Project 1 实验文档

Project 1 实验文档

- 团队
  - 基本信息
  - 每位组员的主要工作内容
  - Git 相关
  - 参考资料
- 实验要求
- 需求分析
  - 问题1：
  - 问题2:
  - 问题3：
- 设计思路
  - 问题1：
  - 问题2：
  - 问题3：
- 重难点讲解
  - 问题1：
  - 问题2:
  - 问题3：
- 用户手册
- 测试报告
- 各成员的心得体会
  - Student1 刘济为
  - Student2 欧阳思问
  - Student3 邓浩
  - Student4 朱穆清
- 其他你认为有必要的内容 (Optional)

Project 1 Design Document

- QUESTION 1: ALARM CLOCK
  - DATA STRUCTURES

        ALGORITHMS
        SYNCHRONIZATION
        RATIONALE
    QUESTION 2: PRIORITY SCHEDULING
        DATA STRUCTURES
        ALGORITHMS
        SYNCHRONIZATION
        RATIONALE
    QUESTION 3: ADVANCED SCHEDULER
        DATA STRUCTURES
        ALGORITHMS
        RATIONALE

团队

基本信息

姓名	学号
欧阳思问	19373432
邓浩	18374046
刘济为	19373720
朱穆清	18375123

每位组员的主要工作内容

Git 相关

Pintos 的 Git 项目地址

参考资料

1. 《操作系统概念》机械工业出版社·第九版

实验要求

需求分析

问题1：

PintOS中 `timer_sleep()` 函数原始的实现方式为使用一个 `while` 循环，在 `ticks` 个时间片内不断调用 `thread_yeild()` 函数，但是这种实现机制可能导致”忙等待“。为了消除这种”忙等待“，使得线程睡眠时不占用CPU，需要让线程在睡眠时间中进入阻塞态，并在睡眠时间结束后唤醒，放入就绪队列。

问题2:

Pintos中线程在就绪队列时的加载方式为：直接将线程加载到队末，未实现让高优先级的线程先调度。同时若锁被低线程所持有，可能会导致一些不顺应高优先级先调度的情况发生。为了解决这两个优先级的问题，需要实现：线程在就绪队列中的优先级调度以及线程在遇到锁被低优先级线程持有时应捐献自己的优先级的行为。

优先级调度：Pintos的原始实现中，线程的调度采用的是先来先服务策略。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。

优先级捐赠：如果线程H拥有较高的优先级，线程M拥有中等的优先级，线程L拥有较低的优先级。此时若线程H正在等待L持有的锁, 且M一直在就绪队列之中，那么线程H将永远无法获得CPU。

问题3：

按照题目说的我们需要减少系统的平均响应时间。根据实验文档的描述，大概指的是这个BSD调度程序维持了64个队列，分别对应PRI_MIN到PRI_MAX，然后需要通过一些公式来计算出每个线程当前的优先级，方便系统调度的时候从高优先级队列开始选择线程执行，而这里的优先级还会随着操作系统的运转数据而动态的改变。

同时priority的计算过程大概一定是会出现小数的，但不幸的是，Pintos并没有定义浮点数的计算方式，这也就意味着，在解决这个多级反馈调度程序之前，我们得先花时间想一个办法实现浮点数的计算，然后有了这个浮点数计算的工具，来完成这个调度程序的设计实现。

设计思路

问题1：

需求的分析中我们已经知道，我们需要将线程处于睡眠时间时使其处于阻塞状态，并在睡眠时间结束后唤醒放入就绪队列。而PintOS中已经实现了 `thread_block()` 函数和 `thread_unblock()` 函数，因此我们只需要实现如何检测线程已经睡眠了多长时间。

我们的设计思路是在 `thread` 结构体中新增 `int_64t ticks_blocked` 属性，用于记录需要睡眠的时间。系统每过一个 `tick` 时间就会调用 `timer_interrupt()` 函数，因此我们只需要在这个函数中更新线程的 `ticks_blocked` 属性即可。

我们先实现 `blocked_thread_check()` 函数，这个函数的功能是将处于 `THREAD_BLOCKED` 状态线程的 `ticks_blocked` 值减一，若减完等于0则，调用 `thread_unblock()` 函数唤醒该线程。

接着我们在 `timer_interrupt()` 函数中调用 `thread_foreach（blocked_thread_check, NULL）` 遍历 `all_list` 线程链表，对所有线程进行检查。

至此，功能基本实现，下面完善一些细节。

1. 在 `thread_create()` 初始化 `ticks_blocked` 属性为0
2. `timer_sleep()` 函数中，先判断参数 `ticks` 是否非正数，若不是正数则直接返回

问题2：

优先级调度的问题分为两个小部分：
1.线程优先级队列的实现
2.优先级捐赠的实现

首先考虑实现线程优先级调度的部分。

在PintOS的原始实现中，线程的就绪队列、信号量等待队列、条件变量等待队列都是FIFO队列，每次调度器都会直接去取处于队首的线程使其运行或解除阻塞。因为线程的优先级定义方式是 `int priority`，我们需要让这个队列根据每个线程的 `priority` 值作为标准，维护一个优先级队列。

最方便的方式就是在每一次将线程放入就绪队列中时确保这个队列是优先级队列。观察以后可以发现：有将线程加入就绪队列这个操作的函数有： `thread_unblock()`，`init_thread()` 和 `thread_yield()` 。他们在原始实现中对于把线程放入队列的操作都是通过 `list_push_back()` 实现的，即简单的把线程放到就绪队列的队尾。我们只需要利用系统提供的另一个函数 `list_insert_ordered`，它可以在将元素按值有序地插入队列，只需额外制定一个比较函数即可。

因此我们这部分的工作就是，首先实现一个按照线程优先级（从大到小）的比较函数 `thread_cmp_priority`，并作为参数配合 `list_insert_ordered` 一起使用，替换系统原本不含优先级排序的 `list_push_back()` （分别在解除阻塞、新建线程和yield函数中）。

特别的，针对 `priority_change` 测试点，我们在创建一个线程的时候，即 `thread_create()` 函数中，如果新线程的优先级高于当前线程，就先运行创建的线程，即把当前线程加入就绪队列（执行 `thread_yield()` ）。如果更改了某个线程的优先级，即 `thread_set_priority` 函数，也直接将其加入就绪队列，队列就保证了执行的正确顺序。

对于2：程序的设计思路为引入新的线程属性，base_priority与locks、lock_waiting分别记录当下线程原优先级、当前所持有锁以及正在等待的锁。同时在lock这个结构体中引入：elem以方便将该结构体加入到list当中，max_priority以记录在请求这个锁的线程中最大的优先级。同时当一个线程获取到一个锁后就记录下并根据锁的最大优先级变更自己的优先级。这样，只需要每次线程在请求锁的时候去递归地捐献优先级，就可以保证持有锁的线程一定是先运行了。

程序还会考察多个锁的情况，这样就需要在线程所获取的锁的list中实现锁根据最大优先级排序，然后线程就能获取持有的多个锁中优先级最大的。thread_hold_lock()与thread_donate_priority()两个看似有重合的部分，功能一样，但是实则不同，hold_lock是在一个线程自身获取锁时执行，donate_priority是催促持有锁的线程时执行。两者都完成了一个功能：将所有锁的最大优先级与线程自身的优先级比较，然后yield重新分配一次调度顺序。

初次之外，还需要实现对于信号量和条件变量的优先级队列实现。条件变量和信号量各有一个waiters列表。区别在于条件变量的waiters的成员是semaphore_elem，而信号量的waiters的成员是thread，可以看出两者的关系。具体实现如下：cond_signal条件满足时，找出优先级最大的semaphore来进行sema_up操作，而semaphore的优先级取决于等待它的线程中最大的优先级。由此，就可以实现最大优先级的线程最先获得semaphore的资源，从而更先执行。

到此，问题2的实现思路就解决了。

问题3：

首先，想办法解决浮点数计算的问题。

根据CS140 Pintos Project给出的方案，直接使用17.14定点数字表示来解决浮点数计算的问题。

这个方法详细来说，就是将整数的最右边几位视为代表小数，而在这里，我们决定使用带符号整数的最低14位作为指定的小数位，这样一个整数x在这里就将表示实数x/（2>>14）.

这种格式的数字最多表示：（2>>31-1）/(2>>14)≈131071.999

但这里存在一个问题，乘法可能会导致溢出问题，所以在定义中我选择使用int64_t来存储17.14表示法的整数形式的浮点数。

在这之后，就是如何实现题目所需要的多级反馈调节程序（MLFQS）

大概思路是这样的（按照文档里规定的各个量的更新频率）：由于我们需要满足这个操作系统状态的动态改变，我们需要固定一段时间就去更新一下线程的优先级，在timer_interrupt()里固定每1秒来更新一次系统的load_avg以及所有的线程的recent_cpu（在系统的timer.h中我们可以发现定义了TIMER_FREQ就是100ticks，即1秒，所以用这个就OK了），并每4次timer_ticks来更新一次线程的优先级，每一个timer_tick运行的线程的recent_cpu+1。

根据提示，MLFQS与正常的优先级调度可能存在部分冲突，所以我们需要增加一个判断，即需要的thread_mlfqs这个布尔变量为true，即系统需要我们进行高级调度的时候，我们再进行高级调度所需要的工作。

按照题目所说实现64个优先级队列显然有点逆天，题目的意思应该是系统通过这个新算法计算出的priority来选取高优先度的线程先运行，本质上应该于优先级调度没有太大区别。所以我们直接使用之前保留的优先级调度也可完成所需要的目标功能。

重难点讲解

问题1：

问题1难点就在于检查线程阻塞的时间，具体实现在设计思路中已经讲解。

问题2:

问题2的难点在于实现优先级队列的调度时，如何去协调请求锁与持有锁的线程的优先级的更替，以及持有多个锁、锁套锁时应该如何去解决。

问题3：

问题3的重难点主要是对于浮点数计算的实现，而这可以根据官方文档所给出的17.14定点数字表示法进行解决。随后新建一个fixed_point.h头文件来存储相关操作，并在相关函数里进行引入即可正常使用。

用户手册

测试报告

各成员的心得体会

Student1 刘济为

本次实验我主要负责线程的优先级调度。一开始对Pintos比较畏难，因为对于里面的函数、文件结构等不是很清楚，但通过后续一系列的学习，查找相关概念，逐渐厘清了对线程的思路与概念。同时将以往线程、函数、栈、进程等区别与关系都弄清楚了。感觉通过直面P1最难的部分，我获得的成长也是最多的。

Student2 欧阳思问

本次实验主要是进行有关Pintos线程的优先级以及调度等方面的实践实验，相较于课上的直接性的知识学习，进行本次实验让我能更好的理解在Linux系统中每一个线程的每一部分功能究竟是怎样进行运行的。同时本次实验的高难度也给我带来了不小的困难。

Student3 邓浩

通过本次实验，熟悉了操作系统关于线程生命周期和线程调度方面的知识。对书上的文字有了更深的理解。完成本次有难度的实验，也有不少成就感。另一方面，对PintOS代码编写者深感佩服。代码看着非常舒服，质量非常高，值得我们学习。

Student4 朱穆清

本次实验是这门课程的第一次实验，我熟悉了PintOS的运行环境，代码结构和系统对于线程优先级的处理方式。在做实验的同时也复习了课上学习的cpu调度方式和同步互斥的概念，加深了自己的理解。此外，给下一次更难的实验提供了宝贵经验。

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.
>
> 答: `int64_t ticks_blocked;/* 记录线程睡眠时间 */`

ALGORITHMS

> A2: Briefly describe what happens in a call to timer_sleep(), including the effects of the timer interrupt handler.
>
> 答：将线程设置为阻塞状态。在时间中断函数中检查线程睡眠时间。

> A3: What steps are taken to minimize the amount of time spent in the timer interrupt handler?
>
> 答：在 `blocked_thread_check()` 函数中先判断线程是否为阻塞状态，若为阻塞状态且 `ticks_blocked` 大于0才继续操作。

SYNCHRONIZATION

> A4: How are race conditions avoided when multiple threads call timer_sleep() simultaneously?
>
> 答： `ASSERT(intr_get_level() == INTR_ON)` 这句断言确保该函数可在这句后中断，同时函数没有调用全局/静态变量，不会造成冲突。

> A5: How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?
>
> 答：通过 `enum intr_level old_level = intr_disable();` 和 `intr_set_level(old_level);` 两行代码确保这两行之间的代码运行时不会被中断。

RATIONALE

> A6: Why did you choose this design? In what ways is it superior to another design you considered?
>
> 答：暂时没考虑其他实现

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 lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging).
*/
    struct semaphore semaphore; /* Binary semaphore controlling access.
*/
    /*m2：加入elem以便于将锁能放进list内部，max_priority是多个thread同时请求该锁
时
    他们当中优先级最大的，以便于赋值给被捐赠的线程。*/
    struct list_elem elem;      /* List element for priority donation.
*/
    int max_priority;          /* Max priority among the threads
acquiring the lock. */
  };

struct thread
  {
    int nice;                           /* Niceness. */
    fixed_t recent_cpu;                 /* Recent CPU. */
    /* 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. */
    /* Record the time the thread has been blocked. */
    int64_t ticks_blocked;

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

    /*m2: base priority用于记录原来的priority，priority代表现在的，locks代表这
个thread有什么锁，lock_waiting代表它正在等待什么锁*/
    unsigned magic;                     /* Detects stack overflow. */
    int base_priority;                  /* Base priority. */
    struct list locks;                  /* Locks that the thread is
holding. */
    struct lock *lock_waiting;          /* The lock that the thread is
waiting for. */
  };
```

> B2: Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation. (Alternately, paste an image.)
>
> The explaination is written as comments in the code above.

ALGORITHMS

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

> 对于等锁：通过将优先级捐赠给低优先级的线程，然后重新再加入优先级就绪队列。对于等信号量、条件变量：通过实现各自等待变量的优先级队列，取waiters中优先级最高的线程唤醒执行。

> B4: Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?
>
> 如果获取的锁正在被其他线程持有，则判断是否该锁的holder也有正在等待的锁，若持有且当前线程的优先级比该锁的最大优先级还大，那么则更新该锁的最大优先级，并捐赠给它的holder，不断重复这个过程，实现递归捐赠。

> B5: Describe the sequence of events when lock_release() is called on a lock that a higher-priority thread is waiting for.
>
> 当持有锁的线程释放锁之后，它的优先级会首先看还有没有其他持有的锁，并从中挑选一个最大的，若没有，则变为原来的优先级，若在持有锁的过程中被更改了优先级，则变为被更改的优先级。

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?
>
> 在创建一个线程的时候，如果线程高于当前线程就先执行创建的线程。可以在设置一个线程优先级要立即重新考虑所有线程执行顺序， 重新安排执行顺序。不必通过锁来实现。

RATIONALE

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

> 选择这个设计是因为优先级队列的实现更容易、理解上更直观，同时因为可以直接调用Pintos内置的list函数，比较轻松。对于其他实现方式，例如采用每次取线程、信号量前排序，这样效率低，而且与Pintos内部的设计不符，因为Pintos的list_elem这个结构体被大规模使用，不能大改设计框架。

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.

[NEW] fixed_point.h 用于处理浮点数运算

in thread.c

> [New] int nice:题目所需的Nice值
>
> [New] fixed_t load_avg :题目所需的load_avg值.
>
> [New] fixed_t recent_cpu:题目所需的recent_cpu值
>
> [Changed] init_thread: 初始化了nice值与recent_cpu值（均为0）
>
> [New] void thread_increase_recent_cpu_by_one (void)：按照题目需要递增更新recent_cpu
>
> [New] void thread_update_load_avg_and_recent_cpu (void)：按照题目需要更新load_avg 与 recent_cpu
>
> [New] void thread_mlfqs_update_priority (struct thread *t): 按照题目的需要进行线程优先级的更新
>
> [Changed] void thread_set_nice (int nice):按照题目要求实现设置nice值功能

> [Changed] int thread_get_nice (void) :按照题目要求实现获取nice值功能
>
> [Changed] int thread_get_load_avg (void): 按照题目要求实现获取load_avg*100的功能
>
> [Changed] int thread_get_recent_cpu (void): 按照题目要求实现获取recent_cpu*100的功能。

in timer.c

> [Changed] static void timer_interrupt (struct intr_frame *args UNUSED): 按照题目需要来实现对于load_avg,recent_cpu,priority的更新

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	1	2	63	61	59	A
4	4	1	2	62	61	59	A
8	8	1	2	61	61	59	B
12	8	5	2	61	60	59	A
16	12	5	2	60	60	59	B
20	12	9	2	60	59	59	A
24	16	9	2	59	59	59	C
24	16	9	6	59	59	58	B
32	16	13	6	59	58	58	A
36	20	13	6	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?
>
> 答：可以发现，当两个线程按照公式计算出priority值相等的时候，单单从现在的条件里我们是无法判断那个线程先运行的。
>
> 在我们的代码中，我们优先级调度所实现的前提是将就绪队列变为优先级队列，也就是每一次放入队列中的时候，会进行一个排序，而这个排序在优先级程度相同的时候是不会进行位置调换的，即按照位置顺序，先插入的线程会先进行运行。

> C4: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?
>
> 答：我们在进行实现的时候，进行了一个断言：ASSERT（intr_context()）即断言这个中断为外部中断。因为内核中断的优先度更高所以我们决定不再内核中进行调度，即做这样一个断言。

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?

对于第三部分的实验，其实实现难度并不很高，因为官方文档基本上已经把实现方式解决完了。

我们在实现对于浮点数等功能均是按照文档里的提示（即17.14定点表示法）来实现的

我们代码的优点是：

1. 通过使用int64_t来实现浮点运算，提高了实现运算的精度（有效的降低了乘法等溢出的反面效果）。
2. 通过使用recent_cpu,load_avg,等量来进行所有线程的优先级更新增加了调度的合理性。

缺点则是在计算的过程中会占据大量的资源，以及在相同优先级的线程进行运行时的顺序具体实现并没有落实。

如果时间更多可能会更注重于对上述情况的线程运行调度进行优化修改。