

# 操作系统课程设计pintos project1实验摘记



双鱼座羊驼 发布于 2 月 6 日

**前言：**本篇意在记录本学期结束的操作系统课程设计pintos project1实验报告和实现过程。整个实验参考了多篇文章也查阅了一些代码，其中部分内容或其他文章相同，还请见谅。

## 第一部分 项目概述

### 一、Pintos简介

sf



注册登录

### 二、项目要求

#### 1、项目一 线程管理

在这一项目中，我们需要进行三部分的改进，以实现如下功能：

- 第一部分：重新实现 `timer_sleep()` 函数，以避免线程在就绪和运行状态间的不停切换。
- 第一部分：实现优先级调度



最终目标：使该项目的27个检测点全部通过。

### 三、环境搭建与配置

1、在VMware Workstation中安装Ubuntu 18.04LTS虚拟机，并进行基本的配置。

2、在终端运行安装Bochs Simulator命令

```
sudo apt-get install bochs
```

3、使用VIM打开`/utils/pintos-gdb`，并编辑`GDBMACROS`变量

```
GDBMACROS=/home/pisces/pintos-anon-master-f685123/src/utils/gdb-macros
```

4、使用VIM打开`Makefile`并将`LOADLIBES`变量名编辑为`LDLIBS`

```
LDLIBS = -lm
```

5、在`/src/utils`中输入`make`来编译`utils`

6、编辑`/src/threads/Make.vars`：将`SIMULATOR`的类型更改为`bochs`

```
SIMULATOR = --bochs
```

7、在`/src/threads`并运行来编译线程目录`make`

8、编辑`/utils/pintos`

(1) `$sim`变量的类型为`bochs`

```
$sim = "bochs" if !defined $sim;
```

(2) 替换`kernel.bin`为完整路径

```
my $name = find_file ('/home/pisces/pintos-anon-master-f685123/src/threads/build/kernel
```



(3) 替换loader.bin为完整路径

```
$name = find_file ("/home/pisces/pintos-anon-master-f685123/src/threads/build/loader.bi
```



9、打开~/.bashrc并添加如下语句至最后一行

```
export PATH=/home/pisces/pintos-anon-master-f685123/src/utils:$PATH
```

10、重新打开终端输入source ~/.bashrc并运行

11、在Pintos下打开终端输入pintos run alarm-multiple会出现如下情形

12、在/threads/build下进行make并make check将会出现如下结果



```
pisces@ubuntu: ~/pintos-anon-master-f685123/src/threads/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
(mlfqs-block) end
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
FAIL tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
FAIL tests/threads/priority-change
FAIL tests/threads/priority-donate-one
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
FAIL tests/threads/priority-fifo
FAIL tests/threads/priority-preempt
FAIL tests/threads/priority-sema
FAIL tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
20 of 27 tests failed.
../../tests/Make.tests:26: recipe for target 'check' failed
make: *** [check] Error 1
pisces@ubuntu:~/pintos-anon-master-f685123/src/threads/build$
```

此时，我们的环境就配置好了，由于项目一还未做修改，因此出现20个failed是正常情况。

## 第二部分 项目一 线程管理的实现

### 一、线程管理概述

对于 alarm 系列测试来说，需要解决的是忙等问题，因此需要引入“线程三态模型”，将暂时不用的线程挂起（yield CPU），可以提高系统的吞吐率。

对于 priority、preempt 系列测试来说，需要实现一个线程形式的“优先队列”数据结构来支持优先级调度，因此可以使用“堆”数据结构（heap）来实现，但是使用  $O(n)$  的扫描算法也可以通过测试，因此以求简单实现和保证系统的吞吐量正常，使用普通的扫描算法来实现优先级调度。基



priority\_donation 要解决的根本问题是“优先级翻转”问题，即低优先级的线程 可以比高优先级的线程先运行，产生一种类死锁的现象。因此，需要使用“信号量”（semaphore）、“条件变量”（condition）和锁（lock）等数据结构。基本思想是，信号量的等待队列应当设计为优先队列，获取锁要考虑优先级捐赠情况，释放锁时要考虑抢占情况。

多级反馈队列 mlfqs 在实现浮点数运算后，根据官方给出的调度公式计算出 nice 以及 recent\_cpu，同时对 timer.c 中的 timer\_interrupt () 函数进行修改即可实现。

## 二、代码文件说明

在项目一中将会用到如下文件和文件夹

### (1) 文件夹功能说明

文件夹	功能
threads	基本内核的代码
devices	IO 设备接口，定时器、键盘、磁盘等代码
lib	实现了 C 标准库，该目录代码会与 Pintos kernel 一起编译，用户的程序也要在此目 录下运行。内核程序和用户程序都可以使用 #include 的方式来引入这个目录下的头 文件

### (2) threads/ 文件夹中文件说明

文件	功能
loader.h	内核加载器
loader.S	
kernel.ld.s.S	连接脚本，用于连接内核，设置负载地址的内核
init.h	内核的初始化，包括 main() 函数



文件	功能
thread.h	实现基础线程功能
thread.c	
switch.h	汇编语言实现常规的用于交换的线程
switch.S	
start.S	跳转到主函数

(3) devices/ 文件夹中文件说明

文件	功能
timer.h	实现系统计时器，默认使每秒运行 100 次
timer.c	
vga.h	显示驱动程序，负责将文本打印在屏幕上
vga.c	
serial.h	串行端口驱动程序，通过 printf() 函数调用，将内容并传入输入层
serial.c	
disk.h	支持磁盘进行读写操作
disk.c	
kbd.h	支持磁盘进行读写操作
kbd.c	
input.h	输入层程序，将传入的字符组成输入队列



文件	功能
input.c	
intq.h	中断队列程序，管理内核线程和中断处理程序
intq.c	

### 三、项目分析

线程管理这一项目的入手点是`timer_sleep()`函数，以下展示该函数的整体结构：

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void timer_sleep(int64_t ticks)
{
    int64_t start = timer_ticks(); //获取开始的时间

    ASSERT(intr_get_level() == INTR_ON);
    while (timer_elapsed(start) < ticks) //查看当前时间是否小于设定的睡眠时间
        thread_yield();                 //将当前线程放入就绪队列，并调度下一个线程
}
```

在第5行中，首先获得了线程休眠的开始时间，`timer_ticks()`函数在获取时间的过程中采用了关中断保存程序状态字，而后开中断恢复程序状态字的办法以防止在执行过程中发生中断，由于后续程序也用到了开关中断的操作，因此将在接下来进行介绍。

在第7行中，断言了当前中断的状态，确保中断是打开的状态。

接下来是重点部分，首先看`timer_elapsed()`函数，其整体结构如下：

```
/* Returns the number of timer ticks elapsed since THEN, which
   should be a value once returned by timer_ticks(). */
int64_t
timer_elapsed(int64_t then)
{
    return timer_ticks() - then;
}
```



我们可以看到这个函数实际是计算当前线程已经休眠的时间，它将结果返回至`timer_sleep()`函数后，利用`while`循环判断休眠时间是否已经达到`ticks`时间（这里的`ticks`时间是传入的拟休眠时间的局部变量，而不是全局变量系统启动后到现在的时间），如果没有达到，就将不停的进行`thread_yield()`。

`thread_yield()`函数的整体结构如下所示：

```
/* Yields the CPU.  The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
void thread_yield(void)
{
    struct thread *cur = thread_current(); //获取当前页面的初始位置（指针指向开始）
    enum intr_level old_level;

    ASSERT(!intr_context());

    old_level = intr_disable();             //关中断
    if (cur != idle_thread)                 //如果当前线程不是空闲线程
        list_push_back(&ready_list, &cur->elem); //把当前线程放入就绪队列
    cur->status = THREAD_READY;             //修改程序状态为就绪
    schedule();                             //调度下一个线程
    intr_set_level(old_level);              //开中断
}
```

### （1）页面指针的获取

在第5行，`cur`指针通过调用`thread_current()`函数来获取指向页面初始位置的指针，由于该函数也进行了多级嵌套，在此仅简要描述一下函数功能实现的流程。首先，这一函数获取了`esp`寄存器的值，这一寄存器是指向栈顶的寄存器，为了获取指向页首的指针，我们知道Pintos中一页的大小是2的12次方，因此其做法就是将1这个数字在二进制下左移12位并取反后，再与`esp`寄存器中的值相与，即可获得页首指针。

### （2）原子化操作

所谓原子化操作即为开篇所提到的关中断和开中断操作，其分别由以下两个语句实现：

```
old_level = intr_disable();             //关中断
////其他操作
intr_set_level(old_level);              //开中断
```

其基本实现步骤是利用堆栈的`push`和`pop`语句得到程序状态字寄存器的值，并利用`CLI`指令关中





### (3) 线程的切换

这一步骤体现在11-14行代码中，如果当前线程不是空闲线程，则把它加入就绪队列，加入的方式是通过指针的改变使其与前后的线程关联起来，形成一个队列，并将这一线程修改为就绪状态。

`schedule()`函数的实现是获取当前线程和即将要运行的线程，其中的`prev = switch_threads(cur, next);`语句利用纯汇编的方式将寄存器中的值改为指向下一线程的值，`thread_schedule_tail(prev)`函数将即将运行的线程标记为运行状态，并判断原先线程是否已经进入消亡的状态，如果是便顺便清空页表信息。由于这些操作过于底层，对其的理解较为浅显，仅在此表明大意，暂不深入。

由此可以得知`thread_yield()`函数的作用便是将当前线程放入就绪队列，并调度下一线程。而`timer_sleep()`函数便是在限定的时间内，使运行的程序不停的放入就绪队列，从而达到休眠的目的。

当然这样去做的一大缺点，就是线程不断的在运行和就绪的状态来回切换，极大的消耗资源，由此我们将进行改进。

## 四、线程管理实现第一部分——`timer_sleep()`函数的重新实现

### 1、实现思想

由于原本的`timer_sleep()`函数采用运行和就绪间切换的方法过于消耗CPU的资源，考虑到Pintos提供了线程阻塞这一模式（见下方线程状态结构体），因此我打算在线程结构体中加入一个用于记录线程睡眠时间的变量，通过利用Pintos的时钟中断（见下方时间中断函数），即每个tick将会执行一次，这样每次检测时将记录线程睡眠时间的变量自减1，当该变量为0时即可代表能够唤醒该线程，从而避免资源的过多开销。

线程状态结构体：

```
enum thread_status
{
    THREAD_RUNNING,      /* Running thread. */
    THREAD_READY,        /* Not running but ready to run. */
    THREAD_BLOCKED,      /* Waiting for an event to trigger. 阻塞状态*/
    THREAD_DYING         /* About to be destroyed. */
};
```

时间中断函数：



```

/* Timer interrupt handler. */
static void
timer_interrupt(struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick();
}

```

## 2、实现步骤

(1) 改写线程结构体，在结构体中增加记录线程睡眠时间的变量 `ticks_blocked`

```

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. */
    int64_t ticks_blocked; //增加的变量->记录要阻塞的时间

    /* Shared between thread.c and synch.c. */
    struct list_elem elem; /* List element. */

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

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

```

(2) 改写线程创建函数，增加记录线程睡眠时间的变量的赋初值操作 `t->ticks_blocked = 0;`

```

/* Initialize thread. */
init_thread (t, name, priority);
tid = t->tid = allocate_tid ();
t->ticks_blocked = 0; //增加的初始化操作

```

```

/* Stack frame for kernel thread(). */

```



```

kf->function = function;
kf->aux = aux;

/* Stack frame for switch_entry(). */
ef = alloc_frame (t, sizeof *ef);
ef->eip = (void (*) (void)) kernel_thread;

/* Stack frame for switch_threads(). */
sf = alloc_frame (t, sizeof *sf);
sf->eip = switch_entry;
sf->ebp = 0;

/* Add to run queue. */
thread_unblock (t);

return tid;
}

```

(3) 改写`timer_sleep()`函数，获取当前的要阻塞的线程，为其设置阻塞时间，并调用Pintos的线程阻塞函数（要注意这一操作不可被中断，因此要加入开关中断操作）

```

void
timer_sleep (int64_t ticks)
{
    if (ticks <= 0)
    {
        return;
    }
    ASSERT (intr_get_level () == INTR_ON);
    enum intr_level old_level = intr_disable ();
    struct thread *current_thread = thread_current ();
    current_thread->ticks_blocked = ticks;           //设置阻塞时间
    thread_block ();                                  //调用阻塞函数
    intr_set_level (old_level);
}

```

(4) 在`timer_interrupt()`中加入`thread_foreach (blocked_thread_check, NULL)`；以对所有被阻塞线程进行阻塞时间的计算

```

static void
timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();
}

```



### (5) 实现blocked\_thread\_check()函数

在thread.h中声明:

```
void blocked_thread_check (struct thread *t, void *aux UNUSED);
```

在thread.c中实现:

实现逻辑即为如果这个线程处于阻塞态且阻塞时间还未结束, 则减小其阻塞时间并判断当其不用再被阻塞, 便调用Pintos函数thread\_unblock()将线程放入就绪队列并更改为就绪态。

```
void  
blocked_thread_check (struct thread *t, void *aux UNUSED)  
{  
    if (t->status == THREAD_BLOCKED && t->ticks_blocked > 0)  
    {  
        t->ticks_blocked--;  
        if (t->ticks_blocked == 0)  
        {  
            thread_unblock(t);  
        }  
    }  
}
```

至此timer\_sleep()的唤醒机制便编写完成了。

## 3、实现结果

此时, 在/threads/build重新make check的结果如下所示:

```
pass tests/threads/alarm-single  
pass tests/threads/alarm-multiple  
pass tests/threads/alarm-simultaneous  
FAIL tests/threads/alarm-priority  
pass tests/threads/alarm-zero  
pass tests/threads/alarm-negative
```

## 五、线程管理实现第二部分——优先级调度的实现




## (1) 实现思路:

由于Pintos预置函数`list_insert_ordered()`的存在, 可以直接使用这个函数实现线程插入时按照优先级完成, 因此只需要将涉及直接在末尾插入线程的函数中的语句进行替换即可。

## (2) 实现步骤:

实现一个优先级比较函数`thread_cmp_priority()`:

```
/* priority compare function. */
bool
thread_cmp_priority (const struct list_elem *a, const struct list_elem *b, void *aux UN
{
    return list_entry(a, struct thread, elem)->priority > list_entry(b, struct thread, el
}
```




调用Pintos预置函数`list_insert_ordered()`替换`thread_unblock()`、`init_thread()`、`thread_yield()`中的`list_push_back()`函数:

```
void
thread_unblock (struct thread *t)
{
    enum intr_level old_level;

    ASSERT (is_thread (t));

    old_level = intr_disable ();
    ASSERT (t->status == THREAD_BLOCKED);
    list_insert_ordered (&ready_list, &t->elem, (list_less_func *) &thread_cmp_priority,
    t->status = THREAD_READY;
    intr_set_level (old_level);
}
```



```
static void
init_thread (struct thread *t, const char *name, int priority)
{
    enum intr_level old_level;

    ASSERT (t != NULL);
```



```

memset (t, 0, sizeof *t);
t->status = THREAD_BLOCKED;
strncpy (t->name, name, sizeof t->name);
t->stack = (uint8_t *) t + PGSIZE;
t->priority = priority;
t->magic = THREAD_MAGIC;

old_level = intr_disable ();
list_insert_ordered (&all_list, &t->elem, (list_less_func *) &thread_cmp_priority,
intr_set_level (old_level);
}

```

```

void
thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    ASSERT (!intr_context ());

    old_level = intr_disable ();
    if (cur != idle_thread)
        list_insert_ordered (&ready_list, &cur->elem, (list_less_func *) &thread_cmp_priority,
cur->status = THREAD_READY;
schedule ();
intr_set_level (old_level);
}

```

(3) 测试结果:

alarm\_priority测试顺利通过!

## 2、优先级机制的继续改进

(1) 思路与实现步骤

根据Pintos给到的测试用例来看，我们可以知道，当一个线程的优先级被改变，则需要立即考虑



还需要考虑创建线程时的特殊情况，如果创建的线程优先级高于正在运行的线程的优先级，则需要将正在运行的线程加入就绪队列，并且使新建线程准备运行。代码如下：

```
void
thread_set_priority (int new_priority)
{
    thread_current ()->priority = new_priority;
    thread_yield (); //加入线程按照优先级排序函数的调用
}
```

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
    struct thread *t;
    struct kernel_thread_frame *kf;
    struct switch_entry_frame *ef;
    struct switch_threads_frame *sf;
    tid_t tid;

    ASSERT (function != NULL);

    /* Allocate thread. */
    t = palloc_get_page (PAL_ZERO);
    if (t == NULL)
        return TID_ERROR;

    /* Initialize thread. */
    init_thread (t, name, priority);
    tid = t->tid = allocate_tid ();
    t->ticks_blocked = 0;

    /* Stack frame for kernel_thread(). */
    kf = alloc_frame (t, sizeof *kf);
```

(2) 测试结果：

priority\_change、priority\_fifo和priority\_preempt均已通过测试！

2 通过其他优先级测试程序的具体实现



priority-donate-one测试用例表明，如果一个线程在获取锁时发现另一个比自己优先级更低的线程已经拥有相同的锁，那么这个线程将会捐赠自己的优先级给另一个线程，即提升另一个线程的优先级与自己相同。

priority-donate-multiple与priority-donate-multiple2测试用例表明，在恢复线程捐赠后的优先级时，也要考虑其他线程对这个线程的捐赠情况，即需要提供一个数据结果来记录给这个线程捐赠优先级的所有线程。

priority-donate-nest测试用例表明，优先级捐赠可以是递归的，因而需要数据结果记录线程正在等待哪个另外线程释放锁。

priority-donate-lower测试用例表明，如果线程处于捐赠状态，在修改时线程优先级依然是被捐赠的优先级，但释放锁后线程的优先级变成了修改后的优先级。

priority-sema和priority-condvar测试用例表明，需要将信号量的等待队列实现为优先级队列，同时也要将condition的waiters队列实现为优先级队列。

priority-donate-chain测试用例表明，释放锁后如果线程没有被捐赠，则需要立即恢复原来的优先级。

## (2) 实现步骤：

在thread结构体中加入记录基本优先级、记录线程持有锁和记录线程等待锁的数据结构：

```
struct thread
{
    ...
    int base_priority;           /* Base priority.新加的 */
    struct list locks;          /* Locks that the thread is holding.新加的 */
    struct lock *lock_waiting;  /* The lock that the thread is waiting for. 新力
    ...
}
```

将上述数据结构在init\_thread中初始化：

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
    ...
    t->base_priority = priority;
```





```
...  
}
```

在lock结构体中加入记录捐赠和记录最大优先级的数据结构:

```
struct lock  
{  
...  
    struct list_elem elem;    /* List element for priority donation. 新加的*/  
    int max_priority;          /* Max priority among the threads acquiring the lock. 新力*/  
};
```

修改synch.c中的lock\_acquire函数, 使其能够以循环的方式实现递归捐赠, 并通过修改锁的max\_priority成员, 再通过thread\_update\_priority函数更新优先级来实现优先级捐赠:

```
void lock_acquire (struct lock *lock)  
{  
    struct thread *current_thread = thread_current ();  
    struct lock *l;  
    enum intr_level old_level;  
  
    ASSERT (lock != NULL);  
    ASSERT (!intr_context ());  
    ASSERT (!lock_held_by_current_thread (lock));  
  
    if (lock->holder != NULL && !thread_mlfqs)  
    {  
        current_thread->lock_waiting = lock;  
        l = lock;  
        while (l && current_thread->priority > l->max_priority)  
        {  
            l->max_priority = current_thread->priority;  
            thread_donate_priority (l->holder);  
            l = l->holder->lock_waiting;  
        }  
    }  
}
```

实现thread\_donate\_priority和lock\_cmp\_priority, 以达到对线程优先级的更新和在队列中位置的重新排布:



```

enum intr_level old_level = intr_disable ();
thread_update_priority (t);

if (t->status == THREAD_READY)
{
    list_remove (&t->elem);
    list_insert_ordered (&ready_list, &t->elem, thread_cmp_priority, NULL);
}
intr_set_level (old_level);
}

bool lock_cmp_priority (const struct list_elem *a, const struct list_elem *b, void *aux)
{
    return list_entry (a, struct lock, elem)->max_priority > list_entry (b, struct lock,
}

```

实现thread\_hold\_the\_lock和lock\_cmp\_priority，以达到对线程拥有锁的记录，同时根据锁记录的线程最大优先级更新当前线程的优先级并重新调度：

```

void thread_hold_the_lock(struct lock *lock)
{
    enum intr_level old_level = intr_disable ();
    list_insert_ordered (&thread_current ()->locks, &lock->elem, lock_cmp_priority, NULL)

    if (lock->max_priority > thread_current ()->priority)
    {
        thread_current ()->priority = lock->max_priority;
        thread_yield ();
    }

    intr_set_level (old_level);
}

```

```

bool lock_cmp_priority (const struct list_elem *a, const struct list_elem *b, void *aux)
{
    return list_entry (a, struct lock, elem)->max_priority > list_entry (b, struct lock,
}

```



```

void lock_release (struct lock *lock)
{
    ASSERT (lock != NULL);
    ASSERT (lock_held_by_current_thread (lock));

    //new code
    if (!thread_mlfqs)
        thread_remove_lock (lock);

    lock->holder = NULL;
    sema_up (&lock->semaphore);
}

```



```

void thread_remove_lock (struct lock *lock)
{
    enum intr_level old_level = intr_disable ();
    list_remove (&lock->elem);
    thread_update_priority (thread_current ());
    intr_set_level (old_level);
}

```

实现thread\_update\_priority, 该函数实现释放锁时优先级的变化, 如果当前线程还有锁, 则获取其拥有锁的max\_priority, 如果它大于base\_priority则更新被捐赠的优先级:

```

void thread_update_priority (struct thread *t)
{
    enum intr_level old_level = intr_disable ();
    int max_priority = t->base_priority;
    int lock_priority;

    if (!list_empty (&t->locks))
    {
        list_sort (&t->locks, lock_cmp_priority, NULL);
        lock_priority = list_entry (list_front (&t->locks), struct lock, elem)->max_priority;
        if (lock_priority > max_priority)
            max_priority = lock_priority;
    }

    t->priority = max_priority;
    intr_set_level (old_level);
}

```



```

void thread_set_priority (int new_priority)
{
    if (thread_mlfqs)
        return;

    enum intr_level old_level = intr_disable ();

    struct thread *current_thread = thread_current ();
    int old_priority = current_thread->priority;
    current_thread->base_priority = new_priority;

    if (list_empty (&current_thread->locks) || new_priority > old_priority)
    {
        current_thread->priority = new_priority;
        thread_yield ();
    }

    intr_set_level (old_level);
}

```

接下来实现sema和condvar这两个优先队列，修改cond\_signal函数，声明并实现比较函数cond\_sema\_cmp\_priority:

```

void cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
    ASSERT (cond != NULL);
    ASSERT (lock != NULL);
    ASSERT (!intr_context ());
    ASSERT (lock_held_by_current_thread (lock));

    if (!list_empty (&cond->waiters))
    {
        list_sort (&cond->waiters, cond_sema_cmp_priority, NULL);
        sema_up (&list_entry (list_pop_front (&cond->waiters), struct semaphore_elem, elem))
    }
}

```

```

bool cond_sema_cmp_priority (const struct list_elem *a, const struct list_elem *b, void
{
    struct semaphore_elem *sa = list_entry (a, struct semaphore_elem, elem);
    struct semaphore_elem *sb = list_entry (b, struct semaphore_elem, elem);
    return list_entry (list_pop_front (&sa->waiters), struct thread_elem, current_thread)

```



最后把信号量的等待队列实现为优先级队列：

```
void sema_up (struct semaphore *sema)
{
    enum intr_level old_level;

    ASSERT (sema != NULL);

    old_level = intr_disable ();
    if (!list_empty (&sema->waiters))
    {
        list_sort (&sema->waiters, thread_cmp_priority, NULL);
        thread_unblock (list_entry (list_pop_front (&sema->waiters), struct thread, elem));
    }

    sema->value++;
    thread_yield ();
    intr_set_level (old_level);
}
```

```
void sema_down (struct semaphore *sema)
{
    enum intr_level old_level;

    ASSERT (sema != NULL);
    ASSERT (!intr_context ());

    old_level = intr_disable ();
    while (sema->value == 0)
    {
        list_insert_ordered (&sema->waiters, &thread_current ()->elem, thread_cmp_priority,
            thread_block ());
    }
    sema->value--;
    intr_set_level (old_level);
}
```

(3) 实现结果：



至此关于优先级的测试案例就全部通过了！

## 六、线程管理实现第三部分——多级反馈调度的实现

### 1、实现思路

根据官方实验指导 [Pintos Projects: 4.4BSD Scheduler \(neu.edu\)](#) 的说明，需要实现维持64个队列的优先级，同时需要一些计算公式来计算当前优先级，计算公式如下图：

因此需要在 `timer_interrupt` 函数中固定时间更新优先级，每4个ticks更新一次，同时保证 `recent_cpu` 自增1。

### 2、实现过程：

(1) 实现运算逻辑：新建 `fixed_point.h` 文件，并按照计算公式编写运算程序

```
/* Convert a value to a 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))
/* Subtract two fixed-point value. */
#define FP_SUB(A,B) (A - B)
/* Subtract 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 the integer part of a fixed-point value. */
#define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)

/* Get the 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 /* threads/fixed-point.h */
```



(2) 修改timer\_interrupt函数，实现每4个ticks更新一次，同时保证recent\_cpu自增1的要求：

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();
    thread_foreach (blocked_thread_check, NULL);

    //new code
    if (thread_mlfqs)
    {
        mlfqs_inc_recent_cpu();
        if (ticks % TIMER_FREQ == 0)
            mlfqs_update_load_avg_and_recent_cpu();
        else if (ticks % 4 == 0)
            mlfqs_update_priority(thread_current());
    }
}
```

(3) 实现recent\_cpu自增函数

```
void mlfqs_inc_recent_cpu()
{
    ASSERT(thread_mlfqs);
    ASSERT(intr_context());

    struct thread *cur = thread_current();
    if (cur == idle_thread)
        return;
    cur->recent_cpu = FP_ADD_MIX(cur->recent_cpu, 1);
}
```

(4) 实现mlfqs\_update\_load\_avg\_and\_recent\_cpu函数

```
void
mlfqs_update_load_avg_and_recent_cpu()
{
    ASSERT(thread_mlfqs);
    ASSERT(intr_context());

    size_t ready_cnt = list_size(&ready_list);
    if (thread_current() != idle_thread)
```



```

struct thread *t;
struct list_elem *e;
for (e = list_begin(&all_list); e != list_end(&all_list); e = list_next(e))
{
    t = list_entry(e, struct thread, allelem);
    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), \
            mlfqs_update_priority(t);
    }
}
}

```

### (5) 实现mlfqs\_update\_priority函数

```

void
mlfqs_update_priority(struct thread *t)
{
    ASSERT(thread_mlfqs);

    if (t == idle_thread)
        return;

    t->priority = FP_INT_PART (FP_SUB_MIX (FP_SUB (FP_CONST (PRI_MAX), \
        FP_DIV_MIX (t->recent_cpu, 4)), 2 * t->nice));
    if (t->priority < PRI_MIN)
        t->priority = PRI_MIN;
    else if (t->priority > PRI_MAX)
        t->priority = PRI_MAX;
}

```

### (6) 在thread结构体中加入成员并在init\_thread中初始化:

```

struct thread
{
    ...
    int nice; /* Niceness. */
    fixed_t recent_cpu; /* Recent CPU. */
    ...
}

```





```
static void init_thread (struct thread *t, const char *name, int priority)
{
    ...
    t->nice = 0;
    t->recent_cpu = FP_CONST (0);
    ...
}
```

(7) 在thread.c中声明全局变量load\_avg:

```
fixed_t load_avg;
```

(8) 在thread\_start中初始化load\_avg:

```
void
thread_start (void)
{
    load_avg = FP_CONST (0);
    ...
}
```

(9) 在thread.h中包含浮点运算头文件:

```
#include "fixed_point.h"
```

(10) 在thread.c中修改thread\_set\_nice、thread\_get\_nice、thread\_get\_load\_avg、thread\_get\_recent\_cpu函数:

```
    thread_yield();
}

/* Returns the current thread's nice value. */
int
thread_get_nice (void)
{
    /* Solution Code */
    return thread_current()->nice;
}

/* Returns 100 times the system load average. */
int
```



```
/* Solution Code */
return FP_ROUND (FP_MULT_MIX (load_avg, 100));
}

/* Returns 100 times the current thread's recent_cpu value. */
int
thread_get_recent_cpu (void)
{
    /* Solution Code */
    return FP_ROUND (FP_MULT_MIX (thread_current()->recent_cpu, 100));
}
```

### 3、实现结果：

至此，27个测试均已通过，项目一全部完成。

操作系统

阅读 653 • 发布于 2 月 6 日



收藏

分享

本作品系原创，采用《署名-非商业性使用-禁止演绎 4.0 国际》许可协议



双鱼座羊驼

1 声望 0 粉丝

关注作者

0 条评论

得票

最新





## 继续阅读

### SpringCloud——Eureka Feign Ribbon Hystrix Zuul等关键组件的学习与记录

前言：本篇是对近日学习狂神SpringCloud教程之后的感想和总结，鉴于对SpringCloud体系的了解尚且处于...

双鱼座羊驼 阅读 468

### Linux操作系统基础

Linux简介Linux是一种自由和开放源码的操作系统，存在着许多不同的Linux版本，但它们都使用了Linux内...

BOOTLOIBF 赞 2 阅读 1.2k

### 操作系统概述

我们平常所说的计算机，严格的讲应该指的是计算机系统，它包含硬件（如CPU，内存，显示器，打印机等...

stan 赞 1 阅读 2.8k

### 【操作系统学习】二、操作系统的引导

根据前面一节的操作，将hit-oslab解压后会出现三个文件夹，其中linux-0.11里面有一个文件夹叫做boot，...

ztyzz 阅读 1.1k

### COMP2211操作系统

Operating Systems (COMP2211)Coursework 5: Independent projectSubmission You should submit...

zi7napyt 阅读 332

### CS3103操作系统

CS3103: Operating SystemsSpring 2021Programming Assignment 21 GoalsThe purpose of this...

kjxt6387 阅读 77

### 啃碎操作系统(一)：操作系统概念

一般而言，现代计算机是一个复杂的系统。如果每位应用程序员都不得不掌握系统的所有细节，那就不可能...

超大只乌龟 阅读 1.5k



