**2020-2021学年第1学期**

**系统软件课程设计**

**报告**

工作量认定矩阵

说明：

1. 小组每一名成员对自己和其他成员的工作量（百分比）进行评价，并亲笔签字。
2. 每小组参加验收检查时，上交报告封面和成绩评定表。报告全文仅提交电子版、不需要提交打印版。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 被评价人姓名  评价人 | | 冯子健 | 曹炜辰 | 杨正午 | 评价人签字 |
| 学号 | 姓名 |
| 18072004 | 冯子健 | 40% | 30% | 30% | 冯子健 |
| 18062208 | 曹炜辰 | 40% | 30% | 30% | 曹炜辰 |
| 18020007 | 杨正午 | 40% | 30% | 30% | 杨正午 |

成绩评定表

|  |  |  |  |
| --- | --- | --- | --- |
| **评审项目** | **评审内容** | 满分 | 得分 |
| **一、平时检查（每周例行检查的平均分）（20分）** | | | |
| 1.1 分析与设计 | 准确理解需求，明确设计目标，提出合理设计方案 | 10 |  |
| 1.2 过程管理 | 时间投入与工作效果，过程管理方法，运用适当工具，设计文档等 | 10 |  |
| **二、验收检查口头报告与演示（25分）** | | | |
| 2.1 对问题/需求的概括 | 准确、完整描述需求 | 2 |  |
| 2.2 设计与实现 | 设计思路的完整性，关键数据结构、算法的合理性及时空效率，实现效果（运行），用户界面设计，程序设计风格等 | 15 |  |
| 2.3 报告与展示效果 | 报告时间控制，通过图表、流程图等方式汇报和展示设计思想的效果，专业术语规范 | 8 |  |
| **三、验收检查回答问题/修改代码（25分）** | | | |
| 3.1 对问题/需求的理解 | 完整、准确理解提出的问题/新需求 | 5 |  |
| 3.2 回答或修改的效果 | 回答问题时论述的充分性、条理性；针对新需求的设计方案合理性与实现效果（参照2.2） | 12 |  |
| 3.3 表达效果 | （参照2.3） | 5 |  |
| 3.4 项目与过程管理 | 项目管理方法与工具运用，版本控制，文档管理等 | 3 |  |
| **四、课程设计报告（30分）** | | | |
| 4.1 问题与需求分析 | 需求分析的完整性、准确性 | 5 |  |
| 4.2 分析与设计方案 | 设计方案的正确性、完整性，包括数据结构、算法的合理性和时空效率，能够恰当地使用图表表达设计思想（如流程图、类图、对象图、活动图、状态图、序列图等） | 15 |  |
| 4.3 书面表达效果 | 报告格式符合要求，主要包括：完整性（组成部分不缺项），图表符合规范，文字通顺，没有错别字，字体大小合理，条理性等 | 10 |  |
| **总分：** | | **100** |  |
| 评语：  指导教师签字： | | | |

目录

[---- GROUP ---- 3](#_Toc58178013)

[---- PRELIMINARIES ---- 3](#_Toc58178014)

[ALARM CLOCK 3](#_Toc58178015)

[---- DATA STRUCTURES ---- 3](#_Toc58178016)

[---- ALGORITHMS ---- 3](#_Toc58178017)

[---- SYNCHRONIZATION ---- 4](#_Toc58178018)

[---- RATIONALE ---- 4](#_Toc58178019)

[PRIORITY SCHEDULING 5](#_Toc58178020)

[---- DATA STRUCTURES ---- 5](#_Toc58178021)

[---- ALGORITHMS ---- 10](#_Toc58178022)

[---- SYNCHRONIZATION ---- 11](#_Toc58178023)

[---- RATIONALE ---- 11](#_Toc58178024)

[ADVANCED SCHEDULER 12](#_Toc58178025)

[---- DATA STRUCTURES ---- 12](#_Toc58178026)

[---- ALGORITHMS ---- 12](#_Toc58178027)

[---- RATIONALE ---- 13](#_Toc58178028)

[SURVEY QUESTIONS 14](#_Toc58178029)

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

| CS 140 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

冯子健 1473992538@qq.com

曹炜辰 caoweichen@126.com

杨正午 2586753502@qq.com

---- 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.

https://en.wikipedia.org/wiki/Priority\_inversion

https://www.cnblogs.com/feisky/archive/2010/03/08/1680950.html

https://zh.wikipedia.org/wiki/%E5%BF%99%E7%A2%8C%E7%AD%89%E5%BE%85

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.

A:

int tickstowake

加在thread结构体中，用以记录线程被唤醒的时间。

static struct list sleep\_list

创建睡眠队列用以保存被阻塞后正在休眠的线程

---- ALGORITHMS ----

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

>> including the effects of the timer interrupt handler.

A:在timer\_sleep()中，先判断ticks计数值是否大于0，如果小于等于0则直接返回不做处理。如果ticks大于0，先进行中断屏蔽将ticks与系统当前的时间相加保存到新增在thread结构体中的属性tickstowake，记录线程被唤醒的时间，然后将线程按要被唤醒的时间按从小到大顺序加入到新增的睡眠列表中后调用thread\_block()来将线程阻塞。然后恢复中断。

在timer\_interrupt()中，调用thread\_foreachsleep()函数对sleep\_list中的线程进行检查，看其是否到达了自己该被唤醒的时间。若到达则调用thread\_unblock()唤醒该线程。遍历sleep\_list时当遇到第一个不满足唤醒条件的线程就直接终止遍历。

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

>> the timer interrupt handler?

A:

概括：

原先线程忙等待对于CPU的开销很大，是因为timer\_sleep()的实现方法是每次ticks检查时，如果该线程休眠时间小于需要休眠的时间就将不断将线程放到就绪队列中，导致线程会不断在CPU的就绪队列和运行状态间切换，最终导致CPU资源被占用。我们采取的措施是新建一个按照休眠时间从小到大顺序排序的队列用以保存阻塞后休眠的线程，在线程属性tickstowake到达系统时间后再唤醒。

具体步骤：

1. 数据结构补充

在thread结构体中补充变量类型为int 型的属性tickstowake，用于计量线程被唤醒的系统。

1. 每次tick后检查

修改时钟中断处理函数timer\_interrupt，在timer\_interrupt()函数中加入thread\_foreachsleep()函数。thread\_foreachsleep()函数作用是遍历sleep\_list中的每一个线程，判断他们是否到达了要被唤醒的系统时间，若到达了要被唤醒的时间，该线程执行thread\_unblocked()操作，将线程放到就绪队列中。这样就只在线程睡眠时间结束后执行一次将线程放入sleep\_list中的操作，不会使线程在就绪队列和运行状态间多次切换，导致线程一直占用CPU。同时在线程加入到sleep\_list的时候采用的是按照要被唤醒的系统时间升序的方式加入，越靠前的线程越先到达被唤醒的时间，因此遍历sleep\_list时，遍历到第一个不符合唤醒条件的线程就直接终止遍历。这样不用对所有线程进行检查，看哪些线程需要被唤醒，这样所花费的时间会变少，最终大大减少了timer\_interrupt的时间。

---- SYNCHRONIZATION ----

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

>> timer\_sleep() simultaneously?

A:timer\_sleep()执行期间，会调用用 intr\_disable（）函数，执行其中的cli汇编指令，使标志寄存器IF为0，屏蔽中断，函数返回之前的中断状态并用变量保存。这些操作会保证timer\_sleep()函数中的一系列操作是原子操作，所以在同一时间内只会有一个线程进行timer\_sleep(),这就解决了多个线程同时调用timer\_sleep()函数的竞争问题。

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

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

A:timer\_sleep()执行期间，会调用用 intr\_disable()函数，执行其中的cli汇编指令，使标志寄存器IF为0，屏蔽中断，函数返回之前的中断状态并用变量保存。在执行完函数的全部主体功能后，调用intr\_set\_level ()函数，恢复之前的中断状态。以此在执行timer\_sleep()函数时保证不会被中断。

---- RATIONALE ----

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

>> another design you considered?

A:总体来说我们最终所选取的方案直接有效，在thread结构体中新增一个成员变量，通过它去记录线程要被唤醒的时间并通过它来判断何时唤醒线程。在睡眠期间，线程被阻塞并加入sleep\_list中，这样的做法避免了一直占用CPU资源的问题。对于原始做法中出现的反复将一个线程放入就绪队列以达到忙等待效果的方式相比，我们最终选取的解决方案，根本上改变了实现线程休眠的方式。并且按tickstowake升序排列的sleep\_list减少了timer\_interrupt()的判断时间，总而言之，我们的方案既能达到让该线程休眠的目的，又不会一直占据CPU资源使资源被浪费，还减少了timer\_interrupt()的执行时间。我们之前还有一个方案就是对所有线程进行检查，看哪些线程需要被唤醒。但是最终方案中通过有序排列sleep\_list队列，我们不用遍历所有线程。因此我们对最终方案会更方便有效。

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.

A:

int initial\_priority

添加在thread结构体中，保留线程原始的优先级，使得线程可以在释放完锁后可以回到初始的优先级。

struct list hold\_locks

添加在thread因为一个线程可能会拥有多个锁，所以添加此队列来存储这些锁，用以释放时找到这些锁。

struct list\_elem elem

添加在lock结构体中，这个属性使得lock结构体可以被放入hold\_locks这个队列中。

struct thread \*donate\_thread

指向被线程直接捐赠了优先级的线程。

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

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

>> .png file.)

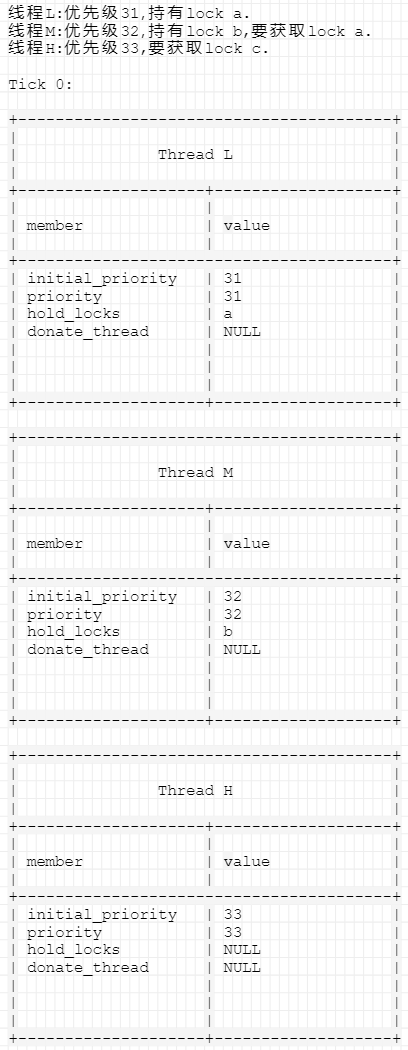
A:

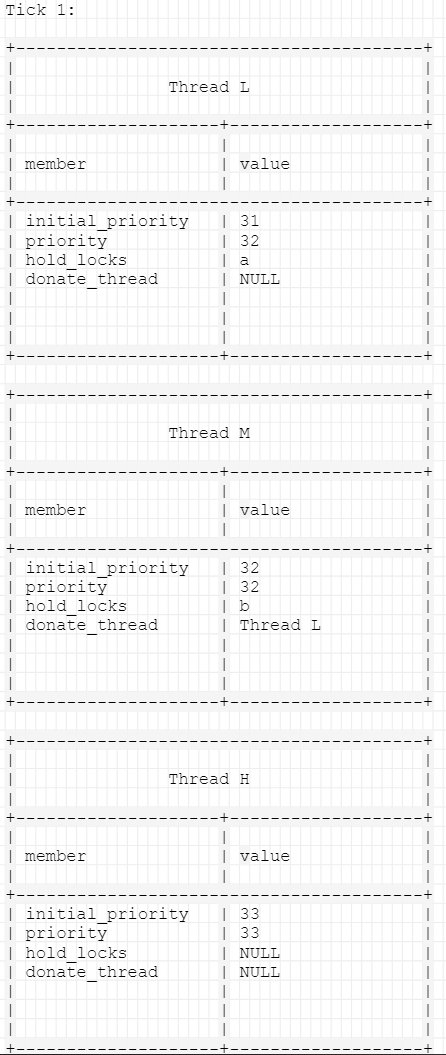
donat\_thread用来在嵌套捐赠优先级的时候，能够一直迭代出将要被当前线程捐赠优先级的捐赠链上的所有线程，并使这个捐赠链上所有线程的优先级都提升到当前线程的优先级来解决优先级反转问题。(举个如同官方档案里面说的，线程H等待线程M，线程M等待线程L，假如当前线程是H，那么就可以根据donate\_thread 将自己的优先级捐赠给M，再通过M的donate\_thread定位到L,并将H自己的优先级捐赠给L。)因此lock\_acquire() 中，就可以先判断是否要捐赠优先级再通过donate\_thread 来嵌套捐赠优先级并最终让线程获得锁。

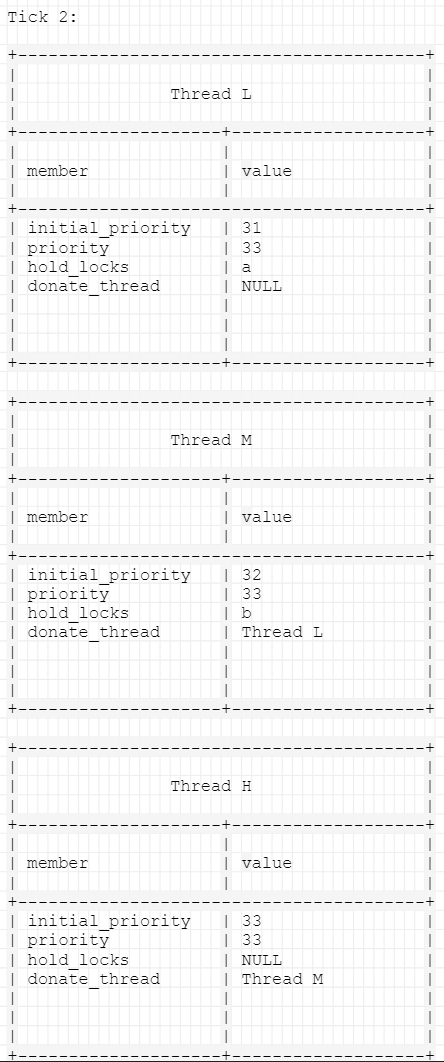
hold\_locks是用来保存某一线程持有的锁的队列，lock\_acquire()时将锁加入到这一队列，以便可以在之后调用lock\_relese()的时候遍历这一队列找到在等待该线程持有的所有锁中的等待者队列中最大优先级的线程，让其与当前线程原优先级比较，取其中较大者作为当前的优先级，从而实现锁的释放以及线程释放完锁后优先级的设置。上述操作中把锁加入到hold\_locks队列中用到了在lock结构体中新添加的属性elem。

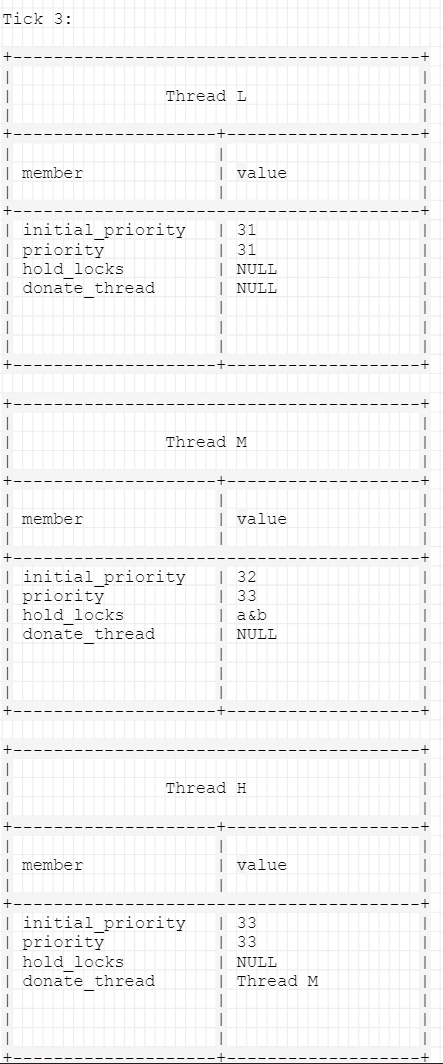
添加的inital\_priority 属性先保存初始优先级，原本在thread\_set\_priority()中将new\_priority 赋值给priority 的操作将赋值给inital\_priority 。如果new\_priority 大于现在的priority 或者当前线程没有被捐赠过,那么就将此时的priority 也给变成new\_priority 。并且initial\_priority属性也保证了线程在释放完锁以后可以最终回到最初的优先级。

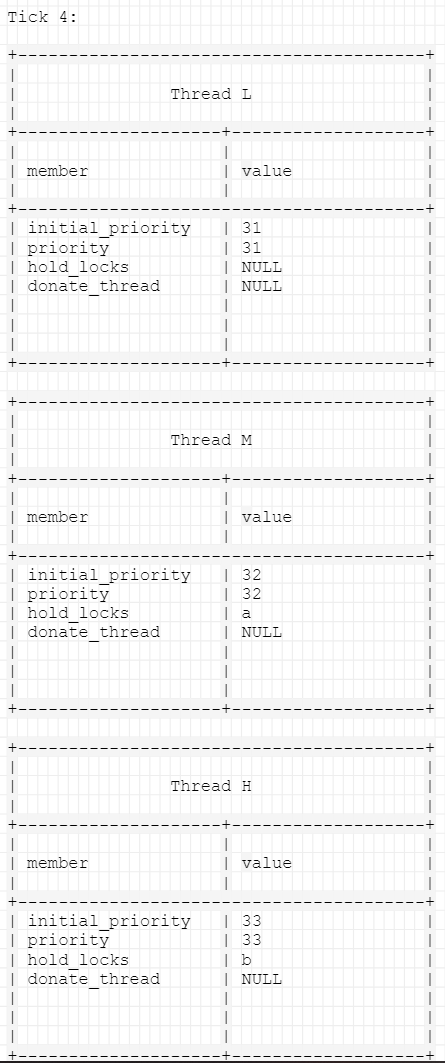
ASCII art:











---- ALGORITHMS ----

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

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

A:原先将线程向队列中放的时候，采取的方式是直接list\_push\_back ()，即直接依次向尾部插入，没有优先级比较。我们现在通过list\_insert\_ordered()将线程按照优先级从大到小的方式插入队列中，维护队列按照从大到小的顺序排列线程。从而在每次唤醒线程时高优先级的线程都会被率先唤醒。

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

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

A:

（1）判断是否需要继续优先级捐赠，如果不需要，进行步骤（3），否则进行步骤（2）。判断是否需要捐赠优先级，即判断当前线程的优先级是否大于正在持有锁的线程的优先级，如果大于则需要捐赠，否则不需要。

（2）优先级捐赠

为了防止优先级反转情况的出现，所以我们一直让锁的持有者是捐赠链上优先级最高的。所以如果需要捐赠优先级，就需要提高捐赠链上的所有线程的优先级。因此我们通过循环进行迭代，把当前线程的优先级赋值给捐赠链上的所有线程，以此防止出现优先级反转情况。并且把当前线程中的donate\_thread属性设置为当前锁的持有者线程。

（3）获取锁

锁的本质是一个信号量，所以在上述过程结束后，对锁的信号量进行sema\_down()操作，并将锁的拥有者设置为当前线程，并将锁加入到当前线程的获得锁的队列中。经过以上步骤，当前线程成功得到需要的锁。

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

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

A:调用lock release()时遍历当前线程所持有的锁的队列，并从所有等待线程中选取优先级最大值。如果取到的最大优先级大于线程的initial\_priority，则将线程优先级设置为该优先级，否则将线程优先级设置为initial\_priority。然后调用将锁的持有者设置为NULL并调用sema\_up()让下一个线程可以获取锁。

---- 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?

A:

在捐赠优先级的过程中，thread\_set\_priority()会改变线程的优先级，优先级捐赠这一过程也会改变线程的优先级。所以这会导致竞争。

为了避免这一竞争。开始时old\_level = intr\_disable()可以禁止中断，避免同时调用时抢占执行，最后intr\_set\_level(old\_level)恢复，允许中断。此做法可以避免同时设置线程优先级的竞争。并且在thread\_set\_priority()中进行判断，如果新的优先级低于线程当前优先级，就只将新的优先级赋值给线程的initial\_priority,这样也能防止重新设置优先级以后发生优先级反转这一情况。这一竞争我们可以用锁来避免，但是我们要在线程结构体中添加一个锁的属性，以此来记录哪个线程被当前线程所捐赠。通过锁结构体中的信号量来实现互斥操作。

---- RATIONALE ----

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

>> another design you considered?

A:我们的方案在捐赠优先级的时候将可能出现的线程之间的一对一捐赠，多对一捐赠以及嵌套捐赠这三种情况合并到了一起，使用一个循环就实现了这三种情况的优先级捐赠。因为不论是这三种情况的哪一种，我们都需要将捐赠链上的线程的优先级提升到当前线程的优先级，所以我们用一个循环递归的完成了捐赠，满足了三种情况的捐赠。因此我们的方案在实行起来十分的简便。在释放锁的时候，我们也考虑的很完全，线程在释放锁后重新设置优先级的时候，线程的优先级之可能是它拥有的锁的等待者中优先级最大者或者它本身的优先级。只要比较出两者的大小就能决定将线程设置为哪个优先级。这种方式也满足三种捐赠情况的需求，因此我们的方案是对三种优先级捐赠情况的一种提炼以及总结，适用于这三种中的任意一种。同时我们还在thread\_set\_priority()中加入了中断屏蔽工作，巧妙的避开了设置优先级时可能会出现的竞争的可能。以上就是我们的设计的优点，总结下来就是简单易行并且满足任何情况。

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.

A:

int64\_t load\_avg

全局变量。

int nice

保存线程优先级的“谦让”程度。

int64\_t recent\_cpu

记录线程最近使用CPU的时间。

---- 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:

A:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| timer ticks |  | recent\_cpu |  |  | priority |  |  |  |
|  | A | B | C | A | B | C | thread to run |  |
| 0 | 0 | 1 | 2 | 63 | 61 | 59 | A |  |
| 4 | 4 | 1 | 2 | 62 | 61 | 59 | A |  |
| 8 | 7 | 2 | 4 | 61 | 61 | 58 | B | 循环调度 |
| 12 | 6 | 6 | 6 | 61 | 59 | 58 | A |  |
| 16 | 9 | 6 | 7 | 60 | 59 | 57 | A |  |
| 20 | 12 | 6 | 8 | 60 | 59 | 57 | A |  |
| 24 | 15 | 6 | 9 | 59 | 59 | 57 | B | 循环调度 |
| 28 | 14 | 10 | 10 | 59 | 58 | 57 | A |  |
| 32 | 16 | 10 | 11 | 58 | 58 | 56 | B | 循环调度 |
| 36 | 15 | 14 | 12 | 59 | 57 | 56 | 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?

A:存在模棱两可的地方，在计算load\_avg,priority,recent\_cpu时，没有考虑到系统计算所花费的时间，这会产生误差，但我们无法直到误差是多少，并把误差考虑进去，能做是忽略误差直接计算，这与多级反馈调度程序的实现匹配，系统每固定周期计算load\_avg,priority,recent\_cpu，未把计算时间考虑进去。

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

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

A:如果CPU花费太长的时间在计算recent\_cpu,load\_avg和priority,就会导致当前线程占用CPU的时间变长，线程的recent\_cpu值变大，导致线程的priority值变化，最终导致线程使用CPU的顺序发生变化。以上是内部中断产生的影响。

---- 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?

A:

设计的优点：

采用编写头文件的方式为定点数操作编写相关宏定义操作，设计的模块化程度提高也减少了代码的重复书写。 同时在进行load\_avg计算时对算式进行优化，减少了操作的项数和步骤，增加了简洁性，程序编写代码整洁规范。将数据类型定义为int\_64防止在计算时出现溢出，提高了程序的鲁棒性。

设计的缺点：

我们的设计只采用了一个就绪队列来保存各就绪的线程，并根据优先级调度，在测试样例中表现正常，这是由于线程数量级不大导致的，时间复杂度为O(n)，如果线程的总数量太大，会导致调度时间过长，影响整个操作系统的效率。

改进：

设置大小为64的队列数字，他们的在数组中的下标对应着队列的优先级，根据优先级将队列插入，这样在调度就绪队列中的程序的时候，效率会更高。

>> 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?

A:我们首先采用头文件的方式对所有定点数运算操作进行整合编写，这提高了功能的可重复利用性，并且采用宏定义编写，减少系统开销，提升运行效率，因为宏定义只占用编译时间，但减少了运行时间，优先级的计算与三个算式相关，且最快的每一秒就要计算一次，如果采用函数调用，在每次调用时都需要时间去分配内存单元，保存现场，传递值，重新载入，这都需要时间，在程序有大量计算时，这些时间是不可被忽视的，这也是我们选择采用宏定义编写而不是函数编写的原因。

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?

A:三个Mission都比较有难度，Mission 2的难度尤其最大。Mission 1在阅读理解代码花费了较多时间，Mission 2在探讨线程优先级捐赠问题上花费了很大精力和时间。

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

A:在Mission 1的设计中，我们学习到了线程阻塞唤醒的机制，以及线程等待时空转问题的解决。在mission2的设计过程中，我们对操作系统中线程的数据结构与优先级调度方法有了更深的印象，同时对系统中资源竞争所造成的优先级翻转现象及其解决方案产生了更深的理解，在实现过程中对操作系统中锁和信号量及其数据结构和锁的获取与释放功能都有了更深的认识。经过Mission 3的学习，我们发现了优先级队列的弊端，并找到了解决方法，那就是动态优先级队列，在实现动态优先级的过程中，对操作系统层面的定点数及其运算有了更深的认识和理解

>> 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?