两个线程的优先级、`recent_cpu`、`nice`都相同那么就随机选取一个线程。

```
bool thread_pr_cmp (const struct list_elem *a, const struct list_elem *b,
void *aux)
{
  struct thread *thread_a, *thread_b;
  thread_a = list_entry(a, struct thread, elem);
  thread_b = list_entry(b, struct thread, elem);
  if(thread_a->priority != thread_b->priority)
    return thread_a->priority > thread_b->priority;
  else if(thread_a->recent_cpu != thread_b->recent_cpu)
    return thread_a->recent_cpu < thread_b->recent_cpu;
  return thread_a->nice > thread_b->nice;
}
```

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

如果CPU花太多时间进用来计算`recent_cpu`，`load_avg`和`priority`的计算，那么在执行抢占之前，线程将花费大量时间。这样，该线程将无法获得预期的足够的运行时间，并且将运行更长的时间。这将导致自己占用更多的CPU时间，并会提高`load_avg`，导致降低优先级。因此，如果在中断上下文中进行调度的成本增加，则会降低性能。所以我们在每4个tick之后才进行一次计算，更新`recent_cpu`和`nice`等相关值。

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. 在原有系统的基础上只做了少量的修改，新添加的数据结构和变量较少，尽可能优化原有系统。
2. 浮点运算利用了系统原有的`int`类型，利用位运算等操作提升计算速度。

缺点：

1. 在本次实验中，我们只选用了一个队列来存储所有等待调用的线程，而线程的优先级只有64个，那么就会包含很多相同优先级的线程，在多级反馈队列调度算法中，应当使用多个队列来存储等待调度的线程，每个队列内含有若干个相同优先级的线程。
2. 浮点运算方面采用了函数来计算，如果用`define`来定义会提升更多的性能。