

中山大学移动信息工程学院本科生实验报告

(2017 学年春季学期)

课程名称: Operating System

任课教师: 饶洋辉

批改人(此处为 TA 填写):

年级+班级	1506	专业(方向)	软件工程(移动信息工程)
学号	15352116	姓名	洪子淇
电话	13726205766	Email	1102229410@qq.com
开始日期	2017/3/31	完成日期	2017/4/12

1. 实验目的

- 1、通过修改 pintos 的线程休眠函数来保证 pintos 不会再一个线程休眠时忙等待。
- 2、通过修改 pintos 排队的方式来使得所有线程按优先级正确地被唤醒。

2. 实验过程

(一) Test 分析

1.alarm-single:

测试目的:

测试线程是否能够按照正确的顺序苏醒

过程分析:

开始分析测试代码实现之前, 我们先来复习测试线程的结构:

```
struct sleep_test
{
    int64_t start;           /*响应时间 */
    int iterations;         /* 休眠次数 */
    struct lock output_lock; /* 锁保护输出缓冲区 , 先被唤醒的线程进入缓冲区*/
    int *output_pos;        /* 缓冲区当前的位置 */
};
```

```
struct sleep_thread
{
    struct sleep_test *test;
    int id;                /* 睡眠 ID */
    int duration;          /* 休眠时间 */
    int iterations;        /* 每个线程的休眠次数 */
};
```

然后我们来看 alarm_single 的实现:

```
void
test_alarm_single(void)
{
    test_sleep(5, 1);
}
```

直接调用了 test_sleep(int thread_cnt, int iterations) 函数

参数意义: thread_cnt 表示线程个数, iterations 表示休眠次数

/* 创建线程 */

```
ASSERT (output != NULL);
```

```
for (i = 0; i < thread_cnt; i++)
```

```
{
```

```
    struct sleep_thread *t = threads + i; //t 表示线程, 通过 i 增加线程序数增加  
    char name[16];
```

```
    t->test = &test;
```

```
    t->id = i;      //第 i 个线程
```

```
    t->duration = (i + 1) * 10; //休眠持续时间, 根据线程顺序递增休眠时间
```

```
    t->iterations = 0; //初始化, 休眠次数为 0
```

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

```
    thread_create (name, PRI_DEFAULT, sleeper, t);
```

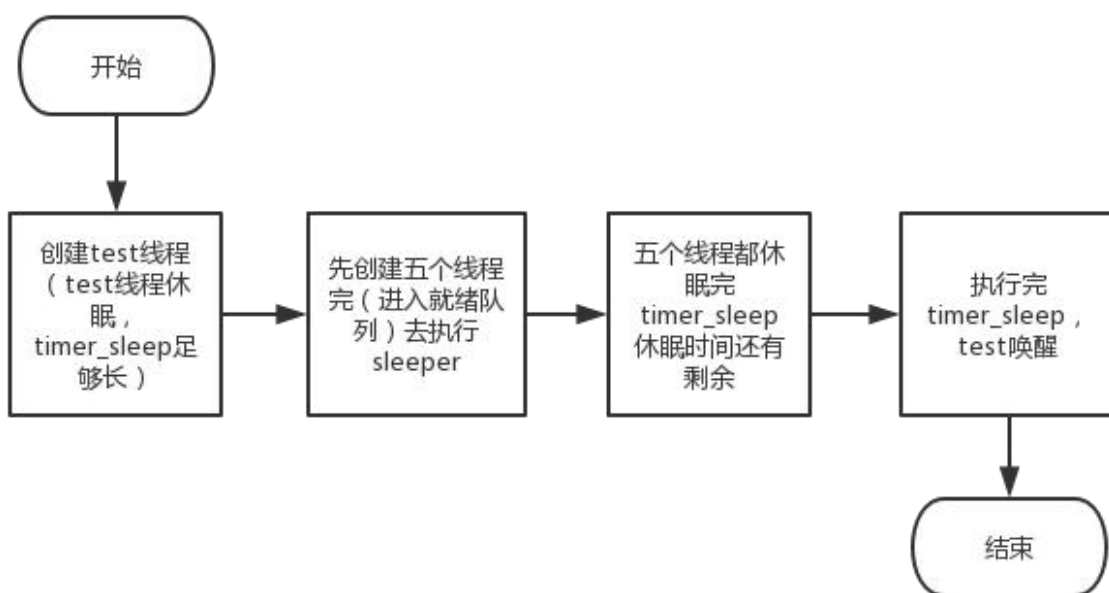
```
}
```

/* 等待足够长的时间使线程完成. */

```
timer_sleep (100 + thread_cnt * iterations * 10 + 100);
```

/* 我们可以看 thread 的参数表 thread_create (const char *name, int priority, thread_func *function, void *aux) , 可以知道每个线程的优先级是一样的, 只是休眠时间不一样。现在我们可以猜想这个函数是怎么实现的了, 因为优先级是相同的, 然后因为休眠时间不一样, 而且是递增的关系, 观测 timer_sleep 函数的休眠时间, 他足够长, 可是使所有的线程都被调度进入休眠, 当 test 里面所有的线程都被休眠了, 后来才唤醒 test。

流程图:



(总共有六个线程在休眠)

(思考: 如果休眠的时候同时有线程被唤醒, 然后整个流程又是怎样的呢? --即 multiple 是如何实现的。)

我们再来看 sleeper 函数, 验证我的猜想:

```
sleeper (void *t_)
{
    .....
    for (i = 1; i <= test->iterations; i++)
    {
        int64_t sleep_until = test->start + i * t->duration;
        timer_sleep (sleep_until - timer_ticks ()); //这一步是关键, 每个线程执行 timer_sleep
        lock_acquire (&test->output_lock); //获得锁
        *test->output_pos++ = t->id;
        lock_release (&test->output_lock); //释放锁, 锁的作用: 防止被中断
    }
}
/**符合猜想
*/
```

我们现在再看 test_sleep 函数下面做了些什么:

```
lock_acquire (&test.output_lock); /**获得输出锁
product = 0;
for (op = output; op < test.output_pos; op++)
{
    .....

    ASSERT (*op >= 0 && *op < thread_cnt); //断言了锁里面的个数和线程数一样
    t = threads + *op;
    new_prod = ++t->iterations * t->duration; //休眠时间就是他的 product
    msg ("thread %d: duration=%d, iteration=%d, product=%d",
        t->id, t->duration, t->iterations, new_prod);
    /*single 成功输出结果: 需要正确被唤醒, 才能保证正确输出*/
    if (new_prod >= product)
        product = new_prod;
    else
        fail ("thread %d woke up out of order (%d > %d)!",
            t->id, product, new_prod);
    /*如果 product < duration, 就是失败, 什么时候会小于呢?
    --当线程没有按照输入顺序正确被唤醒的时候, 例如 thread 2 先被唤醒, 接着 thread 0 再被
    唤醒, 那么此刻 30 < 10, 就会失败输出 fail () */
    .....
}

for (i = 0; i < thread_cnt; i++)
    if (threads[i].iterations != iterations) /* 验证我们是否正确的唤醒线程*/
        fail ("thread %d woke up %d times instead of %d",
            i, threads[i].iterations, iterations);
```

```
lock_release(&test.output_lock);/**解锁，保证输出不被中断
free(output);
free(threads);
```

结果分析：

```
Executing 'alarm-single':
(alarm-single) begin
(alarm-single) Creating 5 threads to sleep 1 times each.
(alarm-single) Thread 0 sleeps 10 ticks each time,
(alarm-single) thread 1 sleeps 20 ticks each time, and so on.
(alarm-single) If successful, product of iteration count and
(alarm-single) sleep duration will appear in nondescending order.
(alarm-single) thread 0: duration=10, iteration=1, product=10
(alarm-single) thread 1: duration=20, iteration=1, product=20
(alarm-single) thread 2: duration=30, iteration=1, product=30
(alarm-single) thread 3: duration=40, iteration=1, product=40
(alarm-single) thread 4: duration=50, iteration=1, product=50
(alarm-single) end
Execution of 'alarm-single' complete.

seven@15352116seven:~/pintos/src/threads/build$ =====
```

2.alarm-multiple:

测试目的：

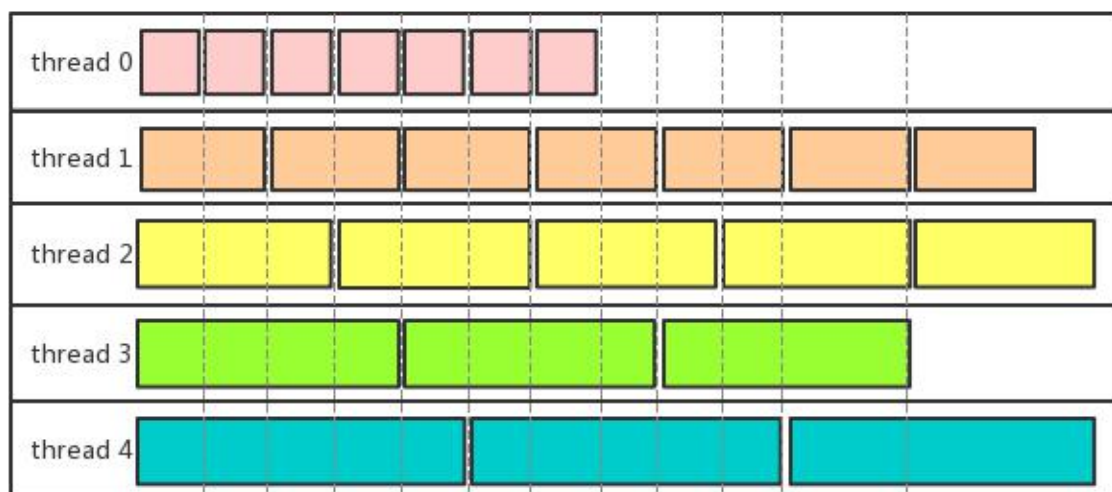
测试线程休眠次数为多个的时候，线程是否能够正确被唤醒；

过程分析：

```
void
test_alarm_single(void)
{
    test_sleep(5, 7);
}
```

代码和 single 一样，分析类似，主要分析被唤醒的顺序：

首先分析他们的休眠时间，以此来确定就绪队列里面的顺序：



有颜色的是休眠时间，每一个线程休眠之间被唤醒，唤醒之后进入就绪队列，最后再根据就绪队列里面的顺序进行唤醒，观察就绪队列里面的顺序可以通过观察休眠之间的唤醒间隔由上至下。

就绪队列顺序：0 0 1 0 2 0 1 3 0 4 0 1 2 0 1 3

结果分析:

通过被框住的线程,可以看出 thread 0 被唤醒的顺序,看是否和我们分析的队列顺序是否一致,观察是一致的,该测试成功。

```
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.

seven@15352116seven:~/pintos/src/threads/build$ =====
```

3.alarm-simultaneous

测试目的:

测试多个线程当他们的休眠时间相同时,是不是被同时唤醒。

过程分析:

首先我们来看一下 sleep_test 的结构:

```
struct sleep_test
{
    int64_t start;    //线程开始时间
```

```

int iterations;           //休眠次数
int *output_pos;          //缓冲区当前位置
}

```

接下来我们来看每个线程休眠的时间，sleeper 函数：

```

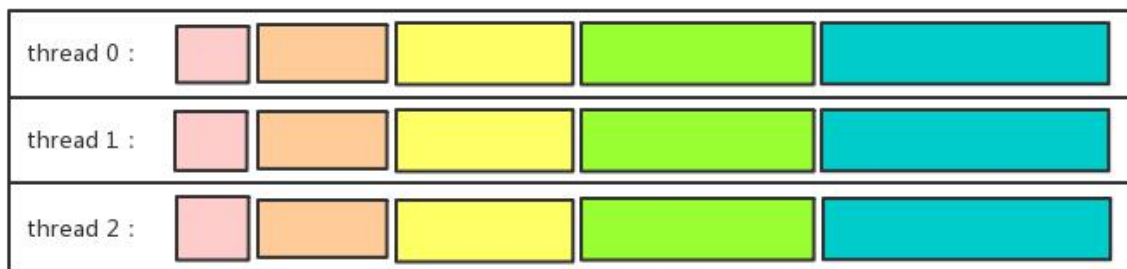
static void
sleeper (void *test_)
{
    struct sleep_test *test = test_;
    int i;

    timer_sleep (1);

    for (i = 1; i <= test->iterations; i++)
    {
        int64_t sleep_until = test->start + i * 10;
//从这里可以看出，每个线程休眠时间成递增关系：10 20 30 40 50
        timer_sleep (sleep_until - timer_ticks ());
        *test->output_pos++ = timer_ticks () - test->start;
        thread_yield ();
    }
}

```

流程图：



所以每一次休眠完成之后有三个线程被同时唤醒，又因为在 test_sleep 函数里面：

输出代码为：

```

for (i = 1; i < test.output_pos - output; i++)
    msg ("iteration %d, thread %d: woke up %d ticks later",
        i / thread_cnt, i % thread_cnt, output[i] - output[i - 1]);
/*

```

output[i] - output[i - 1]:后面一个休眠时间减去前面一个休眠时间，当 thread 0 第一次休眠完成之后，距离开始时间为 0，获得前一个 output = 10，先进入就绪队列，而 thread 1 休眠时间完成之后，此时的 output = 10，所以 thread 1 唤醒输出的是 10-10=0，同理 thread 2 输出 0，轮到第二次轮询，thread 0 第二次休眠时间为 20，减去前一个休眠时间 20-10=10，即输出 10.....这样也可以理解为 thread 0，thread 1，thread 2 是同时被唤醒的

*/

结果分析：

结果输出符合正确输出，所以该测试 pass


```

Executing 'alarm-simultaneous':
(alarm-simultaneous) begin
(alarm-simultaneous) Creating 3 threads to sleep 5 times each.
(alarm-simultaneous) Each thread sleeps 10 ticks each time.
(alarm-simultaneous) Within an iteration, all threads should wake up on
the same tick.
(alarm-simultaneous) iteration 0, thread 0: woke up after 10 ticks
(alarm-simultaneous) iteration 0, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 0, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 1, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 1, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 1, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 2, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 2, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 2, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 3, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 3, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 3, thread 2: woke up 0 ticks later
(alarm-simultaneous) iteration 4, thread 0: woke up 10 ticks later
(alarm-simultaneous) iteration 4, thread 1: woke up 0 ticks later
(alarm-simultaneous) iteration 4, thread 2: woke up 0 ticks later
(alarm-simultaneous) end
Execution of 'alarm-simultaneous' complete.
seven@15352116seven:~/pintos/src/threads/build$ =====

```

4.alarm-priority

测试目的:

要求线程调度是否是按照优先级高低进行调度的，当线程调度没有按照优先级调度的时候会失败。

过程分析:

首先我们来分析这个 test 做了些什么？

```

void
test_alarm_priority(void)
{
    ....
    wake_time = timer_ticks() + 5 * TIMER_FREQ; //test 的苏醒时刻
    sema_init(&wait_sema, 0);           //将 test 线程的 value 设置为 0

    for(i = 0; i < 10; i++)
    {
        int priority = PRI_DEFAULT - (i + 5) % 10 - 1; //PRI_DEFAULT 是最低优先级
        char name[16];
        snprintf(name, sizeof name, "priority %d", priority);
        thread_create(name, priority, alarm_priority_thread, NULL);
    }

    thread_set_priority(PRI_MIN);

    for(i = 0; i < 10; i++) //for 循环没有影响，只是为了把 test 线程阻塞掉。
        sema_down(&wait_sema); //创建完 10 个线程之后把 test 线程阻塞掉
}
/*

```

我们现在来观察 sema_down 函数:

```

void
sema_down(struct semaphore *sema)
{

```

```

.....
old_level = intr_disable ();
while (sema->value == 0)
{
    list_push_back (&sema->waiters, &thread_current ()->elem);
    thread_block ();
}

```

/*由于 test 线程的 value==0，所以 test 线程被阻塞掉，修改之前：然后第一个插入序列，也会第一个被唤醒直接输出 end 导致错误*/

```

sema->value--;
intr_set_level (old_level); //保证原子操作
}
*/

```

```

static void
alarm_priority_thread (void *aux UNUSED)
{
    int64_t start_time = timer_ticks ();
    while (timer_elapsed (start_time) == 0)
        continue;

```

/*防止检查时间的时候被中断，那么记录下的 tick 就会比当前时刻的多一，苏醒时刻就会延迟 1，那么就不是同步苏醒了*/

```

    timer_sleep (wake_time - timer_ticks ());
    //无论你在哪一刻休眠，都会在 wake_time 那一刻醒来
    msg ("Thread %s woke up.", thread_name ());
    //输出苏醒的线程

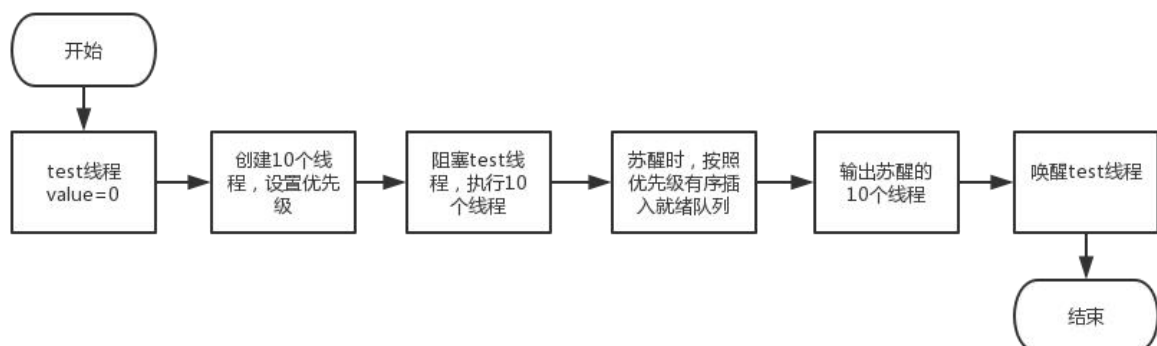
```

```

    sema_up (&wait_sema);
}

```

流程图：



结果分析：


```

Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.

seven@15352116seven:~/pintos/src/threads/build$ ==

```

5.alarm-zero

测试目的：

测试当休眠时间为 0 的时候，测试是否通过。

过程分析：

```

void
test_alarm_zero (void)
{
    timer_sleep (0);
    pass ();
}

```

直接调用了 timer_sleep 函数和一个 pass 函数，我们先来看一下这两个函数的实现：

```

void
timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();

    ASSERT (intr_get_level () == INTR_ON);
    while (timer_elapsed (start) < ticks)
        thread_yield ();
}

void
pass (void)
{
    printf ("%s) PASS\n", test_name);
}

```

/*当传入 ticks 值为 0 的时候，意思是该线程没有休眠时间，所以没有进入 while 循环里面，也就是并没有从 running 序列里面切换出来，这个线程一直在运行。

结束了 timer_sleep 函数之后，test_zero 调用了 pass 函数，直接输出 pass。*/

timer_sleep 函数修改之后：

```

void
timer_sleep (int64_t ticks)
{
    if(ticks <= 0)

```

```

{
    return;
}

```

/*这里判断休眠时间 ≤ 0 的情况，直接 return 结束 timer_sleep 函数，下一步调用 pass 函数，所以测试成功*/

```

ASSERT (intr_get_level () == INTR_ON);
enum intr_level old_level = intr_disable ();
struct thread *current_thread = thread_current ();
current_thread->ticks_blocked = ticks;    //the ticks the thread should sleep
thread_block ();
intr_set_level (old_level);

```

如果 timer_sleep 函数里面不加入对休眠时间的判断，那么每次调用中断的时候会调用到 thread_foreach 函数，thread_foreach 函数每次又会调用到 blocked_thread_check 函数，在 blocked_thread_check 函数里面：

```

if(t->status == THREAD_BLOCKED && t->ticks_blocked > 0)
{
    t->ticks_blocked--;
    if(t->ticks_blocked == 0)

```

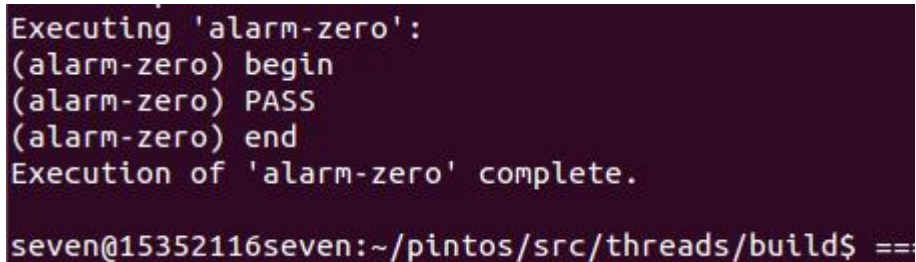
/*由于是负数，所以 ticks_blocked 永远不会等于 0，那么就永远不会被唤醒，而线程永久处于休眠状态*/

```

        {
            thread_unblock(t);
        }
    }
}

```

结果分析：



```

Executing 'alarm-zero':
(alarm-zero) begin
(alarm-zero) PASS
(alarm-zero) end
Execution of 'alarm-zero' complete.

seven@15352116seven:~/pintos/src/threads/build$ ==

```

6.alarm-negative

测试目的：

测试当休眠时间为负数的时候，测试时候通过。

过程分析：

```

void
test_alarm_negative (void)
{
    timer_sleep (-100);
    pass ();
}

```

/*代码和 zero 完全一样，只是输入 timer_sleep 的参数由 0 变成了一个负数。然后一样是不进入 while 循环，直接调用了 pass 函数，输出 pass。*/

而修改之后参照 zero 上面的代码分析

如果把传入的参数改为绝对值之后，那么就会进入 while 循环里面不断轮询，不过休眠时间到达了那个值得时候依旧会结束循环从而而调用 pass 函数，一样会通过。

结果分析：

```
Executing 'alarm-negative':
(alarm-negative) begin
(alarm-negative) PASS
(alarm-negative) end
Execution of 'alarm-negative' complete.

seven@15352116seven:~/pintos/src/threads/build$ ==
```

(二) 实验思路与代码分析

线程休眠问题：

thread_yield 函数只是把线程放进调度队列，然后切换线程，此时休眠线程状态是 ready，然后 running 下一个线程，但此时 timer_sleep 函数的 while 循环还在进行，就导致线程依然不断在 cpu 的 ready 队列和 running 队列之间来回，占用了 cpu 资源，这并不是我们想要的。

实验思路：

重新设计 timer_sleep 函数让休眠线程不再占用 cpu 时间，只在每次 tick 中断把时间交给操作系统时再检查睡眠时间，tick 内则把 cpu 时间让给别的线程。换句话说就是，不是调用 thread_yield 而是调用 thread_block 函数把线程 block 了，这样在 unblock 之前该线程都不会被调度执行。

代码分析：

1. 首先我们分析原 timer_sleep 函数的机制，这有助于我们如何重写 timer_sleep 函数

```
void
timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();

    ASSERT (intr_get_level () == INTR_ON);
    while (timer_elapsed (start) < ticks)
        thread_yield ();
}
```

我们来看第三行代码，这里调用了 timer_ticks 函数，那么这个函数是用来干嘛的呢？

```
int64_t
timer_ticks (void)          //获取当前 ticks 的返回值
{
    enum intr_level old_level = intr_disable ();
    int64_t t = ticks;
    intr_set_level (old_level);
    return t;
}
```

观察第三行和第五行代码可以发现他们明显是一对的关系，猜测应该是保证函数获取 ticks 不被中断，就是说保证这是一个原子操作。现在看一下 intr_disable 来验证猜测是否正确：

```
enum intr_level
intr_disable(void)
{
    enum intr_level old_level = intr_get_level();

    asm volatile ("cli" ::: "memory");
    //clear interrupt,将标志寄存器的 interrupt flag (IF) 位置为 0 , IF=0 时将不响应可屏蔽中断

    return old_level;
}
```

intr_disable(void) 返回值是一个 intr_level 型的(INTR_OFF/INTR_ON //表示中断关和开)，调用了 intr_get_level 且调用汇编指令保证了这个线程不能被中断。下面我们来看一下 intr_get_level 函数：

```
{
    asm volatile ("pushfl; popl %0" : "=g" (flags))
    //把标志寄存器的东西放到处理器栈上，然后把值 pop 到 flags (代表标志寄存器 IF 位) 上
    return flags & FLAG_IF ? INTR_ON : INTR_OFF;
    //通过判断 flags 来返回当前终端状态(intr_level)
}
```

总的来说就是 intr_disable 函数获取了当前的中断状态 (old_level = intr_get_level())，再将当前中断状态改为不能被中断 (通过汇编指令)，最后返回了中断前的中断状态 (return old_level)

那么在 timer_ticks 函数最后的 intr_set_level 函数有做了什么呢？

```
{
    return level == INTR_ON ? intr_enable() : intr_disable();
    //如果中断是开的 (ON) 就 intr_enable 否则就 intr_disable，返回的是之前的中断状态
}
```

理解了 disable 函数，我们来看一下 enable 函数实现了什么：

```
enum intr_level
intr_enable(void)
{
    enum intr_level old_level = intr_get_level();
    ASSERT(!intr_context());
    //断言这个中断不是硬中断而是软中断
    asm volatile ("sti"); //cli 指令的反面，将 IF 值设为 1
    return old_level;
    //和 intr_get_level 对应，返回了之前的中断状态
}
intr_context(void)
{
    return in_external_intr;
```


//返回了是否硬中断的标志

}

总而言之, timer_ticks 函数获取了 ticks 的当前返回值, 并且通过 get_level 和 set_level 来保证这个过程是不能被中断的。(我们获得了一个用法: 如何确保一个原子性的操作, 通过 get_level 和 set_level 函数)

分析完 timer_ticks, start 获取了起始时间, 然后断言必须可以被中断, 不然会一直死循环下去, 然后就是一个 while 循环, 调用了 thread_yield 函数, 这个函数只是把线程放进调度队列, 然后切换线程, 此时休眠线程状态是 ready, 然后 running 下一个线程, 但此时 timer_sleep 函数的 while 循环还在进行, 就导致线程依然不断在 cpu 的 ready 队列和 running 队列之间来回, 占用了 cpu 资源, 这并不是我们想要的。

2. 改写 timer_sleep 函数:

(1) Thread 结构体中新增成员变量 ticks_blocked (int); 记录线程剩余睡眠的时间(单位: tick)。

(2) 在线程被创建时将其初始化为 0。

---这里我遇到一个问题, 在 create 里面初始化会出错, 把初始化放在 init 里面就可以了
写完 sleep 函数后进行 make check:

```
20 of 27 tests failed.  
make: *** [check] Error 1  
seven@15352116seven:~/pintos/src/threads/build$
```

(3) 在 timer_sleep 函数中把 tick 赋值为睡眠时间。

```
enum intr_level old_level = intr_disable();  
struct thread *current_thread = thread_current();  
current_thread->ticks_blocked = ticks; //the ticks the thread should sleep  
thread_block();  
intr_set_level(old_level); //保证该操作是原子操作
```

(4) thread.c 中增加函数 blocked_thread_check 判断线程是否休眠完毕(ticks_blocked 的值), 并且在 thread.h 文件声明该函数。

```
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);  
        }  
    }  
}
```

(5) 处理 timer 中断时(timer_interrupt 函数)遍历所有线程(调用 thread.c 中的 thread_foreach 函数), 并判断被 block 线程是否睡眠完毕, 睡眠完毕则唤醒。

---thread_foreach(blocked_thread_check, NULL);

3. 休眠之后我们需要将程序唤醒, pintos 的基本实现是直接把线程放进 ready_list 尾部,

然后调度的时候是直接取头部的，是 **FIFO** 的调度方式，我们要改为根据优先级调度，那么保证插入 `ready_list` 的线程是有序的即可。可以通过已有的 `list_insert_ordered` 函数来实现有序插入。

首先我们来看一下 `list_insert_ordered` 函数的参数列表

```
void list_insert_ordered (struct list *, struct list_elem *, list_less_func *, void *aux);
typedef bool list_less_func (const struct list_elem *a, const struct list_elem *b, void *aux);
```

可以知道 `list_less_func *` 是一个 **函数指针**，所以我们需要新写一个比较优先级的函数 `cmp_priority`，那么进程如何获得优先级呢？首先让我们去了解一下进程的结构：

```
struct list_elem
{
    struct list_elem *prev;    /* Previous list element. */
    struct list_elem *next;    /* Next list element. */
};
struct list
{
    struct list_elem head;     /* List head. */
    struct list_elem tail;     /* List tail. */
};
```

`list` 中存放的数据类型是 `list_elem`，存放待执行线程靠的就是 `list`。

```
struct thread
{
    .....
    int priority;
    struct list_elem elem;
    .....
}
```

有一个 `list` 变量 `ready_list` 专门存放待执行的线程，由于 `list` 存放的是 `list_elem` 类型，因此 `thread` 中有一个成员变量专门用于在 `ready_list` 中“排队”。既然在 `ready_list` 中并不是 `thread` 而只是 `thread` 的一个成员，那么这个成员如何去访问他的优先级呢？---`List` 中已经提供了一个宏定义解决这个问题。

```
#define list_entry(LIST_ELEM, STRUCT, MEMBER)
    ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next
                  - offsetof(STRUCT, MEMBER.next)))
```

所以我们在写 `cmp_priority` 函数的时候直接比较线程的优先级即可：

```
-return list_entry(a, struct thread, elem)->priority > list_entry(b, struct thread, elem)->priority;
```

写完优先级函数之后我们需要把之前涉及到 `list_push_back` 函数插入 `read_list` 序列的尾部修改为 `list_insert_ordered` 函数，那么我们在什么时候会把一个线程丢到就绪队列呢？

1. `thread_unblock` // 唤醒的时候

2. `init_thread` // 初始化的时候

3. `thread_yield` // 线程切换的时候

修改完之后，现在我们已经完成了休眠函数的更改以及根据优先级进行调度。

3. 实验结果

Alarm_priority 测试通过了:

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
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
19 of 27 tests failed.
make: *** [check] Error 1
seven@15352116seven:~/pintos/src/threads/build$
```

4. 回答问题

- 1. 对于 alarm-single, alarm-multiple, alarm-simultaneous 这三个相似的测试来说, 在改之前虽然由于忙等待占用了 CPU 的资源, 但是休眠时间的计时是没有问题的, 正确唤醒休眠的函数还是可以保证的。

对于 alarm-zero 和 alarm-negative 这两个测试来说, 对于小于等于 0 的值来说, 在休眠函数中, while 循环是不会执行的, 所以在测试中会立即执行 pass 函数输出 PASS, 所以这两个测试也是可以通过的。

详细分析可以参考第三点中的 test 分析

- 2. 中断需要打开是因为操作系统是通过中断来获得 CPU 时间, 打开后可以及时地更新 CPU 时间, 然后通过不断轮询检查经过时间是否达到 ticks, 若还没达到则调用 thread_yield 函数, 达到了 ticks 就会结束休眠。若中断没有打开, 则 CPU 时间没有及时更新, 即使达到了 ticks, 可是也没有结束休眠。中断打开是为了保证可以及时地结束进程的休眠状态而, 修改之后的 sleep 函数的中断也是需要打开的, 中断打开 (获取当前 CPU 时间) -> 中断关闭 (保证原子操作, 把休眠线程阻塞掉) -> 恢复中断的开启状态
- 3. 中断: 指 CPU 在正常运行程序的过程中, 由于预选安排或发生了各种随机的内部或外部事件, 使 CPU 中断正在运行的程序, 而转为相应的服务程序去处理。当出发中断相应后, CPU 会关闭中断, 并且保存现场, 直到完成后, 才会开放中断, 返回到原来被中断的主程序下。中断避免了把大部分时间浪费在等待上查询的操作上。轮询: 是 CPU 决策如何提供周边设备服务的方式, CPU 会定时发出询问, 依序询问每一个周边设备是否需要其服务。如果有, 就马上给予服务, 服务结束后再问下一个周边设备, 如此不断周而复始下去。缺点是效率低下, 开销很大, 等待时间很长, CPU 利用率不高。

5. 实验感想

1. 一开始在看 sleep 函数的时候一遇到新的函数就直接往下看, 然后新的函数又会涉及到新的内容, 如此并不方便我们去了解一开始那个函数, 所以在看函数的时候首先看懂注释, 它是实现些什么的, 如何进行实现, 一开始对底层不熟悉的时候, 掌握用法以及理解函数的意义比无头脑地去翻看一个函数的实现有意义的多。就好比宏定义, 当不能往下找到更

深刻的东西之后，只能先学会去用，毕竟我们根据注释已经可以得知他是允许了 thread 内部变量之间的相互访问。

2.在分析 test 中有一个很大的误区，举个例子来说就是 test_priority 函数里面：

```
for (i = 0; i < 10; i++)
{
    .....
    thread_create (name, priority, alarm_priority_thread, NULL);
}
....
for (i = 0; i < 10; i++)
    sema_down (&wait_sema); //创建完 10 个线程之后把 test 线程阻塞掉
```

一开始总以为在 create 里面不仅是创建了线程，还会调用 alarm_priority 函数去执行，然后才会把 test 线程 block 掉，导致了很久都没能正确理解整个过程。其实 create 函数只是创建，创建之后加入就绪队列，而不是马上执行。然后再 block 掉 test 线程，让测试样例执行完

参考资料：

http://www.cnblogs.com/laiy/p/pintos_project1_thread.html