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

| CS 140 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

张童鑫 [<zhangtongxin@buaa.edu.cn>](mailto:<zhangtongxin@buaa.edu.cn>) 负责部分:ADVANCED SCHEDULER

纪怀宇 [huaiyu11@126.com](mailto:huaiyu11@126.com) 负责部分: PRIORITY SCHEDULING

沙斌竹 [<JerrySha@buaa.edu.cn>](mailto:<JerrySha@buaa.edu.cn>) 负责部分:ALARM CLOCK

黄俊钦 [<email@domain.example>](mailto:<email@domain.example>) 负责部分: PRIORITY SCHEDULING

---- 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.

在 struct thread 中添加：

int64\_t blocked\_time; /\*用于记录线程结束阻塞的时间\*/

添加了静态变量：

static struct list blocked\_list; /\*用以维护一个阻塞中进程的优先队列，阻塞结束时间较早的在前。\*/

---- ALGORITHMS ----

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

>> including the effects of the timer interrupt handler.

当调用 time\_sleep() 后，首先判断参数 ticks 是否非负，若 ticks 非法则直接返回

然后断言当前未屏蔽中断，接着调用 thread\_sleep() 方法

在 thread\_sleep() 方法中首先获取当前线程，断言其处于运行态，并设置其阻塞结束的时间。

然后屏蔽中断，保证后续语句完整执行

接着先将当前进程有序插入到 blocked\_list 中，然后将当前进程阻塞。

最后恢复中断

在计时器中断处理程序中，执行 handle\_blocked\_threads() 函数。

该函数将遍历 blocked\_list ，把已经休息够的线程移出队列并解除其阻塞。

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

>> the timer interrupt handler?

增设的blocked\_list 队列可以减少每个时钟中断处理程序执行的时间：由于维护了优先队列，在遍历时若发现当前遍历进程的阻塞结束时间已经大于当前时间，则可直接结束遍历。

---- SYNCHRONIZATION ----

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

>> timer\_sleep() simultaneously?

对列表 blocked\_list 的插入和移出操作都在屏蔽中断后进行以避免竞争条件。

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

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

在执行timer\_sleep()时，中断被屏蔽，避免了竞争条件。

---- RATIONALE ----

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

>> another design you considered?

最初直接的想法是要使得当前进程被暂停并在tick个时钟周期后重新启动执行，则首先得记录下tick，然后将当前进程阻塞，将资源让出。并且在每个时钟周期检查是否要恢复进程。

一开始设计时是直接在每个时间中断时检查所有进程的 blocked\_time 以判断是否要恢复其继续执行。后来考虑到每次遍历所有进程是一种低效的做法，于是参考 ready\_list 维护了一个有序列表 blocked\_list，用以记录被阻塞的进程。这样在每个时间中断时就无需遍历所有进程而只需要遍历blocked\_list，当遍历到第一个无需唤醒的进程时即可结束遍历，提高了执行效率。

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.

在struct lock中添加：

/\*优先级捐赠部分加的\*/

struct list\_elem elem; // 优先级捐赠的线程的队列

int max\_priority; // 请求该锁的线程中, 优先级最高的

struct thread \*holder; /\* 占有这个锁的线程(for debugging)\*/

在struct thread中添加：

/\* 优先级捐赠这部分\*/

int base\_priority; // 起始优先级

struct list locks; // 当前线程占有的锁

struct lock\* lock\_waitings; // 当前线程请求的锁, 一个锁

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

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

>> .png file.)

线程A，优先级31，拥有lock\_a

线程B，优先级32，拥有lock\_b，想要获取lock\_a

线程C，优先级33，想要获取lock\_b

1. 初始状态

|  |  |
| --- | --- |
| Thread A | |
| 变量 | 值 |
| base\_priority | 31 |
| priority | 31 |
| locks | lock\_a |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread B | |
| 变量 | 值 |
| base\_priority | 32 |
| priority | 32 |
| locks | lock\_b |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread C | |
| 变量 | 值 |
| base\_priority | 33 |
| priority | 33 |
| locks | NULL |
| lock\_waitings | NULL |

2. 线程B获取lock\_a

|  |  |
| --- | --- |
| Thread A | |
| 变量 | 值 |
| base\_priority | 31 |
| priority | 32 |
| locks | lock\_a( max\_priority=32) |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread B | |
| 变量 | 值 |
| base\_priority | 32 |
| priority | 32 |
| locks | lock\_b |
| lock\_waitings | &lock\_a |

|  |  |
| --- | --- |
| Thread C | |
| 变量 | 值 |
| base\_priority | 33 |
| priority | 33 |
| locks | NULL |
| lock\_waitings | NULL |

3. 线程C获取lock\_b

(1)

|  |  |
| --- | --- |
| Thread A | |
| 变量 | 值 |
| base\_priority | 31 |
| priority | 32 |
| locks | lock\_a( max\_priority=32) |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread B | |
| 变量 | 值 |
| base\_priority | 32 |
| priority | 33 |
| locks | lock\_b( max\_priority=33) |
| lock\_waitings | &lock\_a |

|  |  |
| --- | --- |
| Thread C | |
| 变量 | 值 |
| base\_priority | 33 |
| priority | 33 |
| locks | NULL |
| lock\_waitings | &lock\_b |

3. 线程C获取lock\_b

(2)

|  |  |
| --- | --- |
| Thread A | |
| 变量 | 值 |
| base\_priority | 31 |
| priority | 33 |
| locks | lock\_a( max\_priority=32) |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread B | |
| 变量 | 值 |
| base\_priority | 32 |
| priority | 33 |
| locks | lock\_b( max\_priority=33) |
| lock\_waitings | &lock\_a |

|  |  |
| --- | --- |
| Thread C | |
| 变量 | 值 |
| base\_priority | 33 |
| priority | 33 |
| locks | NULL |
| lock\_waitings | &lock\_b |

4. 线程A释放lock\_a

|  |  |
| --- | --- |
| Thread A | |
| 变量 | 值 |
| base\_priority | 31 |
| priority | 31 |
| locks | NULL |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread B | |
| 变量 | 值 |
| base\_priority | 32 |
| priority | 33 |
| locks | lock\_b( max\_priority=33)  & lock\_a( max\_priority=32) |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread C | |
| 变量 | 值 |
| base\_priority | 33 |
| priority | 33 |
| locks | NULL |
| lock\_waitings | &lock\_b |

5. 线程B释放lock\_b

|  |  |
| --- | --- |
| Thread A | |
| 变量 | 值 |
| base\_priority | 31 |
| priority | 31 |
| locks | NULL |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread B | |
| 变量 | 值 |
| base\_priority | 32 |
| priority | 32 |
| locks | lock\_a( max\_priority=32) |
| lock\_waitings | NULL |

|  |  |
| --- | --- |
| Thread C | |
| 变量 | 值 |
| base\_priority | 33 |
| priority | 33 |
| locks | lock\_b( max\_priority=33) |
| lock\_waitings | NULL |

---- ALGORITHMS ----

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

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

当我们需要将一个线程放入就绪队列时，原本使用list\_push\_back，将其改为list\_insert\_ordered。

这样保证就绪队列为一个优先级队列，每次率先唤醒的都是优先级最高的线程。

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

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

事件序列：

1. 判断lock->holder是否为null

2. 若lock->holder不为null，当满足条件：当前线程的优先级>锁的优先级时，执行3，否则执行4

3. 迭代地捐赠优先级

（1）将lock->max\_priority(请求该锁的最高优先级)置为当前线程的优先级

（2）将当前线程的优先级捐赠给持有该锁的线程

（3）如果当前锁还被别的锁控制着，将临时存储锁的临时变量置为新的锁，跳转至2

4. 优先级捐赠结束。执行P操作，请求锁，阻塞然后被唤醒，然后获得锁

解决嵌套捐赠：

按上述过程即可解决此问题，当持有该锁的线程经过优先级捐赠后，若其还在等待其他锁，

则进入持有新锁的线程，并对其进行优先级捐赠，直至所有持锁线程均与第一个请求锁的线程具有相同的优先级为止。

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

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

事件序列：

1. 调用list\_remove将此锁线程移除

2. 调用thread\_update\_priority处理线程优先级的改变

（1）如果这个线程还有锁，就先获取这个线程拥有锁的最大优先级（可能被更高级线程捐赠），

如果这个优先级比base\_priority大的话更新的应该是被捐赠的优先级。

（2）如果这个线程没有锁，就将此线程优先级置为base\_priority（起始优先级）

3. 将此锁的holder置为NULL

4. 释放信号量, 信号量+1

---- 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?

问题：

当我们调用thread\_set\_priority进行线程优先级设置的同时，此线程若为锁的持有者，

则可能被优先级捐赠，从而改变优先级。如果这两个过程的顺序不确定，则无法确定最终的优先级为哪个。

解决方法：

在我们调用thread\_set\_priority时，首先屏蔽中断，保障设置优先级的过程不会被打断

使用锁：

我们的实现中没有使用锁来解决此问题。但我们可以考虑给捐赠者和此线程加一个所结构来避免此问题。

但是使用此方法同时会改变线程持有锁的列表，所以需要注意可能会导致的死锁问题。

---- RATIONALE ----

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

>> another design you considered?

我们使用list\_insert\_ordered而不是list\_push\_back来确保线程的等待队列为一个优先级队列。

这样可以避免每次调度时进行优先级排序从而浪费大量时间。

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.

更改的struct 全局或静态变量 typeof 声明：

**一、thread.h中**

1. 更新了struct thread{

int nice; //每个线程都要有nice值来决定对其他线程CPU影响

int64\_t recent\_cpu;//线程最近使用的CPU时间的估计值，与nice值共同影响线程优先级

}

1. 增加了方法

bool cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux);

//该方法用在更新全部线程优先级的函数与需要更新线程优先级的函数中，实现线程就绪队列按照优先级大小排序

**二、thread.c中**

1、定义了浮点类型int64\_t fixed\_t,通过将整型左移14位得到17.14格式的浮点型来模拟浮点数运算，以便实现之后对于recent\_cpu与load\_avg两个实数的计算。

typedef int64\_t fixed\_t;

#define SHIFT\_AMOUNT 14

#define INT2FLOAT(n) ((fixed\_t)(n << SHIFT\_AMOUNT))

#define FLOAT2INTPART(x) (x >> SHIFT\_AMOUNT)

#define FLOAT2INTNEAR(x) (x >= 0 ? ((x + (1 << (SHIFT\_AMOUNT - 1))) >> SHIFT\_AMOUNT) : ((x - (1 << SHIFT\_AMOUNT)) >> SHIFT\_AMOUNT))

#define FLOATADDFLOAT(x, y) (x + y)

#define FLOATSUBFLOAT(x, y) (x - y)

#define FLOATADDINT(x, n) (x + (n << SHIFT\_AMOUNT))

#define FLOATSUBINT(x, n) (x - (n << SHIFT\_AMOUNT))

#define FLOATMULFLOAT(x, y) ((((int64\_t)x) \* y) >> SHIFT\_AMOUNT)

#define FLOATMULINT(x, n) (x \* n)

#define FLOATDIVFLOAT(x, y) ((((int64\_t)x) << SHIFT\_AMOUNT) / y)

#define FLOATDIVINT(x, n) (x / n)

1. 声明int64\_t load\_avg 作为系统平均负载，它估计过去一分钟内准备运行的线程的平均数量。

---- 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 | | | priority | | | thread to run |
| A | B | C | A | B | C |
| 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 |
| 28 | 16 | 8 | 4 | 59 | 59 | 58 | B |
| 32 | 16 | 12 | 4 | 59 | 58 | 58 | A |
| 36 | 20 | 12 | 4 | 58 | 58 | 58 | 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?

存在歧义——例如上表timer ticks为28时此时A的优先级降低，优先级最高的有B、C此时应该选择哪个进程运行，对此根据文档对同优先级的线程采用循环调度（FCFS），实际上也与调度程序相符合。

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

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

中断切换上下文要保证原子操作就是要进行关闭外中断，而如果原子操作过长导致关闭外中断时间过长就会影响到操作系统的性能，同时在中断上下文的优先级是高于进程上下文的，无论对于用户进程还是内核线程，一般情况下如果中断发生，都会造成中断上下文抢占当前进程上下文执行的，一般都会把调度放在中断外进行，即内核中断中不进行调度。

---- 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?

根据指导手册一步一步实现有关计算nice值、recent\_cpu值、load\_avg值与priority值的函数方法以及细节，另外实现pintos模拟浮点数的运算。

不更改timer.c中的timer\_interrupt()函数而将判断是否是mlfqs调度放到thread\_tick()中

优点：

1、对于nice为负值的线程，其recent\_cpu也可能为负值，如果为负值，优先级更新会超过PRI\_MAX(63)，调度会产生错误，所以在更新优先级函数中对优先级溢出部分做了限定。

/\* 更新单个线程的优先级 \*/

void

renew\_priority(struct thread\* t){

/\* priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2) \*/

t->priority = PRI\_MAX - FLOAT2INTPART(FLOATDIVINT(t->recent\_cpu, 4)) - (t->nice \* 2);

if(t->priority > PRI\_MAX)

t->priority = PRI\_MAX;

else if(t->priority < PRI\_MIN)

t->priority = PRI\_MIN;

}

>> 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?

通过位运算模拟浮点数运算，同时声明宏与一个int64\_t数据类型来完成浮点数运算

typedef int64\_t fixed\_t;

#define SHIFT\_AMOUNT 14

#define INT2FLOAT(n) ((fixed\_t)(n << SHIFT\_AMOUNT))

#define FLOAT2INTPART(x) (x >> SHIFT\_AMOUNT)

#define FLOAT2INTNEAR(x) (x >= 0 ? ((x + (1 << (SHIFT\_AMOUNT - 1))) >> SHIFT\_AMOUNT) : ((x - (1 << SHIFT\_AMOUNT)) >> SHIFT\_AMOUNT))

#define FLOATADDFLOAT(x, y) (x + y)

#define FLOATSUBFLOAT(x, y) (x - y)

#define FLOATADDINT(x, n) (x + (n << SHIFT\_AMOUNT))

#define FLOATSUBINT(x, n) (x - (n << SHIFT\_AMOUNT))

#define FLOATMULFLOAT(x, y) ((((int64\_t)x) \* y) >> SHIFT\_AMOUNT)

#define FLOATMULINT(x, n) (x \* n)

#define FLOATDIVFLOAT(x, y) ((((int64\_t)x) << SHIFT\_AMOUNT) / y)

#define FLOATDIVINT(x, n) (x / n)

首先定义宏来进行简化指令，方便代码书写，再者由于只涉及加减与位运算，篇幅较小，满足宏定义的要求，并且一定程度上能提高程序运行效率。