Project 1 实验文档

团队

基本信息

姓名	学号
王佳媛	79216012
时冉	79216011
朱章涌	79066004
张健泓	79066005

每位组员的主要工作内容

Git 相关

https://github.com/jianhong1009/Operating-System-Pintos.git

参考资料

1. https://yuandaima1.oschina.io/sunpages/2019/04/02/thread/

实验要求

1. 任务一：唤醒时钟

   在原本的代码实现中，`timer_sleep()` 函数实现的是一个“忙等待”，当线程在调用 `timer_sleep()` 函数之后，会睡眠 `ticks` 个单位时间。

   根据题意，我们所要做的事便是消除这种忙等待。

2. 任务二：优先级调度

   在原始的代码实现中，线程的就绪队列基本采用的是先来先服务（FCFS）的调度方式，即先进入就绪队列的线程在调度时会先获得 CPU，这个实验的目的是将这种调度策略改成优先级调度。即当一个线程被添加到就绪列表中，并且该线程的优先级高于当前正在运行的线程时，当前线程应该立即将处理器交付给新线程。类似地，当有多个线程正在等待锁、信号量或条件变量时，优先级最高的等待线程应该首先被唤醒。

3. 任务三：高级调度

   在优先级调度策略之中，高优先级的进程永远抢占着 CPU，而低优先级线程能获得的时间非常少。在本实验中，我们会实现更加复杂的调度器，该实验所实现的调度器会自动维护线程的优先级。同样的，在任何给定的时间，调度程序从最高优先级的非空队列中选择一个线程。

需求分析

1. 任务一：

   对于任务一而言，题目要求改写timer_sleep函数的实现方式，timer_sleep函数本来的实现方式是通过循环判断经过的时间（time_elapsed）是否小于传入的参数睡眠时间（ticks）。要是小于，就会调用函数thread_yield()，此函数会将当前线程从执行态转变为就绪态，再调度一个线程变为执行态执行。

   此实现方式会造成忙等待的现象，即线程不断在执行态和就绪态之间来回调度切换，导致CPU效率受到影响，而题目要求的便是改写timer_sleep函数的实现方式，消除这种忙等待。

2. 任务二：

   任务二要求我们实现优先级调度。在优先级调度中，优先级最高的线程会先被调度执行，然后才是优先级第二高的线程。根据题目的要求，我们不仅需要考虑线程的就绪队列，还要考虑信号量和条件变量的等待队列。在pintos原本的实现中，线程的就绪队列基本采用的是先来先服务（FCFS）的调度方式，即先进入就绪队列的线程在调度时会先获得 CPU。因此我们只需要把就绪队列改变为优先级队列，那么在调度发生时，被调度的就是优先级最高的线程了。

   除此之外，题目也给出了一个在线程调度时出现的概念，那就是优先级捐赠。优先级捐赠在本题目中指的是针对线程对于锁的获取的。例如：如果线程1拥有较高的优先级，线程2拥有中等的优先级，线程3拥有较低的优先级。这个时候要是线程1想要获取线程3的锁，在等待线程3释放锁，就得把线程1的优先级捐赠给线程3，要不然线程1可能永远都无法获得CPU。这是因为要是线程1没把优先级捐赠给线程3，这时线程2的优先级仍然高过线程3，而线程1则在阻塞中，因此线程2就会先运行，但是线程1的优先级是高过线程2的，所以出现了优先级翻转的问题。因此题目也要求我们在实现优先级调度时考虑到优先级捐赠这个线程行为。

3. 任务三：

   任务三要求实现多级反馈队列调度程序。这次的高级调度程序同样是由进程的优先级来调度进程与任务二的优先级调度程序一样。不同的是任务三的高级调度程序不会执行优先级捐赠。我们可以根据题目制定公式确定优先级。更新nice值，当正在运行的进程不再是最高优先级时，阻塞进程。每四个时钟周期重新计算priority值，防止线程出现饥饿现象，同时也必须确保不超过64个优先级。每秒更新`load_avg`值，确定准备队列中的平均线程数，时时更新`ready_threads`和`priority`值。

设计思路

1. 任务一：

   我们可以让线程在调用timer_sleep函数时进入阻塞态，当睡眠时间为0时，线程再从阻塞态转变为就绪态，这样就不会出现忙等待了。由此可见，我们需要知道线程在哪一时间会结束睡眠，然后在睡眠时间结束时让线程变为就绪态。

   我们先在线程结构体上增加一个成员sleeping_time_left来记录这个线程应该睡眠多长时间。在线程被调用timer_sleep时，sleeping_time_left变量便会被赋值为ticks，即传入的睡眠时间。

   对于CPU而言，每经过一个tick的单位时间，就会引发一次时钟中断，因此我们便能很轻松地利用时钟中断机制来完成要求。我们在时间中断函数内新增了一个判断。这个判断会判断所有线程，若有线程处于阻塞态并且其sleeping_time_left不为0，我们就让sleeping_time_left--，而当sleeping_time_left为0时就表示睡眠时间结束了，这时我们就让该线程从阻塞态转变为就绪态，准备继续运行。

   时钟中断是每一个tick都会发生一次，而timer_sleep函数传入的参数也是以tick为单位，因此通过在发生时钟中断时检测每一个线程状态，就能完成本题要求。

2. 任务二：

我们实现优先级调度的主要思路是，只需要在把线程插入到就绪队列时，保证这个就绪队列是优先队列即可。优先队列最重要的部分便是能根据线程优先度顺序执行，因此我们在把线程插入到就绪队列时不能直接使用`list_push_back()`这个函数把线程插入到队列尾部了。而是要替换成能让线程有序插入的函数，在阅读源代码时我们发现了`list_insert_ordered()`这个函数，于是决定用这个函数来实现优先队列。有了这一思路，我们便开始寻找在哪里使用了`list_push_back()`这个函数进行尾部插入就绪队列。接着我们再使用pintos自带的`list_insert_ordered()`函数把`list_push_back()`替换。通过这个函数再加上自己实现的线程priority对比函数，便实现了基本的优先队列。接着我们发现了一些问题，在设置一个线程的优先度时，必须要重新考虑所有线程的优先度，否则就会造成线程混乱的情况。于是我们在设置线程优先级时调用了`thread_yield()`，这样便能保证线程的运行顺序了。此外，当我们在创建新线程的时候，若新线程的优先度比主线程高也需要使用`thread_yield()`。此时我们便成功实现了优先队列。

接着我们再把重心放到优先级捐赠上，我们的主要解决思路是当就绪队列里的较低优先级线程占用了互斥资源时，导致也在等待此锁的高优先级线程阻塞，便将高优先级线程的优先度捐赠给予低优先度的线程。接着我们开始分析代码的实现思路，我们从前面提出的解决方法进行展开。当一个线程获得了一个锁，如果拥有这个锁的线程优先级比等待队列里的线程优先度低，我们便让这个低优先级的线程接受优先级捐赠，然后在这个原本低优先级的线程释放锁后恢复原来的优先级。通过这个实现方法便解决了优先级顺序执行的问题，同时不会改变线程原本的优先度。因此我们可以在thread结构体里设置一个`original_priority`保存该线程最原始的优先度。

接着我们发现，这个方法只能解决一个线程占用了一个锁的情况，若出现两个锁以上的情况便不适用了。因此我们可以对之前的代码进行改写。在释放锁的时候，将占有这个锁的线程的优先度改成被捐赠的第二优先级，直到不存在其他捐献者了，此时恢复该线程原始的优先度。因此我们还需要在结构体里设置一个成员变量来保存对该线程进行了优先度捐赠的线程。

此时我们已经写出了基本的优先级捐赠代码，接着在测试代码时发现我们忘记了考虑优先级嵌套的问题。举个例子有3个线程分别拥有高、中、低的优先级。然后高优先级线程想获取中优先级线程阻塞的锁，中优先级线程想获取低优先级线程阻塞的锁。在这种情况下，优先级提升应该连环提升，就是中优先级线程的优先级被提升了话，低优先级线程的优先级也应该一起被提升。因此，我们又要考虑是否应该增加一个成员变量来记录这个线程想获取的锁在哪个线程来解决嵌套问题。接着通过测试点要求内容，发现我们还需要实现condition里的waiter队列成为优先队列，因此我们可以沿用一开始时优先队列实现的思路。

3. 任务三:

我们实现高级调度的本质上是跟优先级调度一样的，代码只需加上if来判断是不是`thread_mlfqs`变量就可以了。首先，先创立新文件`fixed-point.h`来编写计算定点数的方法。写好打开`threads.h`文件，在`struct thread`结构体中加入`nice`和`recent_cpu`变量，方便后面计算`nice`和`recent_cpu`值。然后，在`init_thread()`线程中，初始化`nice`和`recent_cpu`，设置为零以免数值发生错误。

然后，我们在`timer_interrupt`中中断时间来计算更新线程的优先级。设立函数`thread_mlfqs_increase_recent_cpu_by_one(void)`。如果当前进程不是空闲进程时，当前进程加1。在`thread_mlfqs_update_load_avg_and_recent_cpu(void)`函数中，每一秒更新`load_avg`的值，跟着公式：`load_avg = 59/60 * load_avg + 1/60 * ready_threads`完成代码。然后用`load_avg`的值更新所有进程的`recent_cpu`值，公式是`load_avg = 59/60 * load_avg + 1/60 * ready_threads`和`priority`值。每4个ticks时，通过`thread_mlfqs_update_priority (struct thread *t)`函数，更新一次当前进程的`priority`值，并确保每个线程的优先级介于0到63之间。我们的主要逻辑思路就完成了。

重难点讲解

1. 任务一:

最难的一点在于理解timer_sleep函数原生的实现方式。我们通过阅读源码，花了很长一段时间才大致搞懂其实现方式。在阅读源码过程中，源码中附带的注释帮了很大的忙，注释都对pintos的每个函数进行了解释，让我们得以快速明白每个函数是具体干什么的。

其次，在阅读源码时，我们发现了好几次这样子的函数调用：

```
enum intr_level old_level = intr_disable ();
intr_set_level (old_level);
```

源码中的有些代码被嵌套在这两句代码当中，通过阅读源码，我们发现他们实现的其实就是保证这两句代码包围着的代码是一组原子操作，即不可分割的。intr_disable ()函数的功能是先返回当前的中断状态，然后停止中断的发生，而intr_set_level则是根据传入的参数来设置中断状态，由此便保证了被他们所围起来的代码是原子操作，无法被中断。

另一个难点便是找出适当及正确的pintos函数来实现我想要的功能，比如如何让线程转变为阻塞态。我们在阅读了thread.c文件和timer.c文件中的源码后，才对pintos自己带有的原生函数有了比较深入的理解，才明白了可以通过调用thread_blocked()函数来让线程进入阻塞态。

踩坑点在于一开始我忘记检查timer_sleep函数传入的参数ticks是否是合法的，就是可能传入的是负数，即不合法的数字。在测试代码的过程中发现有几个测试点不过关后才发现忘了进行合理性检查。

2. 任务二：

   本次任务主要难点是如何去实现优先队列。还有解决优先级捐赠里的各种问题。再加上本次的工作量较大加上问题较为复杂，因此我们花费了很长时间在构思代码。在编写优先队列的代码时的难点在与如何已先进先出的方法实现就绪队列。其中最大的坑点就是我们在编写这部分代码时忘了考虑创建新线程或者改变优先级时要重新进行线程排序。

   当在编写优先级捐赠的代码时，多层嵌套地拥有锁这个情况个人感觉是这个mission2里最困难的部分，一开始我们完全没想到会出现这种情况导致思路卡了很久。通过搜索查询资料后，才知道当出现这种情况是要先进行递归的优先级捐赠，最后再让线程恢复到原始优先级。

3. 任务三：

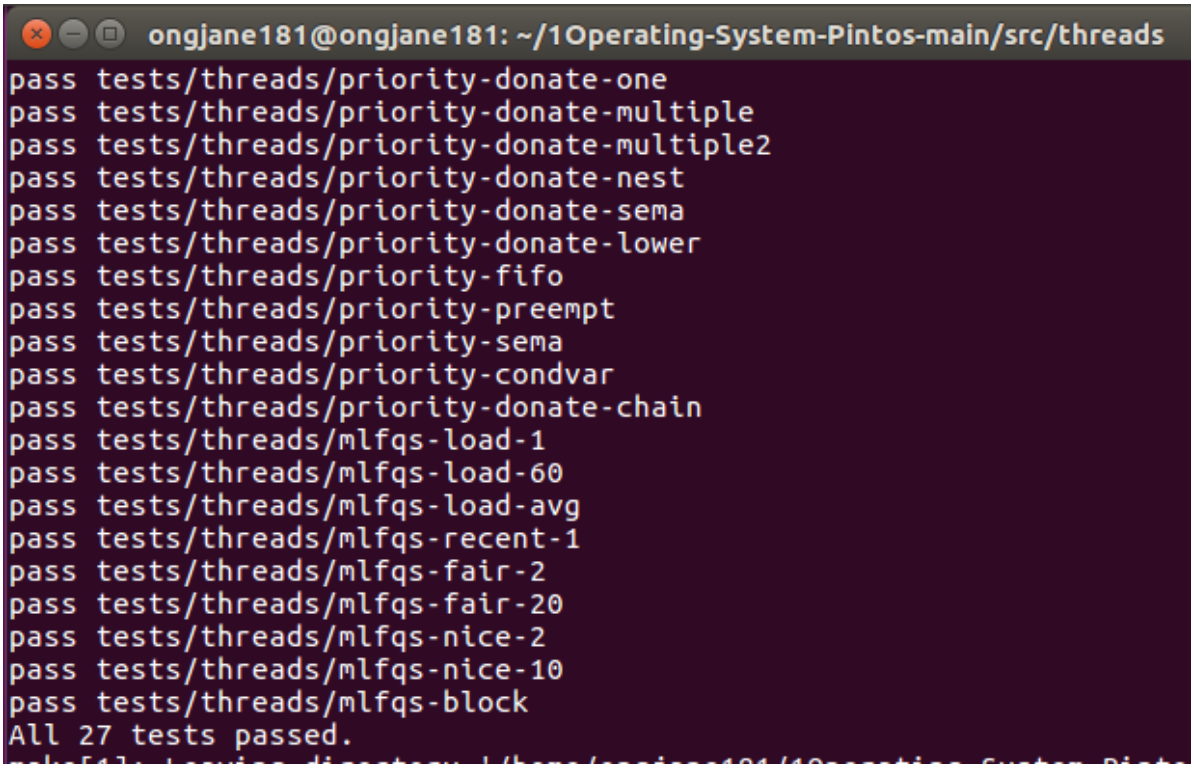
   最难的点是调用浮点运算。这次的任务中主要围绕着浮点运算的公式展开。很容易写着写着就忘了，导致出现很多错误信息。还有一个难点是高级调度问题，任务三的调度与任务二很像系统很难划分出来该使用哪个函数，后来想到了直接设立`thread_mlfqs`变量判断就行了。这任务主要就是环绕着题目给予的公式转换成代码。有个坑点是写了fixed-point.h文件然后忘了把它include到threads.c文件中，导致浪费了很长时间在找bug。

用户手册

以下是代码的运行方式：

1. 进入`src/threads`目录
2. 使用`make clean`命令清除上次编译后的信息
3. 使用`make check`命令对文件进行编译并运行

测试报告



各成员的心得体会

王佳媛：刚开始做这个实验的时候，完全是两眼一抹黑，没接触过，也不知道这是个什么样的概念，随着一步步的实验下来，pintos这个项目让我们脱离了书本上刻板的知识，通过实践更好地理解OS这门课中的许多概念。比如项目中涉及到“优先级调度”、“信号量”、“锁”、“用户栈”、“中断”等等知识点，它们在pintos中串成了一个整体出现，这些概念都以看得见、摸得着的c代码的形式展现于眼前，是一个理解OS非常好的途径。虽说如此，但是我一开始做这个实验的时候，经常有几个测试点通过了，但是几个测试点没通过的情况，很多时候我都不知道是为什么，只能几个组员不断地测试不断地尝试找出问题来（有些时候挺顺利的，但是有些时候找到了问题也是一知半解的感觉）希望随着实验的深入，和学期里的学习，能让我更了解这些，也能让我知道到底是为什么吧.

时冉：在这次的实验来说，pintos是我完全没接触过的东西，安装库和配置环境是非常重要的，在这里我还因为不知名的原因（我觉得我没有漏掉步骤），但是第一个实验的make check就出现了问题，其次很多实验中涉及的内容都是上个学期学过的知识，由于我们留学生一直上网课，我觉得我个人而言，上学期的内容并没有掌握得很扎实，甚至可以说得上是有些懵的地方，这次这个实验，正好是给了我机会去深入学习并且更好的掌握上个学期所学的内容，当然这里也是给了我一个警示，任何知识点都得吃透了。再实验中修改pintos的部分呢，我参考了斯坦福大学官网给出的官方攻略，英语对我来说并不是问题，所以利用英文攻略对我而言可以算是事半功倍。但是这个实验，由于自身的知识储备问题，其实还是有很大的挑战的，我对此是跃跃欲试的，希望能更好地掌握这些知识，并不限于只完成实验作业。

张健泓：我认为这次的实验非常有挑战性。实验内容很难，尤其是一开始面对pintos时完全是懵的。在一开始看到pintos/src下的各个文件夹以及文件夹内的无数源码，我本以为是不可能完成的任务。后来在与组员一起开始阅读源码时，才慢慢的对pintos，或者说操作系统有了更深入的了解。操作系统的复杂性超乎我的现象，每个函数都是层层嵌套调用，因此阅读源码时也非常吃力，幸好源码中的注释提供了不少帮助，让我得以理解函数的使用方式和功能。

朱章涌：这次的实验给我带来了很大的挑战性，光是mission1的里的阅读和理解源代码我就花费了两天。在遇到看不明白的代码部分时在各种谷歌的帮助下，我对pintos操作系统有了个大概的理解。通过这次的试验后我对操作系统里使用到的操作如优先调度这些有了初步的认识。

其他你认为有必要的内容 (Optional)

Project 1 Design Document

QUESTION 1: 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.

```
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. */

    int64_t sleeping_time_left; //记录剩下的睡眠时间，若不为0，表示线程处于睡眠sleep状
态，暂停运行

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
};
```

ALGORITHMS

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

timer_sleep函数的实现方式是通过循环判断经过的时间（time_elapsed）是否小于传入的参数睡眠时间（ticks）。要是小于，就会调用函数thread_yield()，此函数会将当前线程从执行态转变为就绪态，再调度一个线程变为执行态执行。

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

使用最简单的if语句来实现判断。

SYNCHRONIZATION

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

每当我们改变sleeping_time_left的值时，我们都确保已经关闭了中断，使中断不会发生，保证了读写都是原子操作。

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

使用了如下的函数调用来确保被这两个函数包围的代码是原子操作。

```
enum intr_level old_level = intr_disable ();
intr_set_level (old_level);
```

因此在执行被这两句代码所包围的代码时不会产生中断，保证了他们的执行不会被中断所影响。

RATIONALE

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

因为这个实现方式相对而言比较简单，时钟中断本来就是每经过一个tick就发一次，因此可以很方便的借助时钟中断来实现线程的睡眠，只需要在时钟中断发生时判断并且改变线程结构体的sleeping_time_left变量的值就行。

QUESTION 2: PRIORITY SCHEDULING

DATA STRUCTURES

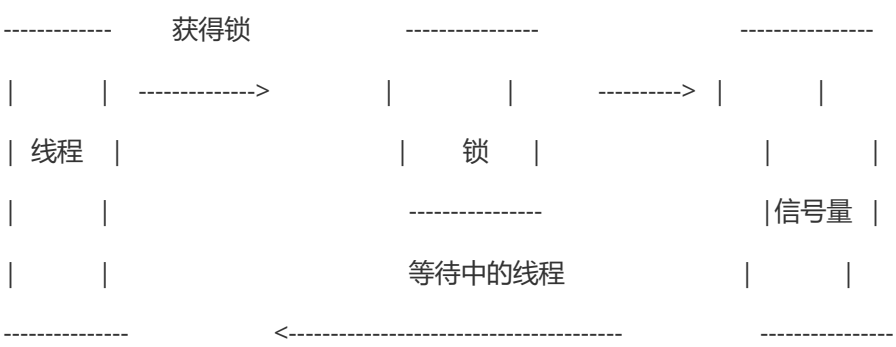
> B1: Copy here the declaration of each new or changed `struct` or `struct` member, global or static variable, `typedef`, or enumeration. Identify the purpose of each in 25 words or less.

```
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. */

    int64_t sleeping_time_left; //记录剩下的睡眠时间，若不为0，表示线程处于睡眠sleep状
态，暂停运行

    int64_t original_priority; // 原本的优先级
    struct list locks; // 线程获得的锁
    struct lock *waiting; // 线程在等待的锁
```

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
};
```

> B2: Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation. (Alternately, paste an image.)

在这个实验里我们使用的是根据某线程所拥有的的锁来记录其他线程的优先度捐赠，同时在加上一个成员变量来保存该线程原本的优先度



ALGORITHMS111

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

如果有锁，就和上一题描述的一样，先把最高优先度需要的锁，一层一层地对线程进行优先级捐赠，在确保了锁释放完毕后，此时调用`thread_yiedl()`，让最高优先度的线程解除阻塞状态并运行

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

当使用`lock_acquire()`函数时，在内部信号量会调用`sema_down()`函数。假如该线程可以可以进入，则此时的信号量会递减。否则，他会被加入等待队列并成为阻塞态，此时调用调度函数`thread_yield()`来选择优先度最高的线程运行.

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

所拥有的锁将会设为null，然后会从成员变量里用来保存锁的数组里进行移除。接着开始调用`sema_up()`函数，开始遍历所有线程并将拥有最高优先度的线程接触阻塞状态。假设刚刚接触阻塞态的线程拥有比现在运行的线程有更高的优先度，则再调用`thread_yield()`函数重新调度

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?

当有新线程被创建或者有线程的优先度被修改时，如果不重新进行调度的话，可能会造成线程混乱的情况。为了避免这种情况，我们在`thread_set_priority()`每当重新赋值线程优先值时，调用`thread_yield()`；`thread_create`则在创建新线程时，假若新线程的优先度比当前运行线程的优先级高，也调用`thread_yield()`重新调度

RATIONALE

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

因为这个实现方法相对比较直观，例如在实现优先队列时，我只需要考虑在把线程插入到就绪队列是已有序排序的方式插入就可以了。通过编写一个`cmp_priority()`函数便能实现了

QUESTION 3: 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.

```
/* threads/fixed-point.h */      //写一个新的文件来编写计算定点数的方法

/*-----------------------------------------------------------------------------*/

/* threads/thread.h */
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. */

    int64_t sleeping_time_left; //记录剩下的睡眠时间，若不为0，表示线程处于睡眠sleep状
态，暂停运行

    int64_t original_priority; // 原本的优先级
    struct list locks; // 线程获得的锁
    struct lock *waiting; // 线程在等待的锁#ifdef USERPROG

    int nice;           //记录nice值
    int recent_cpu;     //记录recent_cpu值

    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    #endif
    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
};

/*-----------------------------------------------------------------------------*/

/* threads/thread.c */
int load_avg;    //用来计算一分钟内准备队列的平均线程数量
```

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	`recent_cpu` **A**	`recent_cpu` **B**	`recent_cpu` **C**	`priority` **A**	`priority` **B**	`priority` **C**	**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	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
24	16	8	4	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	B-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?

是的，如果运行线程与队列中的某个线程具有相同的优先级，则取入就绪队列中的线程，然后使用循环调度规则来实现队列的先后顺序。

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

通过更新当前运行的线程每 4 个ticks的优先级进行更新来减少 `thread_tick()` 中计算出的冗余。

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?

我们的代码设计简单直接，很容易能明白。我们在代码中还减少了新线程产生时在现成的队列上执行排序的次数，我们在每 4 个ticks更新当前运行的线程，避免了计算中的冗余。缺点是我们在队列中经常插入修改导致性能降低。如果有多余的时间，我们想在改进定点数溢出问题。