Pintos系统

**Threads实验报告**

小组成员：

杨珺瑶 安全1601 16281210

王玮誉 安全1601 16281206

宋旻 安全1601 16281078

袁少随 安全1601 16281054

黄春浦 安全1601 16281254

马楠 安全1601 16281200

目录

[一、Pintos介绍与安装 2](#_Toc8397097)

[1、Pintos简介 2](#_Toc8397098)

[2、Pintos安装 2](#_Toc8397099)

[二、pintos基本概念理解 3](#_Toc8397100)

[三、实验任务 5](#_Toc8397101)

[Task1: Alarm Clock 5](#_Toc8397102)

[1.1、原函数分析 6](#_Toc8397103)

[1.2、函数改进 7](#_Toc8397104)

[1.3、新函数实现 8](#_Toc8397105)

[1.4、测试结果 11](#_Toc8397106)

[Task2: Priority Scheduling 11](#_Toc8397107)

[2.1、pass alarm\_priority 12](#_Toc8397108)

[2.2、pass priority-change、priority-preempt 13](#_Toc8397109)

[2.3、pass priority-donate-one、priority-donate-multiple、priority-donate-multiple2、priority-donate-nest、priority-donate-sema、priority-donate-lower、priority-sema、priority-condvar、priority-donate-chain 14](#_Toc8397110)

[2.4、实验结果 46](#_Toc8397111)

[Task3：Advanced Scheduler 46](#_Toc8397112)

[3.1、原理 46](#_Toc8397113)

[3.2、函数实现 47](#_Toc8397114)

[3.3、测试结果 52](#_Toc8397115)

[四、参考文献 53](#_Toc8397116)

# 一、Pintos介绍与安装

## 1、Pintos简介

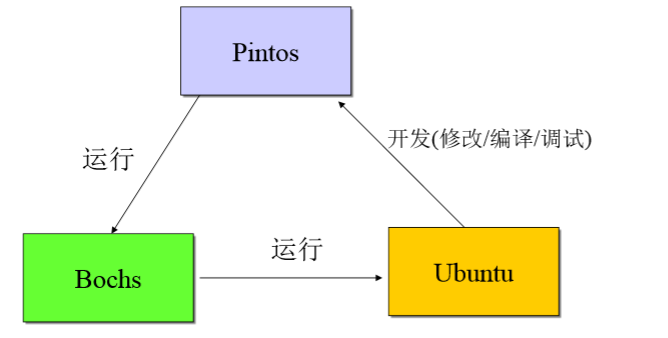
Pintos是Standford大学为操作系统课程专门开发的一个 基于80x86架构的简单操作系统框架(A simple operating system framework)。

其特点如下：

* 支持：内核线程；装载和运行用户程序；文件系统。
* 但这些功能都只是以一种非常简单的方式实现的。
* 我们能做的工作：

1. 从上述三个方面完善该系统
2. 添加虚拟内存的实现

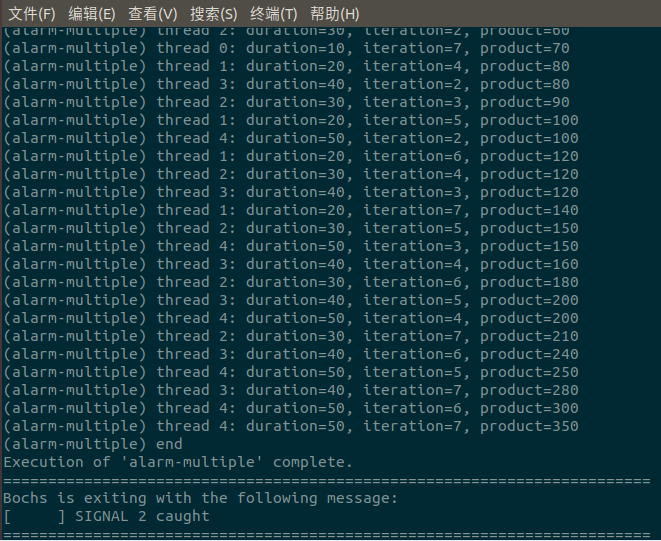
理论上来讲，Pintos是可以运行在IBM兼容PC上的， 但实际中为了开发和调试的方便，我们常让Pintos 运行在虚拟机上(如：Bochs/QEMU/VMWare)。同时因为Pintos是在UNIX下用C语言编写出来的， 因此其开发环境也应为类UNIX系统。当前使用较多 的是Linux系统，Linux发行版可自由选择。Pintos, Bochs和 Ununtu关系如下：



## 2、Pintos安装

具体安装方法见安装教程：<http://www.doc88.com/p-3874510018737.html>

安装成功截图如下：（执行pintos -v -- run alarm-multipe的结果）



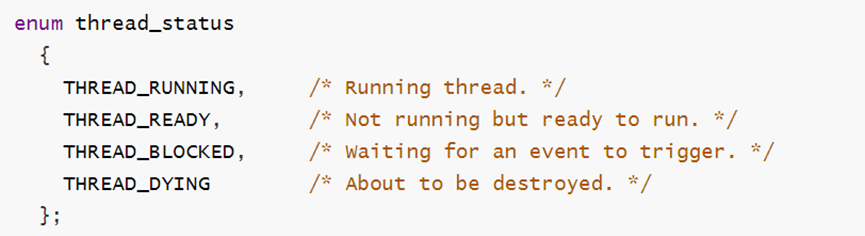
# 二、pintos基本概念理解

本次实验至少需要阅读thread.h，tread.c，interrup.h，time.c这四个文件。

pintos在thread.h中定义了一个结构体struct thread，这个结构体存放了有关进程的基本信息。



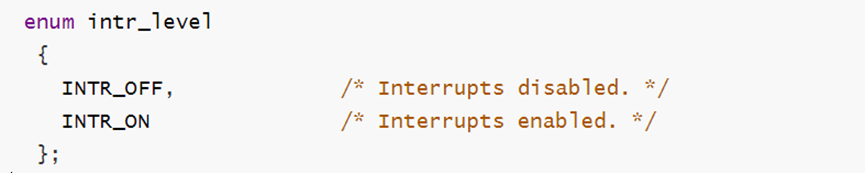
其中enum thread\_status这个枚举类型的变量表示这个线程现在所处的状态。



还有一个重要的概念是中断。中断分两种：

1. IO设备向CPU发出的中断的信息；
2. CPU决定切换到另一个进程时（轮换时间片）发出的指令

我们现在处理第二种。pintos的中断在interrupt.h和interrupt.c之中。其中这个枚举类型intr\_lverl如下：（intr\_level中，intr\_off表示关中断，on表示开中断，执行原子级别操作的时候，中断必须是关着的。）



pintos是以ticks作为基本时间单位的，每秒有TIMER\_FREQ个ticks：

pintos默认每一个ticks调用一次时间中断。换句话说，每一个线程最多可以占据CPU一个ticks的时长，之后就必须放手。

# 三、实验任务

## Task1: Alarm Clock

本实验要求重新实现devices/timer.c中的timer\_sleep()函数。

基本操作函数包括：

以下函数均可以在thread.c中找到：

* thread\_current()获取当前当前的线程的指针。
* thread\_foreach(thread\_action\_func \*func, void \*aux) 遍历当前ready queue中的所有线程，并且对于每一个线程执行一次func操作。注意到这里的func是一个任意给定函数的指针，参数aux则是你想要传给这个函数的参数。pintos中，所有ready的线程被保存在一个链表中。这个函数做得不过是遍历了一遍链表而已。注意这个函数只能在中断关闭的时候调用。
* thread\_block()和thread\_unblock(thread \*t)。 这是一对儿函数，区别在于第一个函数的作用是把当前占用cpu的线程阻塞掉（就是放到waiting里面），而第二个函数作用是将已经被阻塞掉的进程t唤醒到ready队列中。
* timer\_interrupt (struct intr\_frame \*args UNUSED)这个函数在timer.c中，pintos在每次时间中断时（即每一个时间单位（ticks））调用一次这个函数。
* intr\_disable () 这个函数在interrupt.c中，作用是返回关中断，然后返回中断关闭前的状态。（状态为INTR\_OFF,INTR\_ON 这两种 ）

其中代码中普遍用到断言assert，对其进行一个解释：

assert宏的原型定义在assert.h中，其作用是如果它的条件返回错误，则终止程序执行.

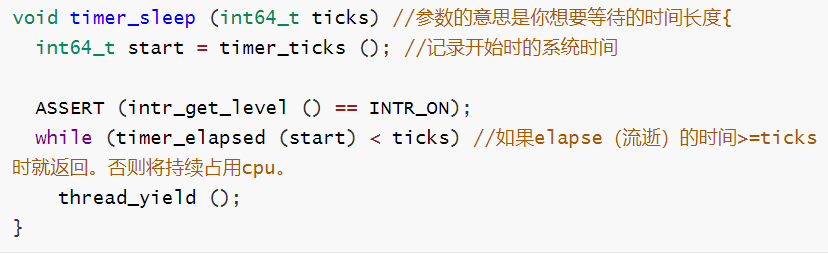
1 #include "assert.h"

2 void assert( int expression );

assert的作用是现计算表达式 expression ，如果其值为假（即为0），那么它先向stderr打印一条出错信息,然后通过调用 abort 来终止程序运行。 使用assert的缺点是，频繁的调用会极大的影响程序的性能，增加额外的开销。

##### 1.1、原函数分析

Pintos系统中原有的timer\_sleep()函数如下:



* **代码注释：**

yield()的作用是让步。它能让当前线程由“运行状态”进入到“就绪状态”，从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保证在当前线程调用yield()之后，其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！ 若此线程处于ready状态，则该线程处于ready队列中，由时间片轮询法可知，此队列的每一个线程都有一定的机会被CPU再次轮询到，即使此时该想要睡眠的线程没有到达时间，也会再次被CPU轮询到，就出现所谓上面的while形式，当轮询到此线程的时候，首先应该判断该线程是否已经睡眠够了，若不够，则立马放弃CPU的使用，继续进入ready队列中。

* **函数功能**

功能是实现让某个线程睡眠ticks时间，即让该线程暂时放弃CPU。

* **原函数缺点**

该函数现有的实现虽然是正确的，但所采用的是“忙等”技术，通过不停地循环调用thread\_yield()并查询是否已经达到等待时间。

若线程由“运行状态”进入到“就绪状态”，timer\_sleep函数使用的方法（忙等待）将利用一个while循环不断地请求CPU来判断是否经过了足够的时间长度， 而通常cpu在一个ticks时间内可以处理10000次这样的循环，而timer\_elapsed()函数只会在ticks＋1时更新一次。因此，在该线程睡眠期间，每过一段时间都要被CPU询问一次，让后又立马放弃使用权，如此反复进行，就是在做无用功。

##### 1.2、函数改进

**1.2.1 改进方案**

改进这个函数，使得它每一个ticks才检查一次时间，而不是每一个ticks检查一万次。设计思路是：在timer\_sleep（）函数中让该进程暂时阻塞（调用thread\_block()），然后过了需要sleep的ticks个时间段后再把它加回到就绪队列中。

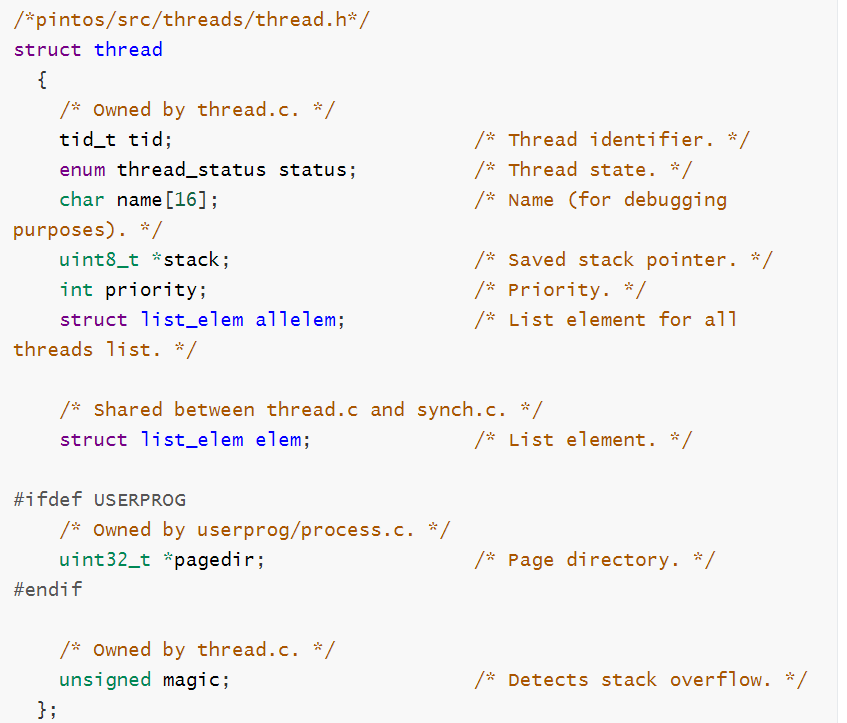
**1.2.2 改进思路**

1. 在thread.h头文件的thread结构体中增加整型变量block\_ticks，记录需要等待的时间长度。当线程被block的时候，将block\_ticks记录为需要等待的时间长度。之后每次中断的时候检查它一次，并使其自减。当block\_ticks等于0时，把线程调到ready queue中。
2. 改动timer.c中的timer\_sleep函数，调用thread\_block()函数使进程暂时阻塞。
3. 修改timer\_interrupt函数，使其每次中断都对所有线程执行block\_check函数。
4. 为timer.c增加block\_check函数用来检查每个进程中的block\_ticks是否为零，若为零将其唤醒。

##### 1.3、新函数实现

**1.3.1 thread.h头文件**

原thread结构体如下图：



增加整型变量block\_ticks，记录需要等待的时间长度：

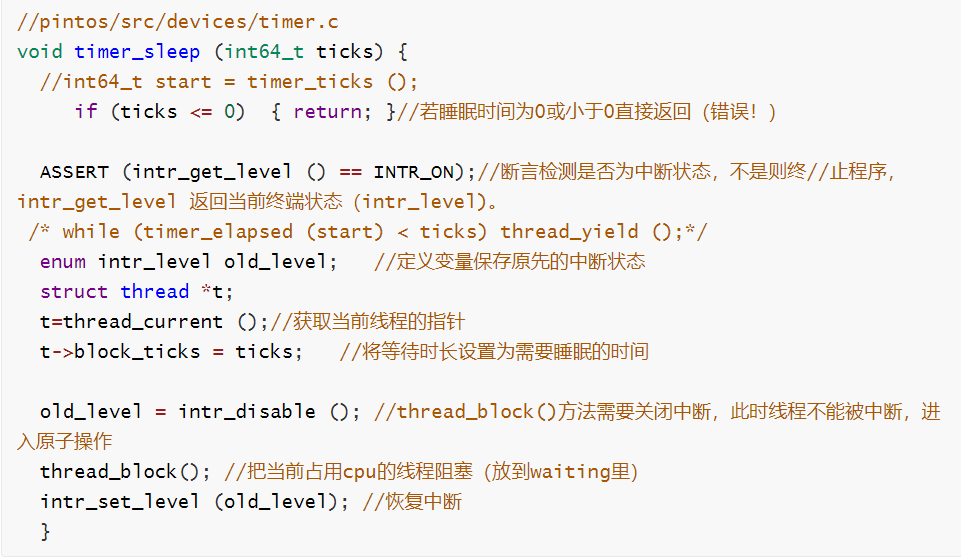


**1.3.2 timer.c文件**

改动timer.c中的timer\_sleep函数和timer\_interrupt函数，增加block\_check函数。

* **timer\_sleep函数**

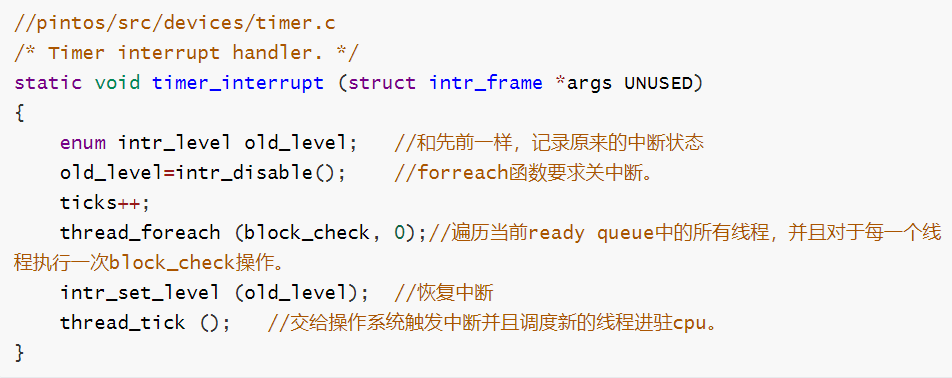
改动timer.c中的timer\_sleep函数，调用thread\_block()函数使进程暂时阻塞。



* **timer\_interrupt函数**

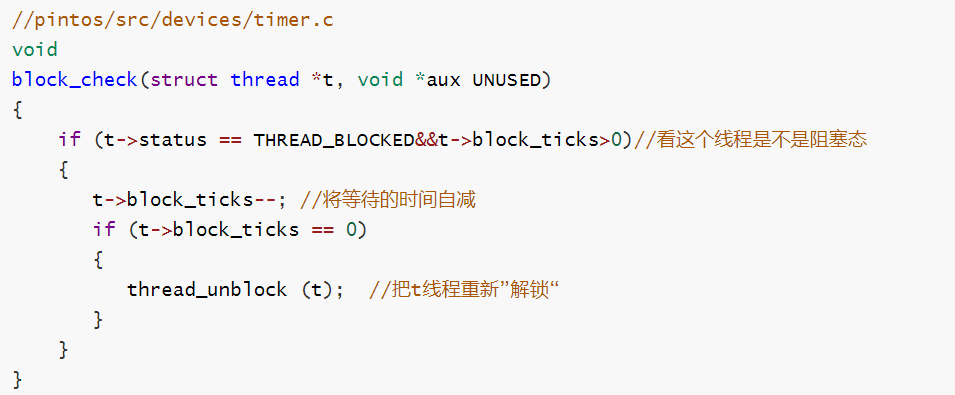
timer\_interrupt (struct intr\_frame \*args UNUSED)这个函数在timer.c中，pintos在每次时间中断时（即每一个时间单位（ticks））调用一次这个函数。

修改timer\_interrupt函数，使其每次中断都对所有线程执行block\_check函数。



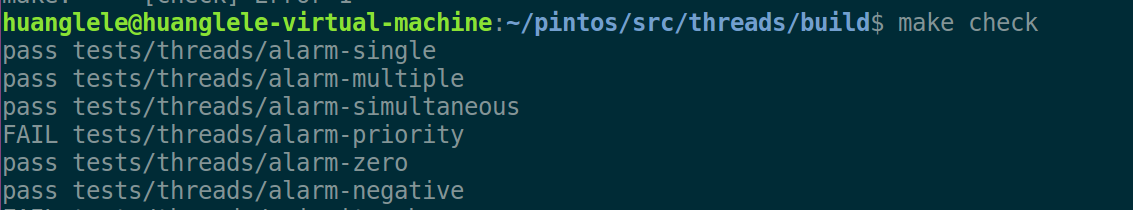
* **block\_check函数**

为timer.c增加block\_check函数，每次中断的时候检查block\_ticks一次，并使其自减，当block\_ticks等于0时，把该线程调到ready queue中。



##### 1.4、测试结果

在/home/（用户名）/pintos/src/thread 中执行make check，得到结果：



以上测试案例分别测试了单线程，多线程，多线程并发，和睡眠值为0和负数的错误情况，均已通过，以上结果说明task1的任务已经完成。

## Task2: Priority Scheduling

实现线程优先级调度：

线程成员本身priority：



实现优先级调度的核心思想就是： 维持就绪队列为一个优先级队列。

换一种说法： 我们在插入线程到就绪队列的时候保证这个队列是一个优先级队列即可。

##### 2.1、pass alarm\_priority

我们在什么时候会把一个线程丢到就绪队列中呢？

1. thread\_unblock 线程解除阻塞

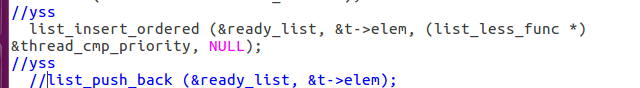
2. init\_thread 初始化（新建）线程

3. thread\_yield 线程中断，放弃CPU

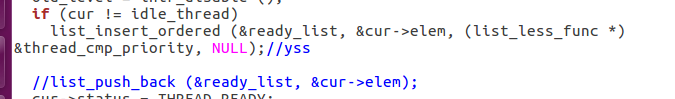
那么我们只要在扔的时候维持这个就绪队列是优先级队列即可。

观察代码我们可以看出这三种情况下插入线程都是直接放到队尾，不符合优先级队列思想。因此做出如下更改：

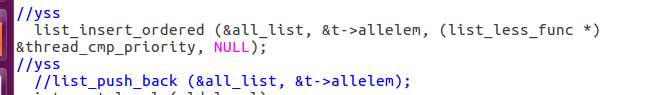
thread\_unblock



thread\_yield

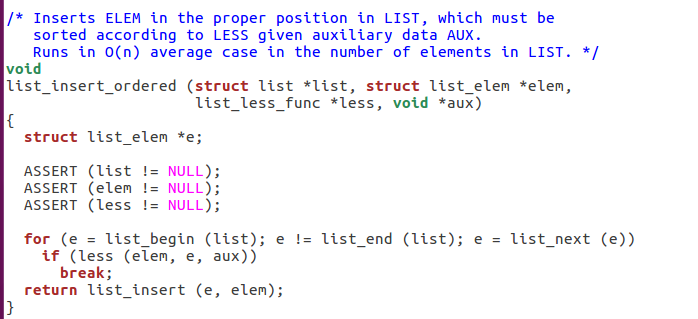


init\_thread



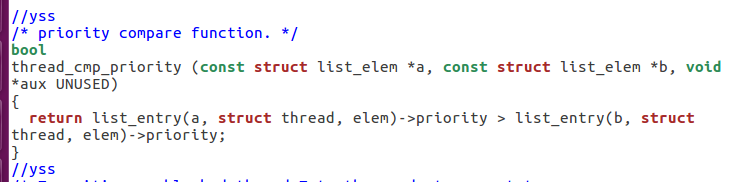
直接修改把list\_push\_back函数改成：list\_insert\_ordered 函数。

list\_insert\_ordered 函数功能是根据优先级比较来确定插入到队列的位置。



注：在列表中的适当位置插入elem，必须根据给定的辅助数据辅助排序。

然后实现一下比较函数thread\_cmp\_priority：

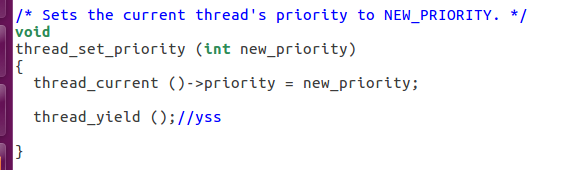


做完这些工作之后， 就可以pass alarm\_priority了。

##### 2.2、pass priority-change、priority-preempt

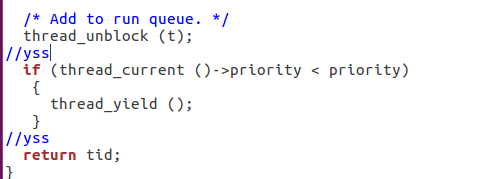
试想一下，当我们创建新线程或者给一个线程设置（或改变）优先级的时候，为维持优先级调度算法，我们应重新考虑所有线程执行顺序，重新安排执行顺序。因此，当我们设置线程优先级后，应立刻调用thread\_yield()函数，中断原线程，放入就绪队列中。

Void thread\_set\_priority (int new\_priority) //线程优先级设置函数



同理，当我们创造新线程后，应立刻比较原线程与新线程的优先级。若原线程优先级小于新线程优先级，则调用thread\_yield()函数，中断原线程，放入就绪队列中，这样即可实现线程优先级抢占调度算法。

tid\_t thread\_create (const char \*name, int priority,thread\_func \*function, void \*aux) //线程创建函数



做完这些工作，就可以pass priority-change 和 priority-preempt了。

##### 2.3、pass priority-donate-one、priority-donate-multiple、priority-donate-multiple2、priority-donate-nest、priority-donate-sema、priority-donate-lower、priority-sema、priority-condvar、priority-donate-chain

###### 优先级倒置问题

1.进程分优先级，高优先级进程需要执行时可打断现正在执行的低优先级进程；

2.普通的临界资源使用方法，如果一个临界资源被获取了，则其它想要获取此资源的程序被阻塞，直到此资源被释放；

3.例：有三个进程（其优先级从高到低分别为T1、T2、T3），有一个临界资源CS（T1与T3会用到）。这时，T3先执行，获取了临界资源CS。然后T2打断T3。接着T1打断T2，但由于CS已被T3获取，因此T1被阻塞，这样T2获得时间片。直到T2执行完毕后，T3接着执行，其释放CS后，T1才能获取CS并执行。

这时，我们看T1与T2，虽然T1优先级比T2高，但实际上T2优先于T1执行。从而产生了优先级倒置问题。

解决方案：

当发现高优先级的任务因为低优先级任务占用资源而阻塞时，就将低优先级任务的优先级提升到等待它所占有的资源的最高优先级任务的优先级。

###### 代码分析：

2.3.1、priority-donate-one

**测试代码：**

void

test\_priority\_donate\_one (void)

{

struct lock lock;

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

/\* Make sure our priority is the default. \*/

ASSERT (thread\_get\_priority () == PRI\_DEFAULT);

//当前线程是一个优先级为PRI\_DEFAULT的线程

lock\_init (&lock); //创建了一个锁（线程执行的时候会调用）

lock\_acquire (&lock); //加锁

thread\_create ("acquire1", PRI\_DEFAULT + 1, acquire1\_thread\_func, &lock);

//创建一个线程acquire1，优先级为PRI\_DEFAULT+1, 传了一个参数为这个锁的函数过去

msg ("This thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 1, thread\_get\_priority ());

thread\_create ("acquire2", PRI\_DEFAULT + 2, acquire2\_thread\_func, &lock);

//创建一个线程acquire2，优先级为PRI\_DEFAULT+2, 传了一个参数为这个锁的函数过去

msg ("This thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 2, thread\_get\_priority ());

lock\_release (&lock); //释放锁

msg ("acquire2, acquire1 must already have finished, in that order.");

msg ("This should be the last line before finishing this test.");

}

static void

acquire1\_thread\_func (void \*lock\_)

{

struct lock \*lock = lock\_;

lock\_acquire (lock); //加锁

msg ("acquire1: got the lock");

lock\_release (lock); //释放锁

msg ("acquire1: done");

}

static void

acquire2\_thread\_func (void \*lock\_)

{

struct lock \*lock = lock\_;

lock\_acquire (lock); //加锁

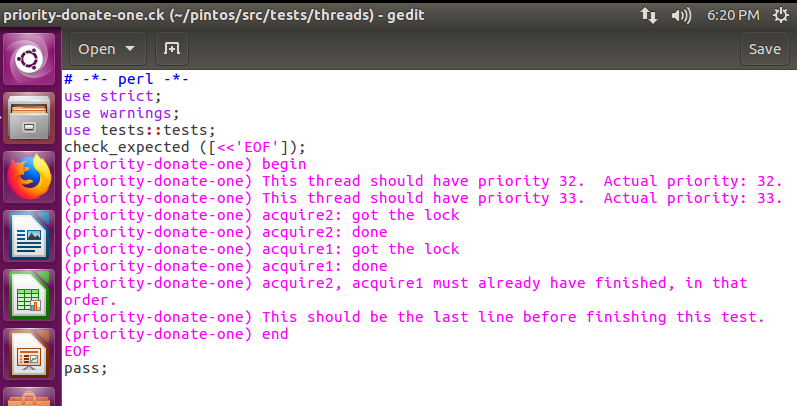
msg ("acquire2: got the lock");

lock\_release (lock); //释放锁

msg ("acquire2: done");

}

**希望输出：**



**执行过程**

把线程丢到这个信号量的队列waiters里， 阻塞该线程等待唤醒， value--。

注意， 这里acquire1\_thread\_func阻塞了， msg这个时候并不会输出， 这时会继续执行original\_thread, 然后输出msg， 输出当前线程应该的优先级和实际的优先级。

然后继续创建一个线程acquire2, 优先级为PRI\_DEFAULT+2，这里调用和acquire1一致， 然后original\_thread继续输出msg。

然后original\_thread释放了这个锁（V操作），释放的过程会触发被锁着的线程acquire1, acquire2， 然后根据优先级调度，先执行acquire2, 再acquire1, 最后再执行original\_thread。

那么这里应该是acquire2, acquire1分别释放锁然后输出msg， 最后original\_thread再输出msg。

实现思路是：

在一个线程获取一个锁的时候，如果拥有这个锁的线程优先级比自己低就提高它的优先级，然后在这个线程释放掉这个锁之后把原来拥有这个锁的线程改回原来的优先级。

2.3.2、priority-donate-multiple

**测试代码：**

void

test\_priority\_donate\_multiple (void)

{

struct lock a, b;

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

/\* Make sure our priority is the default. \*/

ASSERT (thread\_get\_priority () == PRI\_DEFAULT);

//当前线程是一个优先级为PRI\_DEFAULT的线程

lock\_init (&a); //创建锁a

lock\_init (&b); //创建锁b

lock\_acquire (&a); //加锁a

lock\_acquire (&b); //加锁b

thread\_create ("a", PRI\_DEFAULT + 1, a\_thread\_func, &a);

//创建一个线程a，优先级为PRI\_DEFAULT+1, 传了一个参数为这个锁的函数过去

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 1, thread\_get\_priority ());

thread\_create ("b", PRI\_DEFAULT + 2, b\_thread\_func, &b);

//创建一个线程b，优先级为PRI\_DEFAULT+1, 传了一个参数为这个锁的函数过去

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 2, thread\_get\_priority ());

lock\_release (&b); //释放锁b

msg ("Thread b should have just finished.");

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 1, thread\_get\_priority ());

lock\_release (&a); //释放锁a

msg ("Thread a should have just finished.");

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT, thread\_get\_priority ());

}

static void

a\_thread\_func (void \*lock\_)

{

struct lock \*lock = lock\_;

lock\_acquire (lock); //加锁

msg ("Thread a acquired lock a.");

lock\_release (lock); //释放锁

msg ("Thread a finished.");

}

static void

b\_thread\_func (void \*lock\_)

{

struct lock \*lock = lock\_;

lock\_acquire (lock); //加锁

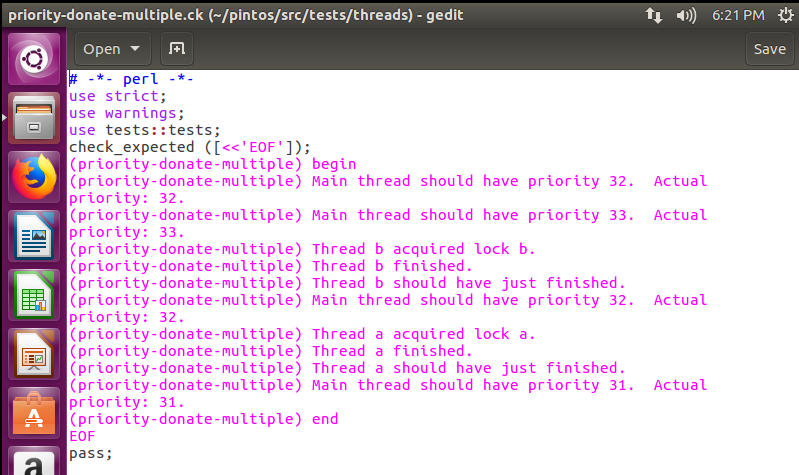
msg ("Thread b acquired lock b.");

lock\_release (lock); //释放锁

msg ("Thread b finished.");

}

**希望输出**



**执行过程**

original\_thread是优先级为PRI\_DEFAULT的线程， 然后创建2个锁， 接着创建优先级为PRI\_DEFAULT+1的线程a， 把锁a丢给这个线程的执行函数。

这时候线程a抢占式地调用a\_thread\_func， 获取了a这个锁， 阻塞。

然后original\_thread输出线程优先级的msg。

然后再创建一个线程优先级为PRI\_DEFAULT+2的线程b，和a一样做同样的操作。

然后original\_thread释放掉了锁b，此时线程b被唤醒，抢占式执行b\_thread\_func。

2.3.3、priority-donate-multiple2

**测试代码**

void

test\_priority\_donate\_multiple2 (void)

{

struct lock a, b;

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

/\* Make sure our priority is the default. \*/

ASSERT (thread\_get\_priority () == PRI\_DEFAULT);

//当前线程是一个优先级为PRI\_DEFAULT的线程

lock\_init (&a);//创建锁a

lock\_init (&b);//创建锁b

lock\_acquire (&a);//加锁a

lock\_acquire (&b);//加锁b

thread\_create ("a", PRI\_DEFAULT + 3, a\_thread\_func, &a);

//创建一个线程a，优先级为PRI\_DEFAULT+3, 传了一个参数为这个锁的函数过去

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 3, thread\_get\_priority ());

thread\_create ("c", PRI\_DEFAULT + 1, c\_thread\_func, NULL);

//创建一个线程c，优先级为PRI\_DEFAULT+1

thread\_create ("b", PRI\_DEFAULT + 5, b\_thread\_func, &b);

//创建一个线程b，优先级为PRI\_DEFAULT+5, 传了一个参数为这个锁的函数过去

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 5, thread\_get\_priority ());

lock\_release (&a);//释放锁a

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 5, thread\_get\_priority ());

lock\_release (&b);//释放锁b

msg ("Threads b, a, c should have just finished, in that order.");

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT, thread\_get\_priority ());

}

static void

a\_thread\_func (void \*lock\_)

{

struct lock \*lock = lock\_;

lock\_acquire (lock);//加锁

msg ("Thread a acquired lock a.");

lock\_release (lock);//释放锁

msg ("Thread a finished.");

}

static void

b\_thread\_func (void \*lock\_)

{

struct lock \*lock = lock\_;

lock\_acquire (lock);//加锁

msg ("Thread b acquired lock b.");

lock\_release (lock);//释放锁

msg ("Thread b finished.");

}

static void

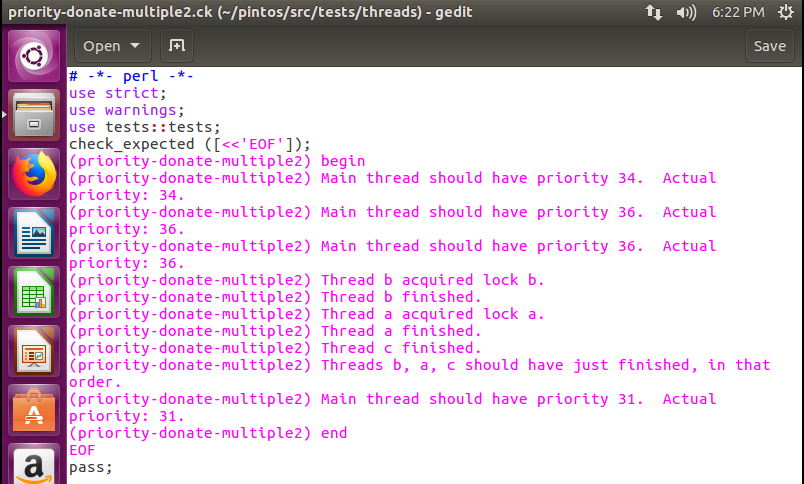
c\_thread\_func (void \*a\_ UNUSED)

{

msg ("Thread c finished.");

}

**希望输出**



测试： 多锁情况下优先级逻辑的正确性。

**执行过程**

original\_thread拥有2个锁， 然后创建PRI\_DEFAULT+3的线程a去拿a这个锁， PRI\_DEFAULT+5的线程b去拿b这个锁， 中间创建了一个PRI\_DEFAULT+1的c线程， 但是因为创建的时候当前线程的优先级已经被a线程捐赠了所以抢占调度并没有发生。 然后分别释放掉a和b， 释放a的时候线程a被唤醒， 但是优先级依然不如当前线程， 此时当前线程优先级仍然被b捐赠着， 优先级最高继续执行， 然后释放掉b， 释放掉b之后，original\_thread的优先级降到初始，应该最后被调用， 线程b抢占调度， 然后线程a， 再是线程c， 最后才original\_thread输出msg。

实现思路是： 释放一个锁的时候， 将该锁的拥有者改为该线程被捐赠的第二优先级，若没有其余捐赠者， 则恢复原始优先级。那么我们的线程必然需要一个数据结构来记录所有对这个线程有捐赠行为的线程。

2.3.4、priority-donate-nest

**测试代码**

struct locks

{

struct lock \*a;

struct lock \*b;

};

void

test\_priority\_donate\_nest (void)

{

struct lock a, b;

struct locks locks;

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

/\* Make sure our priority is the default. \*/

ASSERT (thread\_get\_priority () == PRI\_DEFAULT);

//当前线程是一个优先级为PRI\_DEFAULT的线程

lock\_init (&a);//创建锁a

lock\_init (&b);//创建锁b

lock\_acquire (&a);//加锁a

locks.a = &a;//赋值

locks.b = &b;//赋值

thread\_create ("medium", PRI\_DEFAULT + 1, medium\_thread\_func, &locks);

//创建一个线程medium，优先级为PRI\_DEFAULT+1, 传了一个参数为这个锁的函数过去

thread\_yield ();//线程中断

msg ("Low thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 1, thread\_get\_priority ());

thread\_create ("high", PRI\_DEFAULT + 2, high\_thread\_func, &b);

//创建一个线程high，优先级为PRI\_DEFAULT+2, 传了一个参数为这个锁的函数过去

thread\_yield ();//线程中断

msg ("Low thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 2, thread\_get\_priority ());

lock\_release (&a);//释放锁a

thread\_yield ();//线程中断

msg ("Medium thread should just have finished.");

msg ("Low thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT, thread\_get\_priority ());

}

static void

medium\_thread\_func (void \*locks\_)

{

struct locks \*locks = locks\_;

lock\_acquire (locks->b);//加锁locks->b

lock\_acquire (locks->a);//加锁locks->a

msg ("Medium thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 2, thread\_get\_priority ());

msg ("Medium thread got the lock.");

lock\_release (locks->a);//释放锁locks->a

thread\_yield ();//线程中断

lock\_release (locks->b);//释放锁locks->b

thread\_yield ();//线程中断

msg ("High thread should have just finished.");

msg ("Middle thread finished.");

}

static void

high\_thread\_func (void \*lock\_)

{

struct lock \*lock = lock\_;

lock\_acquire (lock);//加锁lock

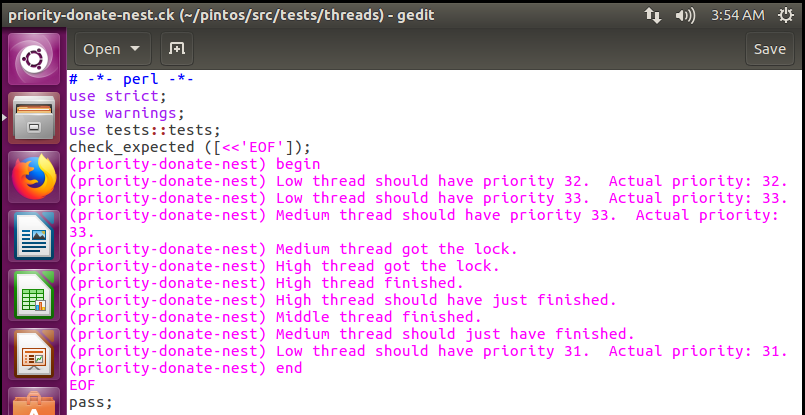
msg ("High thread got the lock.");

lock\_release (lock);//释放锁lock

msg ("High thread finished.");

}

**希望输出**



测试是一个优先级嵌套问题， 重点在于medium拥有的锁被low阻塞， 在这个前提下high再去获取medium的说阻塞的话， 优先级提升具有连环效应， 就是medium被提升了， 此时它被锁捆绑的low线程应该跟着一起提升。

**执行过程**

original\_thread只获取了锁a， 它并不拥有锁b。

这里创建了一个locks的结构体装着2个锁a和b， 然后创建medium线程， 优先级为PRI\_DEFAULT+1, 把这个locks作为线程medium执行函数的参数。

然后抢占调用medium\_thread\_func，此时这个函数获取b这个锁，此时medium成为锁b的拥有者，并不阻塞，继续执行，然后medium在获取锁a的时候阻塞了。

此时original\_thread继续跑，它的优先级被medium提到了PRI\_DEFAULT+1, 输出优先级Msg。

然后创建优先级为PRI\_DEFAULT+2的high线程，抢占调用high\_thread\_func， 然后这里high拿到了b这个锁， 而b的拥有者是medium， 阻塞，注意，这里medium被high捐赠了，优先级到PRI\_DEFAULT+2，此时original\_thread也应该一样提到同样优先级。

然后original\_thread输出一下优先级msg之后释放掉锁a，释放出发了medium\_thread\_func抢占调用， 输出此时优先级为PRI\_DEFAULT+2，然后medium释放掉a, 释放掉b，释放b的时候被high\_thread\_func抢占，high输出完之后medium继续run， 输出两句之后再到original\_thread， 输出两句msg完事。

实现方案：

线程需要加一个数据结构， 获取这个线程被锁于哪个线程。

2.3.5、priority-donate-sema

**测试代码**

struct lock\_and\_sema

{

struct lock lock;

struct semaphore sema;

};

void

test\_priority\_donate\_sema (void)

{

struct lock\_and\_sema ls;

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

/\* Make sure our priority is the default. \*/

ASSERT (thread\_get\_priority () == PRI\_DEFAULT);

//当前线程是一个优先级为PRI\_DEFAULT的线程

lock\_init (&ls.lock);//创建锁ls.lock

sema\_init (&ls.sema, 0);//初始化信号量ls.sema值为0

thread\_create ("low", PRI\_DEFAULT + 1, l\_thread\_func, &ls);

//创建一个线程low，优先级为PRI\_DEFAULT+1, 传了一个参数为这个锁的函数过去

thread\_create ("med", PRI\_DEFAULT + 3, m\_thread\_func, &ls);

//创建一个线程med，优先级为PRI\_DEFAULT+3, 传了一个参数为这个锁的函数过去

thread\_create ("high", PRI\_DEFAULT + 5, h\_thread\_func, &ls);

//创建一个线程high，优先级为PRI\_DEFAULT+5, 传了一个参数为这个锁的函数过去

sema\_up (&ls.sema);//信号量ls.sema值加1

msg ("Main thread finished.");

}

static void

l\_thread\_func (void \*ls\_)

{

struct lock\_and\_sema \*ls = ls\_;

lock\_acquire (&ls->lock);//加锁ls->lock

msg ("Thread L acquired lock.");

sema\_down (&ls->sema);//信号量ls.sema值减1

msg ("Thread L downed semaphore.");

lock\_release (&ls->lock);//释放锁ls->lock

msg ("Thread L finished.");

}

static void

m\_thread\_func (void \*ls\_)

{

struct lock\_and\_sema \*ls = ls\_;

sema\_down (&ls->sema);//信号量ls.sema值减1

msg ("Thread M finished.");

}

static void

h\_thread\_func (void \*ls\_)

{

struct lock\_and\_sema \*ls = ls\_;

lock\_acquire (&ls->lock);//加锁ls->lock

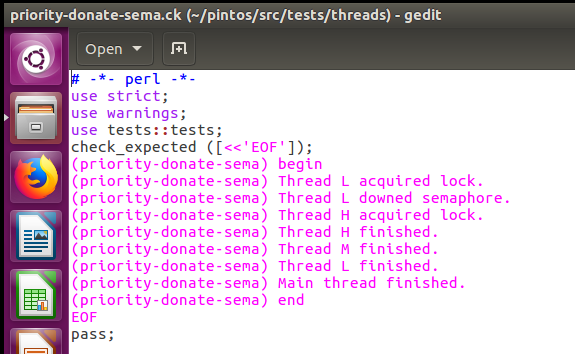
msg ("Thread H acquired lock.");

sema\_up (&ls->sema);//信号量ls.sema值加1

lock\_release (&ls->lock);//释放锁ls->lock

msg ("Thread H finished.");

}

**希望输出**

**执行过程**

lock\_and\_sema是包含一个锁和一个信号量的结构体，初始化信号量为0，然后创建PRI\_DEFAULT+1的线程low, 获取ls内的锁成为拥有者，然后sema\_down（P）阻塞。

然后创建PRI\_DEFAULT+3的线程med，这里也直接调用sema\_down(P)阻塞了。

最后创建PRI\_DEFAULT+5的线程high, 这里获取锁，阻塞。

然后回到original\_thread， 调用V操作， 此时唤醒了l\_thread\_func， 因为low被high捐献了优先级高于med， 然后l\_thread\_func跑， 释放掉了锁。

此时直接触发h\_thread\_func，输出，然后V操作，释放掉锁，由于优先级最高所以V操作之后不会被抢占， 这个函数跑完之后m\_thread\_func开始跑（被V唤醒），然后输出一句完事，再到l\_thread\_func中输出最后一句回到original\_thread。

这里包含了信号量和锁混合触发， 实际上还是信号量在起作用， 因为锁是由信号量实现的。

2.3.6、priority-donate-lower

**测试代码**

void

test\_priority\_donate\_lower (void)

{

struct lock lock;

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

/\* Make sure our priority is the default. \*/

ASSERT (thread\_get\_priority () == PRI\_DEFAULT);

//当前线程是一个优先级为PRI\_DEFAULT的线程

lock\_init (&lock);//创建锁lock

lock\_acquire (&lock);//加锁lock

thread\_create ("acquire", PRI\_DEFAULT + 10, acquire\_thread\_func, &lock);

//创建一个线程acquire，优先级为PRI\_DEFAULT+10, 传了一个参数为这个锁的函数过去

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 10, thread\_get\_priority ());

msg ("Lowering base priority...");

thread\_set\_priority (PRI\_DEFAULT - 10);//原线程优先级减10

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT + 10, thread\_get\_priority ());

lock\_release (&lock);//释放锁lock

msg ("acquire must already have finished.");

msg ("Main thread should have priority %d. Actual priority: %d.",

PRI\_DEFAULT - 10, thread\_get\_priority ());

}

static void

acquire\_thread\_func (void \*lock\_)

{

struct lock \*lock = lock\_;

lock\_acquire (lock);//加锁lock

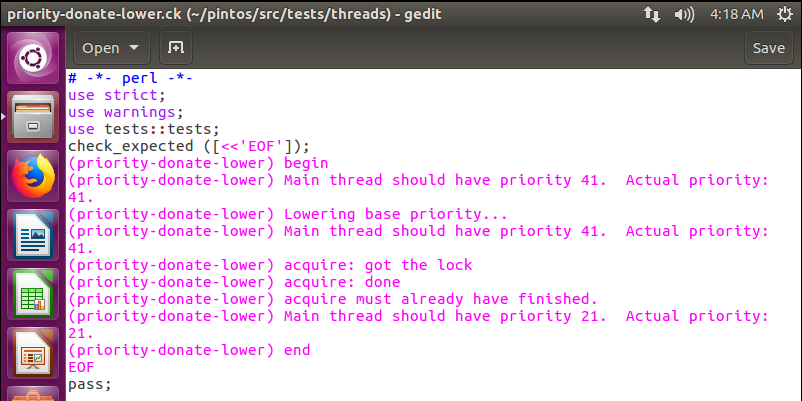
msg ("acquire: got the lock");

lock\_release (lock);//释放锁lock

msg ("acquire: done");

}

**希望输出**



**执行过程**

这里当前线程有一个锁，然后创建acquire抢占式获取了这个锁阻塞，然后此时original\_thread优先级为PRI\_DEFAULT+10，然后这里调用thread\_set\_priority， 此时当前线程的优先级应该没有改变，但是它以后如果恢复优先级时候其实是有改变的，就是说，我们如果用original\_priority来记录他的话，如果这个线程处于被捐赠状态的话则直接修改original\_priority来完成逻辑，此时函数过后优先级还是PRI\_DEFAULT+10，然后释放掉锁， acquire抢占输出和释放，然后original\_thread的优先级应该变成了PRI\_DEFAULT-10。

测试的是：当修改一个被捐赠的线程优先级的时候的行为正确性。

2.3.7、priority-sema

**测试代码**

void

test\_priority\_sema (void)

{

int i;

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

sema\_init (&sema, 0);//初始化信号量sema值为0

thread\_set\_priority (PRI\_MIN);//原线程优先级变为最小值0

for (i = 0; i < 10; i++)

{

int priority = PRI\_DEFAULT - (i + 3) % 10 - 1;

char name[16];

snprintf (name, sizeof name, "priority %d", priority);

thread\_create (name, priority, priority\_sema\_thread, NULL);

//创建一个线程，优先级为PRI\_DEFAULT-(i+3)%10-1

}

for (i = 0; i < 10; i++)

{

sema\_up (&sema);//信号量sema值加1

msg ("Back in main thread.");

}

}

static void

priority\_sema\_thread (void \*aux UNUSED)

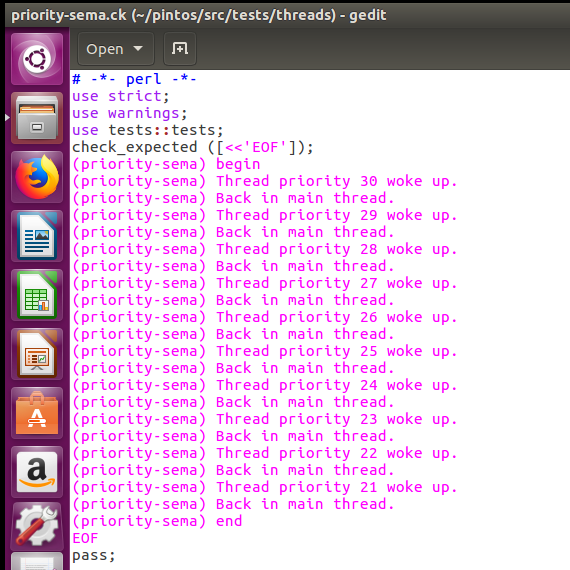
{

sema\_down (&sema);//信号量sema值减1

msg ("Thread %s woke up.", thread\_name ());

}

**希望输出**



**执行过程**

创建10个线程阻塞于P操作，然后用一个循环V操作。

实现方案：

V唤醒的时候也是优先级高的先唤醒，换句话说，信号量的等待队列是优先级队列。

2.3.8、priority-condvar

**测试代码**

void

test\_priority\_condvar (void)

{

int i;

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

lock\_init (&lock);//创建锁lock

cond\_init (&condition);//初始化条件变量condition

thread\_set\_priority (PRI\_MIN);//原线程优先级变为最小值0

for (i = 0; i < 10; i++)

{

int priority = PRI\_DEFAULT - (i + 7) % 10 - 1;

char name[16];

snprintf (name, sizeof name, "priority %d", priority);

thread\_create (name, priority, priority\_condvar\_thread, NULL);

//创建一个线程，优先级为PRI\_DEFAULT-(i+7)%10-1

}

for (i = 0; i < 10; i++)

{

lock\_acquire (&lock);//加锁lock

msg ("Signaling...");

cond\_signal (&condition, &lock);//条件变量condition唤醒

lock\_release (&lock);//释放锁lock

}

}

static void

priority\_condvar\_thread (void \*aux UNUSED)

{

msg ("Thread %s starting.", thread\_name ());

lock\_acquire (&lock);//加锁lock

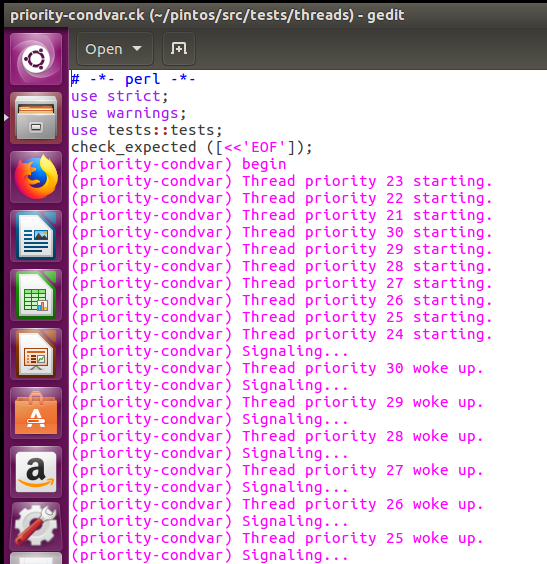
cond\_wait (&condition, &lock);//条件变量condition阻塞

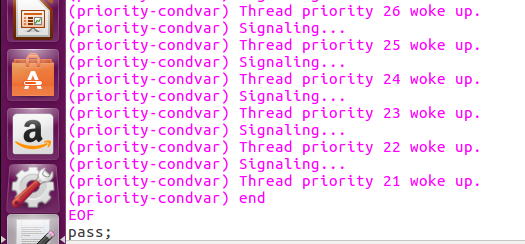
msg ("Thread %s woke up.", thread\_name ());

lock\_release (&lock);//释放锁lock

}

**希望输出**





**执行过程**

cond\_wait和cond\_signal就是释放掉锁，等待signal唤醒，然后再重新获取锁。

创建10个线程，每个线程调用的时候获取锁，然后调用cond\_wait把锁释放阻塞于cond\_signal唤醒，然后连续10次循环调用cond\_signal。

实现方案：

要求的实质就是： condition的waiters队列是优先级队列。

2.3.9、priority-donate-chain

**测试代码**

#define NESTING\_DEPTH 8//嵌套深度

struct lock\_pair //包含两个lock指针的结构体

{

struct lock \*second;

struct lock \*first;

};

void

test\_priority\_donate\_chain (void)

{

int i;

struct lock locks[NESTING\_DEPTH - 1];

struct lock\_pair lock\_pairs[NESTING\_DEPTH];//lock\_pairs数组用来装lock\_pair

/\* This test does not work with the MLFQS. \*/

ASSERT (!thread\_mlfqs);

thread\_set\_priority (PRI\_MIN);//原线程优先级变为最小值0

for (i = 0; i < NESTING\_DEPTH - 1; i++)

lock\_init (&locks[i]);//创建锁lock[i]

lock\_acquire (&locks[0]);//加锁lock[0]

msg ("%s got lock.", thread\_name ());

for (i = 1; i < NESTING\_DEPTH; i++)

{

char name[16];

int thread\_priority;

snprintf (name, sizeof name, "thread %d", i);

thread\_priority = PRI\_MIN + i \* 3;//3,6,9,12,15,18,21

lock\_pairs[i].first = i < NESTING\_DEPTH - 1 ? locks + i: NULL;

lock\_pairs[i].second = locks + i - 1;

//lock\_pairs[i]的first记录locks[i]的指针,second记录locks[i-1]指针

thread\_create (name, thread\_priority, donor\_thread\_func, lock\_pairs + i);

//创建一个线程，优先级为3,6,9,12,15,18,21, 传了一个参数为这个锁的函数过去

msg ("%s should have priority %d. Actual priority: %d.",

thread\_name (), thread\_priority, thread\_get\_priority ());

snprintf (name, sizeof name, "interloper %d", i);

thread\_create (name, thread\_priority - 1, interloper\_thread\_func, NULL);

//创建一个线程，优先级为2,5,8,11,14,17,20

}

lock\_release (&locks[0]);//释放锁lock[0]

msg ("%s finishing with priority %d.", thread\_name (),

thread\_get\_priority ());

}

static void

donor\_thread\_func (void \*locks\_)

{

struct lock\_pair \*locks = locks\_;

if (locks->first)

lock\_acquire (locks->first);//获取锁first, 即locks[i]

lock\_acquire (locks->second);//获取锁second, 即locks[i-1]

msg ("%s got lock", thread\_name ());

lock\_release (locks->second);//释放锁second, 即locks[i-1]

msg ("%s should have priority %d. Actual priority: %d",

thread\_name (), (NESTING\_DEPTH - 1) \* 3,

thread\_get\_priority ());

if (locks->first)

lock\_release (locks->first);//释放锁first, 即locks[i]

msg ("%s finishing with priority %d.", thread\_name (),

thread\_get\_priority ());

}

static void

interloper\_thread\_func (void \*arg\_ UNUSED)

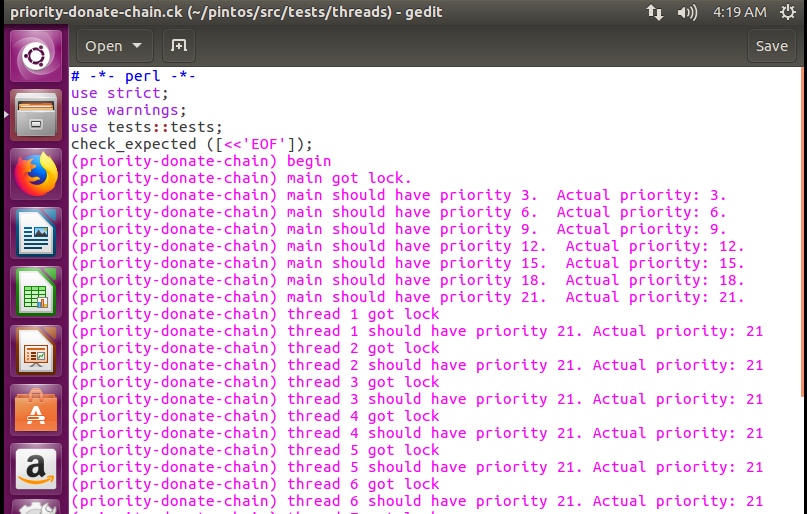
{

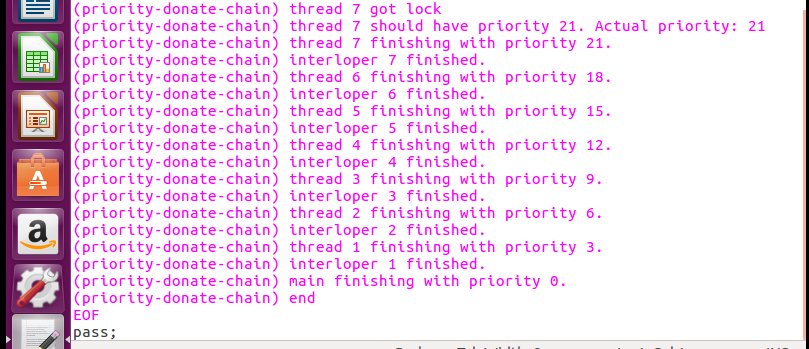
msg ("%s finished.", thread\_name ());

}

// vim: sw=2

**希望输出**





**执行过程**

首先lock\_pair是包含两个lock指针的结构体， 然后将当前线程优先级设为PRI\_MIN，然后这里有个locks数组，容量为7，然后lock\_pairs数组用来装lock\_pair，容量也是7。

然后当前线程获取locks[0]这个锁，接着跳到7次循环里，每次循环thread\_priority为PRI\_MIN+i\*3，也就是3,6,9,12... 然后对应的lock\_pairs[i]的first记录locks[i]的指针， second记录locks[i-1]指针，

然后创建线程，优先级为thread\_priority, 执行参数传的是&lock\_pairs[i]， 注意这里由于优先级每次都底层，所以每次循环都会抢占调用donor\_thread\_func，然后分别获取lock\_pairs[i]里装的锁，然后每次循环先获取first, 即locks[i], 然后获取second, 由于second是前一个，而前一个的拥有者一定是前一次循环创建的线程，第一次拿得的是locks[0]，最后一次循环first为NULL，second为locks[6]，即最后一个线程不拥有锁，但是阻塞于前一个创建的线程，这里还会输出信息，即创建的线程阻塞之后会输出当前线程的优先级msg，当然这里必然是每一次都提升了的，所以每次都是thread\_priority。

然后每次循环最后还创建了1个线程，优先级为thread\_priority-1，但是这里由于上一个线程创建和阻塞的过程中优先级捐献已经发生，所以这里并不发生抢占，只是创建出来了而已。

然后original\_thread释放掉locks[0]，释放掉这个之后thread1得到了唤醒，输出信息，释放掉这个锁，然后输出当前优先级，由于这个线程还是被后面最高优先级的线程说捐赠的，所以每次往后优先级都是21，然后释放掉first， 这里又触发下一个线程继续跑，注意当后面的全部跑完的时候当前线程的优先级其实是不被捐赠的，这里就变成了原来的优先级，但是是所有线程都释放了之后才依次返回输出结束msg。

实现方案：

测试是一个链式优先级捐赠，本质测试还是多层优先级捐赠逻辑的正确性。

需要注意的是一个逻辑：释放掉一个锁之后，如果当前线程不被捐赠即马上改为原来的优先级，抢占式调度。

**代码更改：**

综合以上9个测试，做出总结：

1.  在一个线程获取一个锁的时候， 如果拥有这个锁的线程优先级比自己低就提高它的优先级，并且如果这个锁还被别的锁锁着， 将会递归地捐赠优先级， 然后在这个线程释放掉这个锁之后恢复未捐赠逻辑下的优先级。

2. 如果一个线程被多个线程捐赠， 维持当前优先级为捐赠优先级中的最大值（acquire和release之时）。

3. 在对一个线程进行优先级设置的时候， 如果这个线程处于被捐赠状态， 则对original\_priority进行设置， 然后如果设置的优先级大于当前优先级， 则改变当前优先级， 否则在捐赠状态取消的时候恢复original\_priority。

4. 在释放锁对一个锁优先级有改变的时候应考虑其余被捐赠优先级和当前优先级。

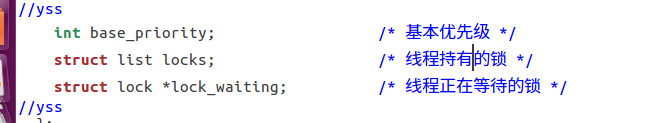
5. 将信号量的等待队列实现为优先级队列。

6. 将condition的waiters队列实现为优先级队列。

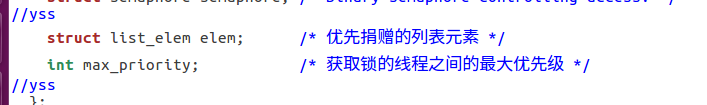
7. 释放锁的时候若优先级改变则可以发生抢占。

根据总结，做出代码修改：

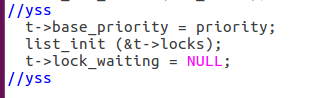
1.thread数据结构， 加入成员：



1. lock数据结构， 加入成员：

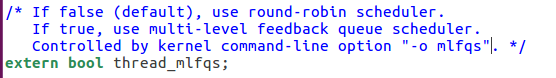


1. 在init\_thread函数中加入初始化



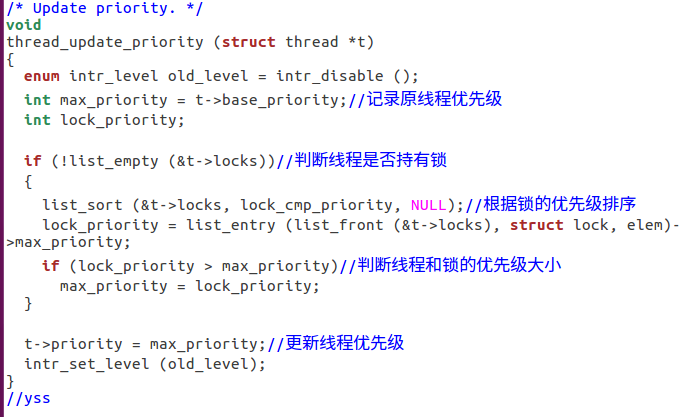
用来表示线程所持有的锁。

注：我们先看一个bool类型的变量：



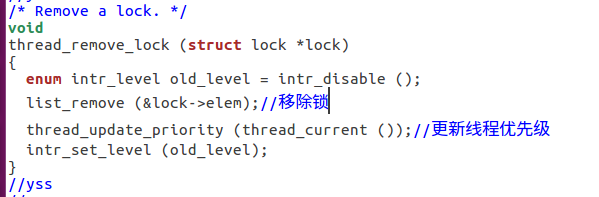
如果错误（默认），使用循环轮转调度；如果正确，使用多级反馈队列调度。

1. thread\_update\_priority函数实现（当释放掉一个锁的时候， 当前线程的优先级可能发生变化）

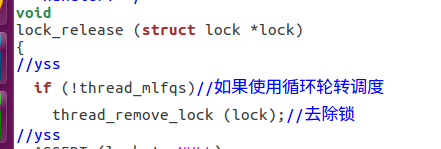


如果这个线程还有锁， 就先获取这个线程拥有锁的最大优先级（可能被更高级线程捐赠）， 然后如果这个优先级比base\_priority大的话更新的应该是被捐赠的优先级。

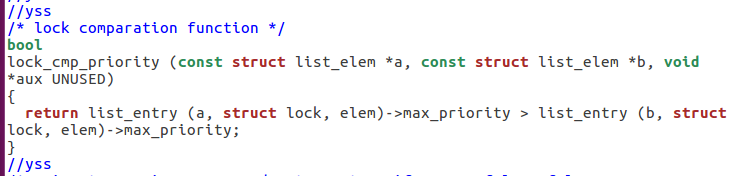
1. thread\_remove\_lock函数实现



1. 在lock\_release函数加入以下语句：



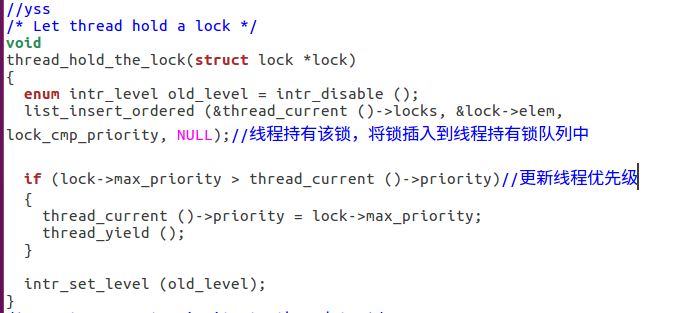
1. 实现锁队列排序函数lock\_cmp\_priority（稍后会用到）:

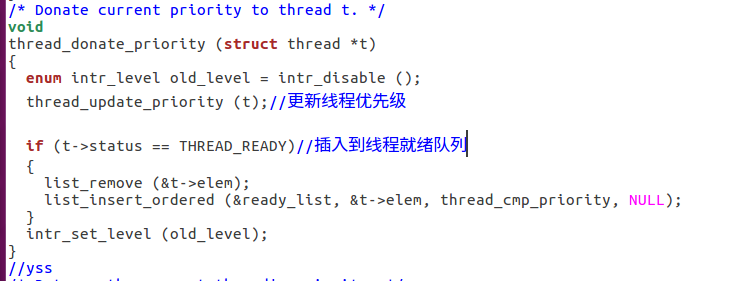


1. 封装thread\_donate\_priority和thread\_hold\_the\_lock函数

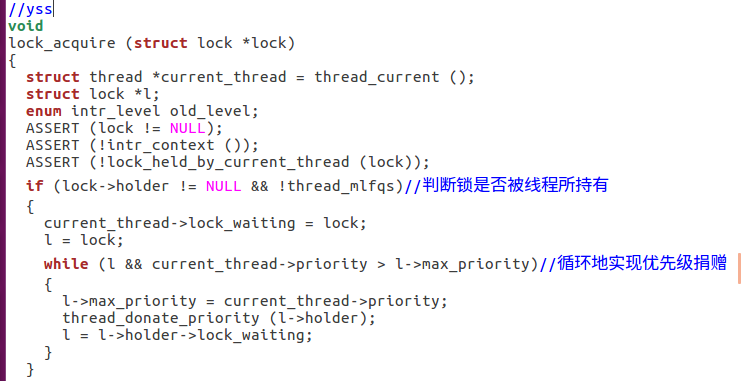
注：优先级捐赠是通过直接修改锁的最高优先级， 然后调用update的时候把现成优先级更新实现。

thread\_hold\_the\_lock（线程持有锁）：

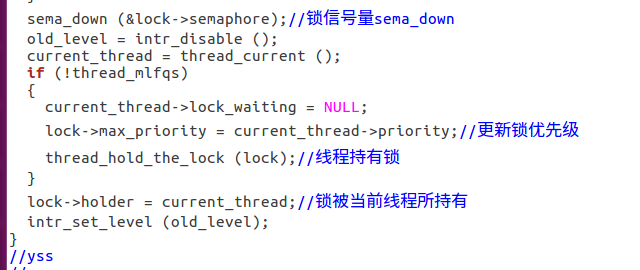
thread\_donate\_priority（线程捐献优先级）：



1. 修改lock\_acquire函数



在P操作之前递归地实现优先级捐赠。



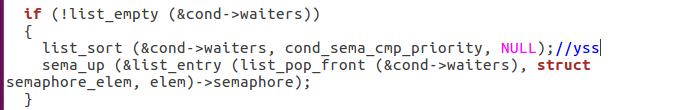
在被唤醒之后（此时这个线程已经拥有了这个锁），成为这个锁的拥有者。

1. 修改thread\_set\_priority：

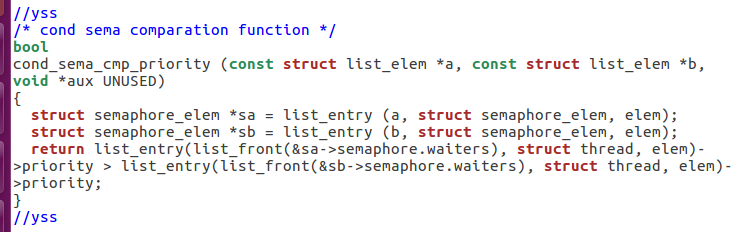


1. 修改cond\_signal函数，将condition的队列改成优先级队列：

添加一行代码，对condition的队列排序。



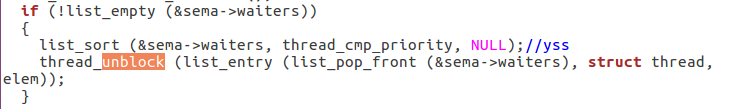
实现cond\_sema\_cmp\_priority排序函数：



1. 把信号量的等待队列实现为优先级队列，修改sema\_up和sema\_down函数：

sema\_up：

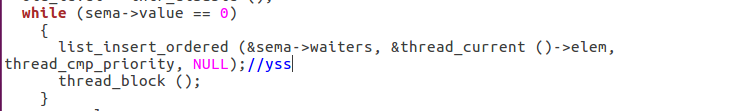
添加一行代码，对信号量的等待队列排序。



sema\_down：

将代码：list\_push\_back (&sema->waiters, &thread\_current ()->elem)； 修改

在信号量的等待队列中的适当位置插入elem。



##### 2.4、实验结果



## Task3：Advanced Scheduler

##### 3.1、原理

在多级反馈队列调度器中，线程的优先级是通过下面的公式动态计算的：

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

recent\_cpu = (2\*load\_avg)/(2\*load\_avg + 1)\*recent\_cpu + nice

load\_avg= (59/60)\*load\_avg + (1/60)\*ready\_threads

变量含义如下：

nice ---- 线程属性，取值[-20,+20]，越大表示该线程出让更多的CPU时间

priority ---- 每个线程有一个优先级，介于0（PRI\_MIN）到63（PRI\_MAX）之间，每4个timer\_ticks更新一次，ticks作为基本时间单位

recent\_cpu ---- 线程消耗的CPU时间tick。每个tick加1；当timer\_ticks () % TIMER\_FREQ == 0时对所有线程更新，即每1秒也加1

load\_avg ---- 过去一分钟内准备运行的平均线程数，初始0，每1秒更新

ready\_threads ---- 更新时运行或准备运行的线程数（不包括空闲线程）

##### 3.2、函数实现

* **定义及初始化：**

**thread.h文件中，在结构体thread中添加成员**

struct thread

{

……

int nice;

int recent\_cpu;

};

**thread.c中定义全局变量**

int load\_avg

**thread.c中thread\_init()函数内初始化**

t->recent\_cpu=0;

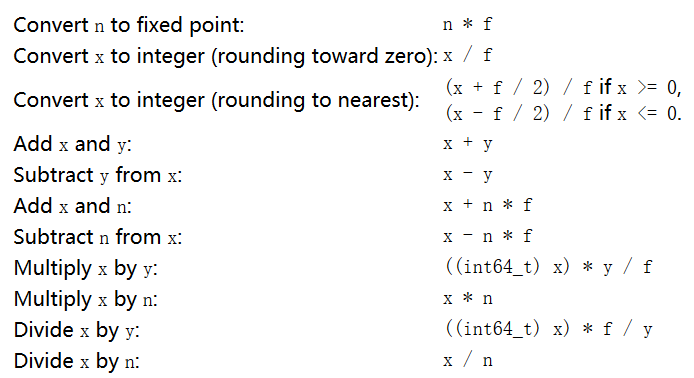
t->nice=0;

load\_avg=0;

* **浮点数运算：**

本实验中priority，nice和ready\_threads是整数，recent\_cpu和load\_avg是实数，由于pintos不支持浮点数运算，所以需要我们自己实现浮点数运算。这里用16位数（FP\_SHIFT\_AMOUNT）作为浮点数的小数部分，运算是要维持整数部分不变，从第17位开始运算。

下表总结了C语言中如何实现定点算术运算。x，y为定点数，n为整数，定点数用符号p.q格式表示，其中p+q=31, f=1<<q。



浮点运算逻辑实现在fixed\_point.h中：

#ifndef \_\_THREAD\_FIXED\_POINT\_H

#define \_\_THREAD\_FIXED\_POINT\_H

/\* Basic definitions of fixed point. \*/

typedef int fixed\_t;

/\* 16 LSB used for fractional part. \*/

#define FP\_SHIFT\_AMOUNT 16

/\* Convert a value to fixed-point value. \*/

#define FP\_CONST(A) ((fixed\_t)(A << FP\_SHIFT\_AMOUNT))

/\* Add two fixed-point value. \*/

#define FP\_ADD(A,B) (A + B)

/\* Add a fixed-point value A and an int value B. \*/

#define FP\_ADD\_MIX(A,B) (A + (B << FP\_SHIFT\_AMOUNT))

/\* Substract two fixed-point value. \*/

#define FP\_SUB(A,B) (A - B)

/\* Substract an int value B from a fixed-point value A \*/

#define FP\_SUB\_MIX(A,B) (A - (B << FP\_SHIFT\_AMOUNT))

/\* Multiply a fixed-point value A by an int value B. \*/

#define FP\_MULT\_MIX(A,B) (A \* B)

/\* Divide a fixed-point value A by an int value B. \*/

#define FP\_DIV\_MIX(A,B) (A / B)

/\* Multiply two fixed-point value. \*/

#define FP\_MULT(A,B) ((fixed\_t)(((int64\_t) A) \* B >> FP\_SHIFT\_AMOUNT))

/\* Divide two fixed-point value. \*/

#define FP\_DIV(A,B) ((fixed\_t)((((int64\_t) A) << FP\_SHIFT\_AMOUNT) / B))

/\* Get integer part of a fixed-point value. \*/

#define FP\_INT\_PART(A) (A >> FP\_SHIFT\_AMOUNT)

/\* Get rounded integer of a fixed-point value. \*/

#define FP\_ROUND(A) (A >= 0 ? ((A + (1 << (FP\_SHIFT\_AMOUNT - 1))) >> FP\_SHIFT\_AMOUNT) \

: ((A - (1 << (FP\_SHIFT\_AMOUNT - 1))) >> FP\_SHIFT\_AMOUNT))

#endif /\* thread/fixed\_point.h \*/

* **timer\_interrupt函数：**

在timer\_interrupt中固定一段时间计算更新线程的优先级，这里是每TIMER\_FREQ时间更新一次系统load\_avg和所有线程的recent\_cpu， 每4个timer\_ticks更新一次线程优先级，每个timer\_tick running线程的recent\_cpu加一，虽然这里说的是维持64个优先级队列调度，其本质还是优先级调度， 我们保留之前写的优先级调度代码即可，去掉优先级捐赠（之前donate相关代码已经对需要的地方加了thread\_mlfqs的判断了）。

* **先实现timer\_interrupt函数的逻辑，加入如下代码：**

if (thread\_mlfqs)

{

**thread\_mlfqs\_increase\_recent\_cpu\_by\_one ()**; /\* 每个timer\_tick running线程的recent\_cpu加一 \*/

if (ticks % TIMER\_FREQ == 0)

**thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu ()**; /\* 每TIMER\_FREQ时间更新一次系统load\_avg和所有线程的recent\_cpu \*/

else if (ticks % 4 == 0)

**thread\_mlfqs\_update\_priority (thread\_current ())**; /\* 每4个timer\_ticks更新一次线程优先级 \*/

}

**然后需要分别实现上述三个函数**

* **每个timer\_tick running线程的recent\_cpu加一**

recent\_cpu为线程消耗的CPU时间，需要判断当前线程是否空闲，如果不空闲则recent\_cpu的值+1。

void thread\_mlfqs\_increase\_recent\_cpu\_by\_one (void)

{

ASSERT (thread\_mlfqs); //断言

ASSERT (intr\_context ());

struct thread \*current\_thread = thread\_current ();

if (current\_thread == idle\_thread)

return; //更新时运行或准备运行的线程数（不包括空闲线程）

current\_thread->recent\_cpu = FP\_ADD\_MIX (current\_thread->recent\_cpu, 1); //实现加1

}

* **每TIMER\_FREQ时间更新一次系统load\_avg和所有线程的recent\_cpu**

判断当前线程是否空闲，若不空闲，则利用load\_avg= (59/60)\*load\_avg + (1/60)\*ready\_threads和recent\_cpu = (2\*load\_avg)/(2\*load\_avg + 1)\*recent\_cpu + nice两个公式对load\_avg和所有线程的recent\_cpu进行更新。

其中可以看到我们通过使用list\_entry（）函数得到了一个线程（list\_entry(...)->priority）,再看list\_entry（）的三个传入参数list\_begin(&ready\_list)，struct thread,elem，那么我们是不是可以大胆猜想list\_entry()函数通过elem在ready\_list中查找，找到了begin元素相应的thread.其实list\_entry()函数的作用就是如此，不仅可以在ready\_list中查找，还可以在all\_list,和waiters中查找相应的实体结构。

void thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu (void)

{

ASSERT (thread\_mlfqs);

ASSERT (intr\_context ());

size\_t ready\_threads = list\_size (&ready\_list);

if (thread\_current () != idle\_thread) //非空闲进程

ready\_threads++;//线程队列

load\_avg = FP\_ADD (FP\_DIV\_MIX (FP\_MULT\_MIX (load\_avg, 59), 60), FP\_DIV\_MIX (FP\_CONST (ready\_threads), 60));

/\* load\_avg= (59/60)\*load\_avg + (1/60)\*ready\_threads \*/

struct thread \*t;

struct list\_elem \*e = list\_begin (&all\_list);

for (; e != list\_end (&all\_list); e = list\_next (e))

{

t = list\_entry(e, struct thread, allelem);//所有线程

if (t != idle\_thread)

{

t->recent\_cpu = FP\_ADD\_MIX (FP\_MULT (FP\_DIV (FP\_MULT\_MIX (load\_avg, 2), FP\_ADD\_MIX (FP\_MULT\_MIX (load\_avg, 2), 1)), t->recent\_cpu), t->nice);

/\* 实现recent\_cpu = (2\*load\_avg)/(2\*load\_avg + 1)\*recent\_cpu + nice \*/

thread\_mlfqs\_update\_priority (t);

}

}

}

* **每4个timer\_ticks更新一次线程优先级**

判断当前线程是否空闲，若不空闲，则利用priority = PRI\_MAX - (recent\_cpu/4) - (nice \* 2)公式对线程优先级进行更新。由于priority的值介于0（PRI\_MIN）到63（PRI\_MAX）之间，所以计算值若大于PRI\_MAX，则更新的线程优先级为63（PRI\_MAX），计算值若小于PRI\_MIN，则更新的线程优先级为0（PRI\_MIN）。

void thread\_mlfqs\_update\_priority (struct thread \*t)

{

if (t == idle\_thread)

return;

ASSERT (thread\_mlfqs);

ASSERT (t != idle\_thread);

t->priority = FP\_INT\_PART (FP\_SUB\_MIX (FP\_SUB (FP\_CONST (PRI\_MAX), FP\_DIV\_MIX (t->recent\_cpu, 4)), 2 \* t->nice)); //实现priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2)

if (t-> priority > PRI\_MAX) t-> priority = PRI\_MAX;

if (t-> priority < PRI\_MIN) t-> priority = PRI\_MIN;

//介于0（PRI\_MIN）到63（PRI\_MAX）之间

}

* **四个辅助函数：**

thread.c文件中还有四个辅助函数需要我们自己实现。

* **thread\_set\_nice函数**

将当前线程的nice值设置为nice。然后线程的优先级发生了改变，需要重新计算优先级并调度

void thread\_set\_nice (int nice)

{

thread\_current ()->nice = nice;

thread\_mlfqs\_update\_priority (thread\_current ());

thread\_yield ();

}

* **thread\_get\_nice函数**

获取当前线程的nice值

int thread\_get\_nice (void)

{

return thread\_current ()->nice;

}

* **thread\_get\_load\_avg函数**

获取load\_avg值的100倍。四舍五入取整。

int thread\_get\_load\_avg (void)

{

return FP\_ROUND (FP\_MULT\_MIX (load\_avg, 100));

}

* **thread\_get\_recent\_cpu函数**

获取recent\_cpu值的100倍。四舍五入取整。

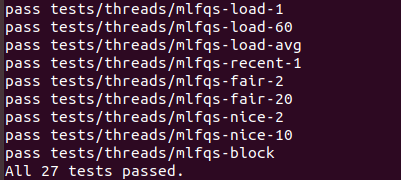
int thread\_get\_recent\_cpu (void)

{

return FP\_ROUND (FP\_MULT\_MIX (thread\_current ()->recent\_cpu, 100));

}

##### 3.3、测试结果



27个测试全部通过

# 四、参考文献

http://www.ccs.neu.edu/home/amislove/teaching/cs5600/fall10/pintos/pintos\_7.html

https://www.cnblogs.com/laiy/p/pintos\_project1\_thread.html