

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	1501	专业 (方向)	移动信息工程
学号	15352006	姓名	蔡丽芝
电话	13538489980	Email	314749816@qq.com
开始日期	2017/4/5	完成日期	2017/4/13

1. 实验目的 (线程的休眠与唤醒)

- 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:

一共会产生 5 个线程, 每个线程都会休眠一段固定不同的时间, 会休眠 1 次。如果实验测试成功的话, 每个线程的休眠 duration 和 product 会按以公差为 10 的递增数列逐渐增大。

过程分析:

首先在 test 中这条测试的执行代码为

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

test_alarm_single 函数无返回值, 为 void, 其中调用了 test_sleep 函数, 需要传入两个参数, 第一个参数表示线程的个数, 第二个参数表示线程休眠的迭代次数。

test_sleep 函数分析:

首先声明了两个结构体的指针 test, threads

```
struct sleep_test test;
struct sleep_thread *threads;
```

其中两个结构体的具体代码如下

```
/* Information about the test. */
struct sleep_test
{
    int64_t start;           /* Current time at start of test. */
    int iterations;         /* Number of iterations per thread. */

    /* Output. */
    struct lock output_lock; /* Lock protecting output buffer. */
    int *output_pos;         /* Current position in output buffer. */
};
```

sleep_test 中会储存当前测试的开始时间和每个线程睡眠的迭代次数, 创建并保存了一个锁。因此在 test_sleep 函数中 test 变量保存了整个测试过程的信息, 包括测试的开始时间, 休眠的迭代次数。

```

/* Information about an individual thread in the test. */
struct sleep_thread
{
    struct sleep_test *test;    /* Info shared between all threads. */
    int id;                    /* Sleeper ID. */
    int duration;              /* Number of ticks to sleep. */
    int iterations;            /* Iterations counted so far. */
};

```

sleep_thread 结构体中储存着该测试中每一个独立的需要休眠的线程的信息。包括 sleeper id,休眠时间和剩余的休眠迭代次数。

回到 test_sleep 函数：

```

/* Allocate memory. */
threads = malloc (sizeof *threads * thread_cnt);
output = malloc (sizeof *output * iterations * thread_cnt * 2);
if (threads == NULL || output == NULL)
    PANIC ("couldn't allocate memory for test");

```

分别给声明的指针 threads 和 output 分配空间。

```

/* Initialize test. */
test.start = timer_ticks () + 100;
test.iterations = iterations;
lock_init (&test.output_lock);
test.output_pos = output;

```

进行初始化，100 即 1 秒，test 的开始时间在当前时间再加上 1s，目的是为了等足够长的时间，因为其他的操作也需要一些时间。如 lock_init 等。

```

/* Start threads. */
ASSERT (output != NULL);
for (i = 0; i < thread_cnt; i++)
{
    struct sleep_thread *t = threads + i;
    char name[16];

    t->test = &test;
    t->id = i;
    t->duration = (i + 1) * 10;
    t->iterations = 0;

    snprintf (name, sizeof name, "thread %d", i);
    thread_create (name, PRI_DEFAULT, sleeper, t);
}

```

```

/* Wait long enough for all the threads to finish. */
timer_sleep (100 + thread_cnt * iterations * 10 + 100);

```

开始线程，对每个线程的结构体内部信息进行初始化，赋值操作。

sprintf(name, sizeof name, "thread %d", i)：将字符按照 thread i 的格式输出到 name 中。

thread_create(name, PRI_DEFAULT, sleeper, t)：产生一个名为 name 的内核线程，它的优先级是默认的，这个线程会执行 sleeper 函数，同时 thread_create 函数还会将这个线程添加到就绪队列中。

timer_sleep(100 + thread_cnt * iterations * 10 + 100)：其等待时间为所有线程休眠完的时间再加上 test.start 多加上的 2s，目的是等待足够长的时间使所有的线程都完成。

再看涉及到的 sleep 函数：

```

/* Sleeper thread. */
static void
sleeper (void *t_)
{
    struct sleep_thread *t = t_;
    struct sleep_test *test = t->test;
    int i;

    for (i = 1; i <= test->iterations; i++)
    {
        int64_t sleep_until = test->start + i * t->duration;
        timer_sleep (sleep_until - timer_ticks ());
        lock_acquire (&test->output_lock);
        *test->output_pos++ = t->id;
        lock_release (&test->output_lock);
    }
}

```

由于 alarm-single 的迭代次数是 1，所以这个线程在创建的时候只执行一次这个函数。

```

/* Print completion order|. */
product = 0;
for (op = output; op < test.output_pos; op++)
{
    struct sleep_thread *t;
    int new_prod;

    ASSERT (*op >= 0 && *op < thread_cnt);
    t = threads + *op;

    new_prod = ++t->iterations * t->duration;

    msg ("thread %d: duration=%d, iteration=%d, product=%d",
        t->id, t->duration, t->iterations, new_prod);

    if (new_prod >= product)
        product = new_prod;
    else
        fail ("thread %d woke up out of order (%d > %d)!",
            t->id, product, new_prod);
}

```

product 记录的实际上是线程最终的睡眠结束的时间，对于这个 test 中线程的睡眠时间===最终睡醒时间，因此会有 duration == product

结果分析：

```

simmon@15352006lizhica: ~/pintos/src/threads/build
=====
Pilo hda1
Loading.....
Kernel command line: run alarm-single
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
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.

```

在测试中，每个线程按照 alarm-single 函数中的规则休眠 10(i+1)个 ticks，休眠的迭代次数为 1。因此休眠依次被唤醒，唤醒时间 product 都和它们的 duration 相等。

② alarm-multiple

```

void
test_alarm_multiple (void)
{
    test_sleep (5, 7);
}

```

test_alarm_multiple 函数调用了 test_sleep(5, 7)，表示创建了 5 个线程，休眠时间是 10, 20, 30, 40, 50，并且每个线程会迭代 7 次。test_sleep 的分析不再赘述。

thread 0: 休眠时间为 10ticks，一共会迭代 7 次，因此：第一次休眠结束时间是 10ticks，接着又进入睡眠，第二次休眠结束时间是 20ticks，一次到第七次休眠结束时间是 70ticks，product = 10*7

threads 1: duration 为 20ticks，第一次睡醒时间 20ticks，第二次睡醒时间 40ticks，。。，第七次睡醒时间 20*7，product = 20*7;

同理可得 thread 4, product = 50*7 = 350;

结果分析:

```
simmon@15352006lizhcai: ~/pintos/src/threads/build
(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
```

最终 product = 350, 符合前面的分析

③ alarm-simultaneous

测试目的: 用来测试线程能不能在正确的时间被唤醒。

过程分析:

```
void
test_alarm_simultaneous(void)
{
    test_sleep(3, 5);
}
```

调用 test_sleep(3,5)函数, 表示创建 3 个线程, 每个线程休眠迭代 5 次, 但与上述 test_sleep()不同的是, 这里每个线程的休眠时间都是 10ticks

```
for (i = 0; i < thread_cnt; i++)
{
    char name[16];
    snprintf(name, sizeof name, "thread %d", i);
    thread_create(name, PRI_DEFAULT, sleeper, &test);
}

/* Wait long enough for all the threads to finish. */
timer_sleep(100 + iterations * 10 + 100);

/* Print completion order. */
msg("iteration 0, thread 0: woke up after %d ticks", output[0]);
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]);
```

thread 0: 在 10ticks 后被唤醒, current time = 10ticks, old time = 0 ticks, 所以时间差为 10, 时间差被写入数组 output, 所以 woke time = 10ticks; thread1 和 thread2 休眠时间为 10ticks, 但被唤醒时间也为 10ticks, 所以 woke time = 0, 同理可推出后面的唤醒时间。

结果分析:


```

simmon@15352006lizhical: ~/pintos/src/threads/build
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.

```

与分析结果一致

④ alarm-priority

测试目的：测试能否在唤醒后进行优先级调度

过程分析：

```

static void
alarm_priority_thread (void *aux UNUSED)
{
    /* Busy-wait until the current time changes. */
    int64_t start_time = timer_ticks ();
    while (timer_elapsed (start_time) == 0)
        continue;

    /* Now we know we're at the very beginning of a timer tick, so
       we can call timer_sleep() without worrying about races
       between checking the time and a timer interrupt. */
    timer_sleep (wake_time - timer_ticks ());

    /* Print a message on wake-up. */
    msg ("Thread %s woke up.", thread_name ());

    sema_up (&wait_sema);
}

```

表示了 PV 操作中的 V 操作，其表示的含义是优先级高的先唤醒，优先级低的后唤醒。

```

void
test_alarm_priority (void)
{
    int i;

    /* This test does not work with the MLFQS. */
    ASSERT (!thread_mlfqs);

    wake_time = timer_ticks () + 5 * TIMER_FREQ;
    sema_init (&wait_sema, 0);

    for (i = 0; i < 10; i++)
    {
        int priority = PRI_DEFAULT - (i + 5) % 10 - 1;
        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++)
        sema_down (&wait_sema);
}

```

wake_time = timer_ticks() + 5 * TIMER_FREQ: 表示的是每个线程的睡眠时间是 500ticks

Int priority = PRI_DEFAULT - (i+5)%10 -1: 在这行代码的后面，创建线程的时候，将 priority 传入线程创建函数，给不同的线程分配不同的优先级。

结果分析：

```

simmon@15352006lizhcai: ~/pintos/src/threads/build
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: run alarm-priority
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
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.

```

线程唤醒是按照优先级进行唤醒的。

⑤ alarm-zero

测试目的：测试当传入的 ticks 时，线程是否会进行休眠。

过程分析：

```

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

```

在 timer_sleep(0) 中

```

// ASSERT (intr_get_level (
if (ticks <= 0)
{
    return;
}

```

当 ticks=0, 函数会返回 return; 休眠不会执行。

结果分析：

```

simmon@15352006lizhcai: ~/pintos/src/threads/build
=====
Bochs x86 Emulator
Built from SVN snapshot d
Compiled on Mar 23 2017
=====
00000000000i[      ] reading configuration
00000000000e[      ] bochsrc.txt:8: 'user_
oard' option.
00000000000i[      ] installing nogui modu
00000000000i[      ] using log file bochs
PiLo hda1
Loading.....
Kernel command line: run alarm-zero
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'alarm-zero':
(alarm-zero) begin
(alarm-zero) PASS
(alarm-zero) end
Execution of 'alarm-zero' complete.

```

分析后，不会进行休眠

⑥ alarm-negative

测试目的：测试当 $ticks < 0$ 时，线程不进行休眠。

过程分析：与 alarm-zero, 原理相同，此处略。

结果分析：

```
simmon@15352006lizhical: ~/pintos/src/threads/build
=====
Bochs x86 Emulator 2.6.2
Built from SVN snapshot on May 26, 2013
Compiled on Mar 23 2017 at 08:08:46
=====
00000000000i[      ] reading configuration from bochsrc.tx
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will b
oard' option.
00000000000i[      ] installing nogui module as the Bochs
00000000000i[      ] using log file bochsout.txt
Pilo hda1
Loading.....
Kernel command line: run alarm-negative
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'alarm-negative':
(alarm-negative) begin
(alarm-negative) PASS
(alarm-negative) end
Execution of 'alarm-negative' complete.
```

分析后，不会进行休眠。

（二）实验思路与代码分析

未做任何改动时 timer_sleep 代码

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

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

int64_t ticks: timer_sleep() 传入参数 ticks, 其表示的是该线程休眠等待的时间周期的的大小。

start = timer_ticks(): 调用了 timer_ticks 函数，并将该函数的返回值赋值给变量 start。

timer_ticks 函数返回的是系统的开始时间。因此: start 表示开始启动系统到当前的系统时间。

ASSERT(intr_get_level() == INTR_ON): 断言系统的中断机制是打开的，确保线程休眠的时候中断是打开的。因为将一个线程从运行队列切换到就绪队列中是通过 CPU 调度，需要中断的。

while(timer_elapsed(start) < ticks): 当系统的流逝时间小于线程所需的休眠时间，则会进入到循环中，执行循环中的代码，直到不满足上述条件。

thread_yield(): 将当前线程放进就绪队列中，并调用 schedule() 函数，调度下一个线程。

```
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_push_back (&ready_list, &cur->elem);
    cur->status = THREAD_READY;
    schedule ();
    intr_set_level (old_level);
}
```

old_level = intr_disable(): intr_disable() 将当前线程的中断状态赋值给 old_level, 并且关闭中断

Intr_set_level(old_level): 重新恢复之前的中断状态，在这两行代码之间的代码的中断被关闭，可以保证原子操作。

在上述两行代码的中间的代码: 改变了当前线程的状态，从运行状态切换到了就绪状态，并将当前线程放入到就绪队列中，调用 schedule 函数，调度下一个线程。

存在问题：timer_sleep 函数的作用是使线程休眠相应的时间，正确的理解是，当线程进行休眠的时候，该线程就和 CPU 没有任何关系，调度程序会调度一个新的线程进入 cpu。然而在原有的代码中，当休眠时间还没有结束的时候，当前线程不断地在 cpu 的 ready 队列和 running 队列之间来回，占用了 cpu 资源。因为操作系统是直接从队列的头部获取线程运行，而当前线程会被仍会 ready 队列的头部，因此当调用 schedule 的时候，下次执行的线程仍然是该线程，是忙等待，耗费了资源。

解决方法：（参考 TA 给的 ppt 思路）

通过重新设计 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，重写 timer_sleep 函数，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。

```
int ticks_blocked; /* record the rest time of sleep time*/
```

在 thread 结构体中新增成员变量 ticks_blocked，用于记录线程剩余睡眠的时间。

```
/*Initialize tics_blocked*/
t->ticks_blocked = 0;
```

在线程被创建时将其初始化为 0。

```
/* check the thread if it is time to end of the sleeping time.*/
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);
        }
    }
}
```

用于判断线程是否休眠完毕，当休眠时间到了，就停止阻塞。同时要将这个函数的声明加到对应的 thread.h 文件上。

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();
    enum intr_level old_level = intr_disable();
    thread_foreach(blocked_thread_check, NULL);
    intr_set_level(old_level);
}
```

在每个 timer 中断的时候，调用 thread_foreach 函数来遍历所有线程，并传入 blocked_thread_check，来判断每个线程是否休眠完毕，如果睡眠完毕则唤醒。


```

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

    ASSERT (intr_get_level () == INTR_ON);
    while (timer_elapsed (start) < ticks)
        thread_yield ();
    */
    // ASSERT (intr_get_level () == INTR_ON);
    if (ticks <= 0)
    {
        return;
    }
    else
    {
        enum intr_level old_level = intr_disable();    /* get the old interrupt state and off the interrupt*/
        struct thread *current_thread = thread_current(); /* get the current thread */
        current_thread->ticks_blocked = ticks;          /* set the ticks_blocked */
        thread_block();                                /* block the thread */
        intr_set_level(old_level);                      /* restore the old interrupt state*/
    }
}

```

首先判断 ticks 是否小于 0，当 ticks 小于等于 0 的时候，timer_sleep 函数直接 return；因为 ticks<=0 的时候，当前线程不进行休眠，但是如果没有这个判断条件，若 ticks<0，该线程会直接永远被阻塞，从而进行无穷的休眠，ticks=0 时，说明不需要休眠，会进行一些重复无用的操作。

唤醒：

未改变代码之前存在的问题：

alarm-priority 要求线程是根据优先级进行调度的，而 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，所以是 FIFO 的调度方式，这样会导致一种现象，如果有一个线程的优先级很高，但是休眠以后却要排在就绪队列的末尾，等待前面优先级低的线程，会导致一些紧急的线程无法优先运行。

解决方法：（参考 TA 给的 ppt）

基本思路：保证插入到 ready_list 中线程是有序的，因此我们必须先找到那些地方会把一个线程插入到就绪队列，分析代码可知：在 thread_unblock, init_thread, thread_yield 这三个地方会将线程插入到就绪队列中，因此需要在这些地方将线程进行排序。

```

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 elem;     /* List element for all threads list.
                               /* record the rest time of sleep time

    /* 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. */
};

```

struct list_elem elem: 因为线程很忙，所以线程会自己其中一个成员去负责排队和同步。而有一个 list 变量 ready_list 专门用来存放执行的线程，由于 list 存放的是 list_elem，因此 thread 中有一个成员变量专门用于在 ready_list 中排队，也就是在 thread 结构体中声明的 elem。

```

/* Converts pointer to list element LIST_ELEM into a pointer to
the structure that LIST_ELEM is embedded inside. Supply the
name of the outer structure STRUCT and the member name MEMBER
of the list element. See the big comment at the top of the
file for an example. */
#define list_entry(LIST_ELEM, STRUCT, MEMBER) \
((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next \
- offsetof (STRUCT, MEMBER.next)))

```

(来自 TA 给的 PPT)

这里实现了宏定义，可以将它当作函数使用，它的功能是从结构体(STRUCT)某成员变量(MEMBER)利用指针(LIST_ELEM)来求出该结构体的首指针。比如说当前 struct list_elem 类型的指针为 p，那么 LIST_ELEM 就是 &p，STRUCT 就是我们的目标类型 struct thread，MEMBER 就是 struct thread 中 element 成员的名字，就是 elem。

利用已有的 list_insert_ordered 函数进行有序的排序(参考 ppt)

在 list.h 中

```
void list_insert_ordered (struct list *, struct list_elem *, list_less_func *, void *aux);
```

存在上面已有的函数，我们可以用将线程有序地插入就绪队列中。

```

/* compare function */
bool
compare_thread_priority (const struct list_elem *i, const struct list_elem *j, void *aux UNUSED)
{
    return list_entry(i, struct thread, elem)->priority > list_entry(j, struct thread, elem)->priority;
}

```

利用宏定义的 list_entry 找到 i, j 对应的线程结构体的首指针，比较其优先级。由于在这个过程中用到了 struct list 中的成员变量 elem，因此这个比较函数必须声明定义在 thread.h 中，在 thread.c 中实现。

thread_unblock 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_push_back (&ready_list, &t->elem); */
    list_insert_ordered (&ready_list, &t->elem, (list_less_func *)&compare_thread_priority, NULL);
    t->status = THREAD_READY;
    intr_set_level (old_level);
}

```

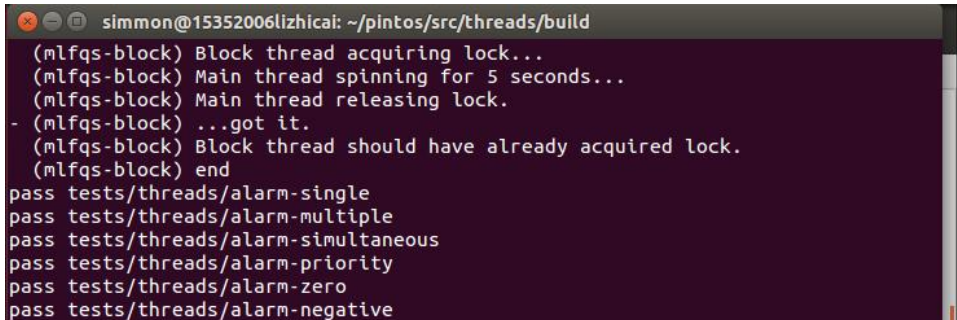
将原来的 `list_insert` 函数改为用 `list_insert_ordered`, 并将 `compare_thread_priority` 传入函数中同理将 `thread_yield` 中的 `list_insert` 函数改成如下:

```
if (cur != idle_thread)
    list_insert_ordered (&ready_list, &cur->elem, (list_less_func *)&compare_thread_priority, NULL);
```

将 `init_thread` 中的 `list_insert` 函数改成如下:

```
list_insert_ordered (&all_list, &t->allelem, (list_less_func *)&compare_thread_priority, NULL);
```

3. 实验结果



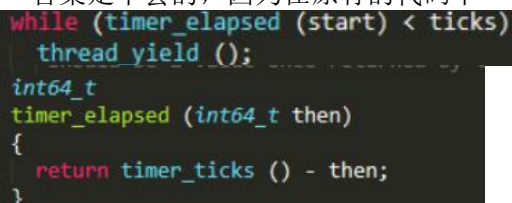
```
simmon@15352006lizhical: ~/pintos/src/threads/build
(mlfqs-block) Block thread acquiring lock...
(mlfqs-block) Main thread spinning for 5 seconds...
(mlfqs-block) Main thread releasing lock.
- (mlfqs-block) ...got it.
(mlfqs-block) Block thread should have already acquired lock.
(mlfqs-block) end
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
```

```
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
simmon@15352006lizhical:~/pintos/src/threads/build$
```

4. 回答问题

1. 之前为什么 20/27? 为什么那几个 test 在什么都没有修改的时候还过了?

答: 原来的 `pintos` 休眠机制是通过不断轮询检查是否到达休眠时间来实现的, 虽然这样导致了线程在 `running` 和 `ready` 队列中不断来回, 占用了 `cpu` 资源, 但是还是一样实现了休眠唤醒的功能。因此 `alarm-single`, `alarm-simultaneous` 和 `alarm-multiple` 还是可以正常 `pass`。`alarm-zero`, `alarm-negative` 主要和传入的参数 `ticks` 有关, 当 `ticks` 即休眠时间 ≤ 0 时, 线程是否会休眠呢, 答案是不会的, 因为在原有的代码中



```
while (timer_elapsed (start) < ticks)
    thread_yield ();

int64_t
timer_elapsed (int64_t then)
{
    return timer_ticks () - then;
}
```

明显 `timer_elapsed` 的返回值永远都大于 0, 则循环条件不成立, 并不会休眠, 所以这个 `test pass`

2. 为什么休眠的时候要保证中断打开?

答: 因为休眠的时候需要涉及到线程状态的切换, 线程需从运行状态切换到就绪状态, 这个过程是需要中断的。

3. 一道经典面试题: 中断和轮询的区别? (参考博客)

答: 轮询: `polling`, 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。

中断: `interrupt`, 是当出现一个必须由 `CPU` 立即处理的情况, `CPU` 暂时停止当前程序的执行转而

执行处理新情况的程序和执行过程。当触发中断相应后，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，使得效率大大的提高了。

5. 实验感想

在本次试验中一开始就出现了问题，因为在第一次配置 pintos 的时候，就出现了错误，我由于粗心大意，忽略了 pass 样例结果应该为 20/27，然而我的是 21/27，因此出现了巨大的错误。其次，由于想到备份版本回退的问题，我一开始使用了 git 来存自己的 pintos，由于在 make check 过程中出现了很多错误，但是不知道原来是因为用 git 才出现了问题，做了很多无用功，浪费了很多时间。接下来就是改代码，在改代码的过程中，通过 TA 给的 ppt 思路还是比较容易的写出，过程中主要遇到的问题是没看懂 make check 过程中出现的问题，导致入了很多坑。最后在寻求他人帮助的条件下，解决了我的问题。我认为这一次实验中改代码相对于分析 tests 还是简单的，因为只要理解了思路即可，但是真正难的是分析 test，test 中涉及到了许多知识点，比如锁，pv 操作，虽然这些老师上课都有提及过，但是自己并没有很好的掌握，所以一开始看的时候都不知道代码到底是讲什么东西，最难的是这些函数里有很多嵌套函数，所以有时候要跳来跳去，弄的头很大。而且有些代码比较复杂又多，自己没什么耐性，所以分析了很久，尤其是分析 alarm-single 函数的时候遇到了很多困难。但是分析完一些代码后，对 pintos 的休眠机制和优先级调度有了很深刻的理解。但是说实话 pintos 里的有些代码我还是看不懂。