目录

1. 实验目的

线程休眠与唤醒

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。(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。)

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) 机制分析

1. 系统时间

每隔 TIMER_FREQ 就会通过中断，获取当前 CPU 时间。中断频率是 10ms。通过一开始的 timer_init 函数设置定时器和注册中断。中断时 timer_interrupt 会被调用，调用之后完成系统时间 tick+1。

```
/* Number of timer interrupts per second. */
#define TIMER_FREQ 100
```

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

2. 中断机制

pintos 的中断机制结果如右图

```
enum intr_level old_level = intr_disable ();
/* do something */
intr_set_level (old_level);
```

intr_level 结构体是中断开关判断能否被中断。intr_disable 中做了两件事通过 intr_get_level 获得中断开关的值并返回保存在 old_level 中，同时使用确保这个过程不会被中断的汇编（他的翻译是这样的..）。之后的 intr_set_level 就是在恢复之前中断开关的值。

3. 休眠机制

首先，start 记录了进入这个函数的当前时间。之后利用 timer_elapsed () 判断当前时间和 start 的差值，就是已休眠时间 判断当前已休眠时间的时间是否超过给定的 ticks，如果没有，执行 thread_yield ()。

```
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);
}
```

所以现在分析 thread_yield ()，其获使用 thread_current 获取现在 running 的线程后关中断。然后判断其不是空闲进程之后直接将其放入 ready 队列尾部，并将线程状态改为 ready。之后执行 schedule，再恢复中断。

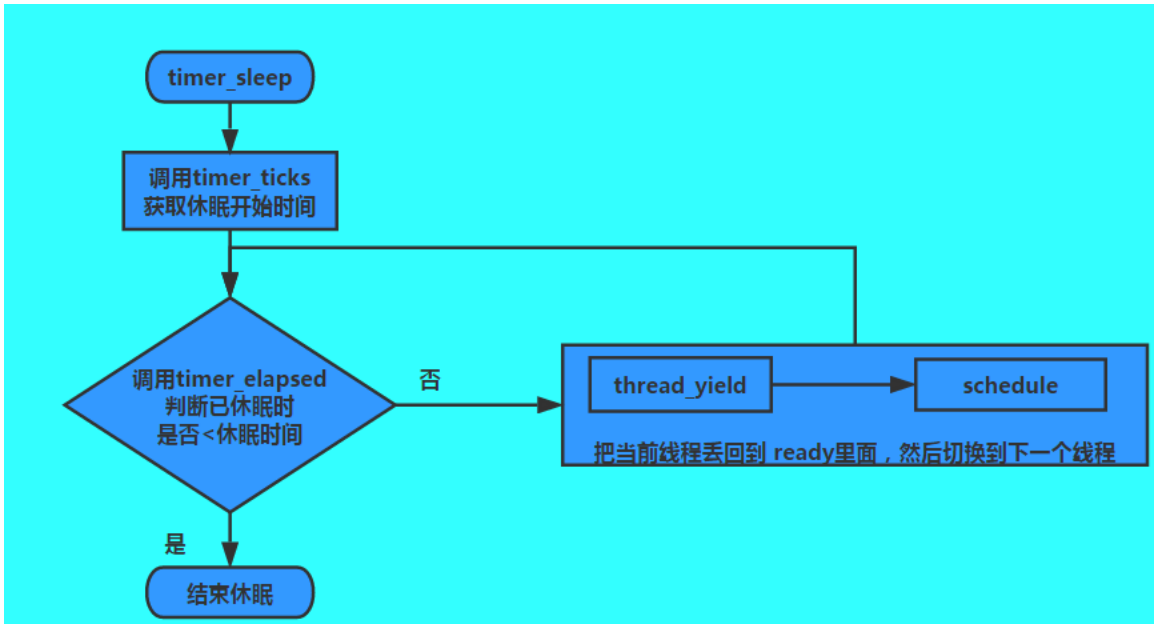
所以现在需要分析 schedule，running_thread 获得当前进程，next_thread_to_run 获得 ready 队列头进程，如果 ready 为空获得的 NULL。后面的内容函数打开已经看不懂了，但是从其注释上来看就是对于线程的调度。就是当前进程切换到了下一个线程。如果 ready 队列为空就还是调度当前线程。

```
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

Timer_sleep 流程如下



总结一下 timer_sleep 就是在循环里面不断把线程丢回 ready 队列，然后再被调度算法调度回 running 队列，不让它执行来实现休眠。这样虽然线程休眠了，但 CPU 也不能空闲，出于忙等待状态忙等待。此处为什么休眠时要保证中断开关打开，主要在于 CPU 不可能让我们一直处理休眠的线程调度问题，当出现更高优先级事件的时候必须去处理，唤醒机制就不在解释了休眠结束后直接放入 ready 队列。

(二) Test 分析

1. Test：alarm-single & alarm-multiple

1. 测试目的

a) 测试线程能否休眠，能否被按照休眠时间唤醒。

b) Single 测试为测试单次线程单次休眠唤醒。

c) Multiple 中还增加测试线程能否多次休眠唤醒。

2. 过程分析

test_alarm_single 与 test_alarm_multiple 函数都位于 alarm-wait.c 中。

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

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

其调用了`test_sleep (int thread_cnt, int iterations)`传入了两个参数：创建线程的数量和休眠迭代的次数。alarm_single 只需要进行一次休眠所以迭代次数为 1，alarm_multiple 进行 7 次休眠。test_sleep 分析如下：

```
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);
  }
```

首先使用一个循环创建线程。这里我们可以看到这里规定了线程的休眠时间等信息。在此处除了线程的休眠时间不同外，线程运行的函数和优先级都是一样的。

```
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);
  }
```

之后我们看一下作为线程创建时传入的 sleeper 中的部分，可以看到这个函数让当前线程休眠了 iterations 次。而在 timer_sleep 中 sleep_until 随着 iterations 和 duration 的增大而增大。确保了每次休眠时间都是 duration。储存信息前获取了锁，确保储存信息的正确之后又释放了锁，防止出现死锁。

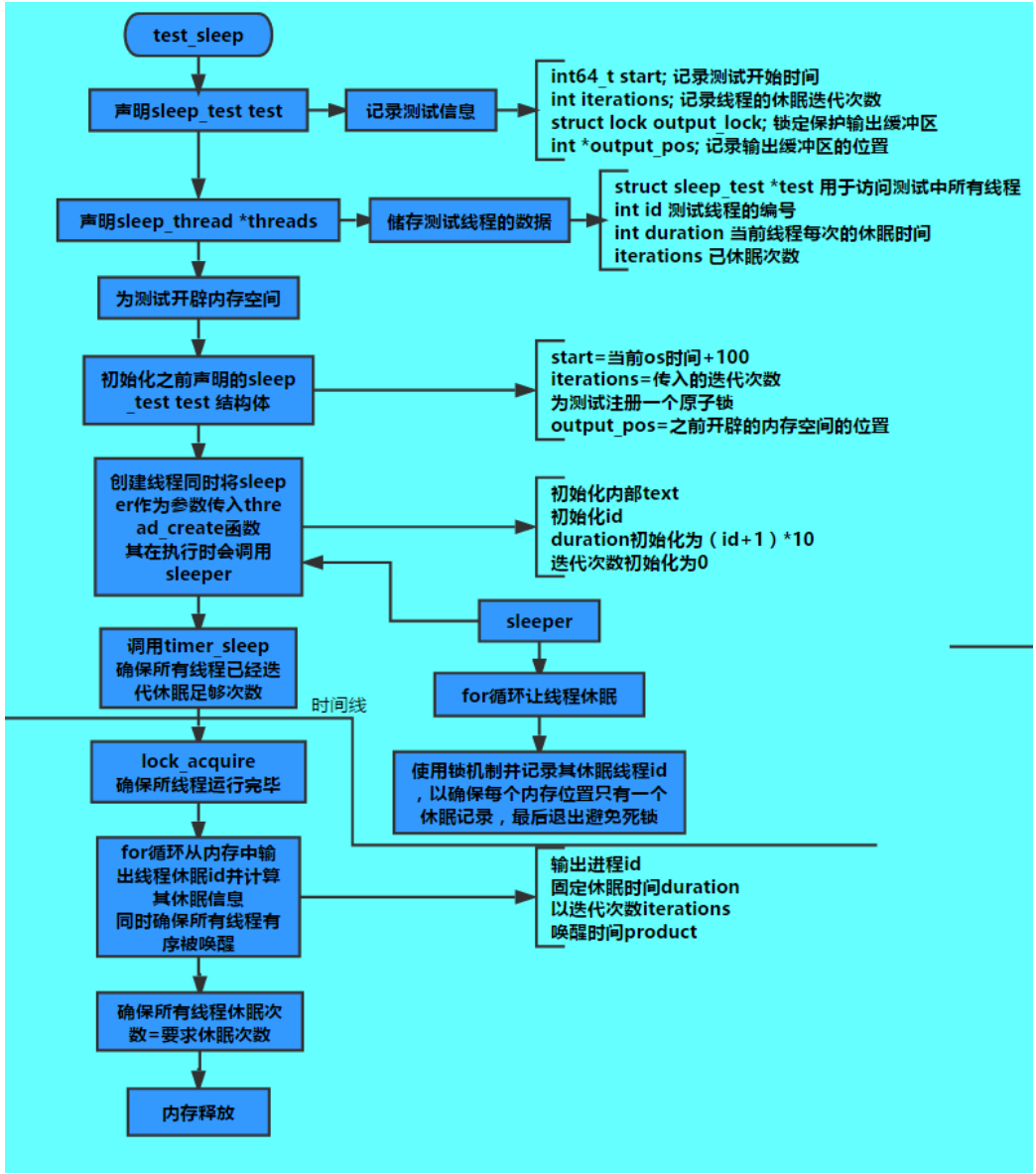
```
new_prod = ++t->iterations * t->duration;

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);
```

之后是输出函数，直接从之前储存的 id 信息获得 id 之后获取线程，之后通过计算迭代次数*休眠时间的方式输出唤醒时间。之后就是检查时间的不是递减确保是顺序唤醒。然后通过另个一个循环确保迭代次数足够。

Test_sleep 具体流程如下：



3. 结果分析

a) alarm-single：每个线程按规则休眠 10(i+1)个 ticks，休眠的迭代次数为 1。因此，线程依次被唤醒，唤醒时间 product 均它们的 duration 相等

```
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.

root@15352008蔡荣裕:~/pintos/src/threads/build# ==================
```

b) alarm-multiple：每个线程按 id 休眠 10(i+1)个 ticks，休眠的迭代次数为 7。所以我们可以看到 5 个进程按照不同的休眠时间休眠然后被唤醒然后再次休眠，直至休眠次数=迭代次数,其唤醒时间 product=当前迭代次数 iterations*休眠时间 duration。最后分析一下为什么这两个样例一开始就会通过，因为其只用到了线程的休眠与唤醒，所以只要线程的休眠唤醒机制没有问题这个样例就可以通过，不论这个休

眠唤醒机制是否会造成忙等待。

```
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(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.

root@15352008蔡荣裕:~/pintos/src/threads/build# ==================
```

2. Test：alarm-simultaneous

1. 测试目的

测试所有线程能否多次同时休眠相同时间，并且同时唤醒。

2. 过程分析

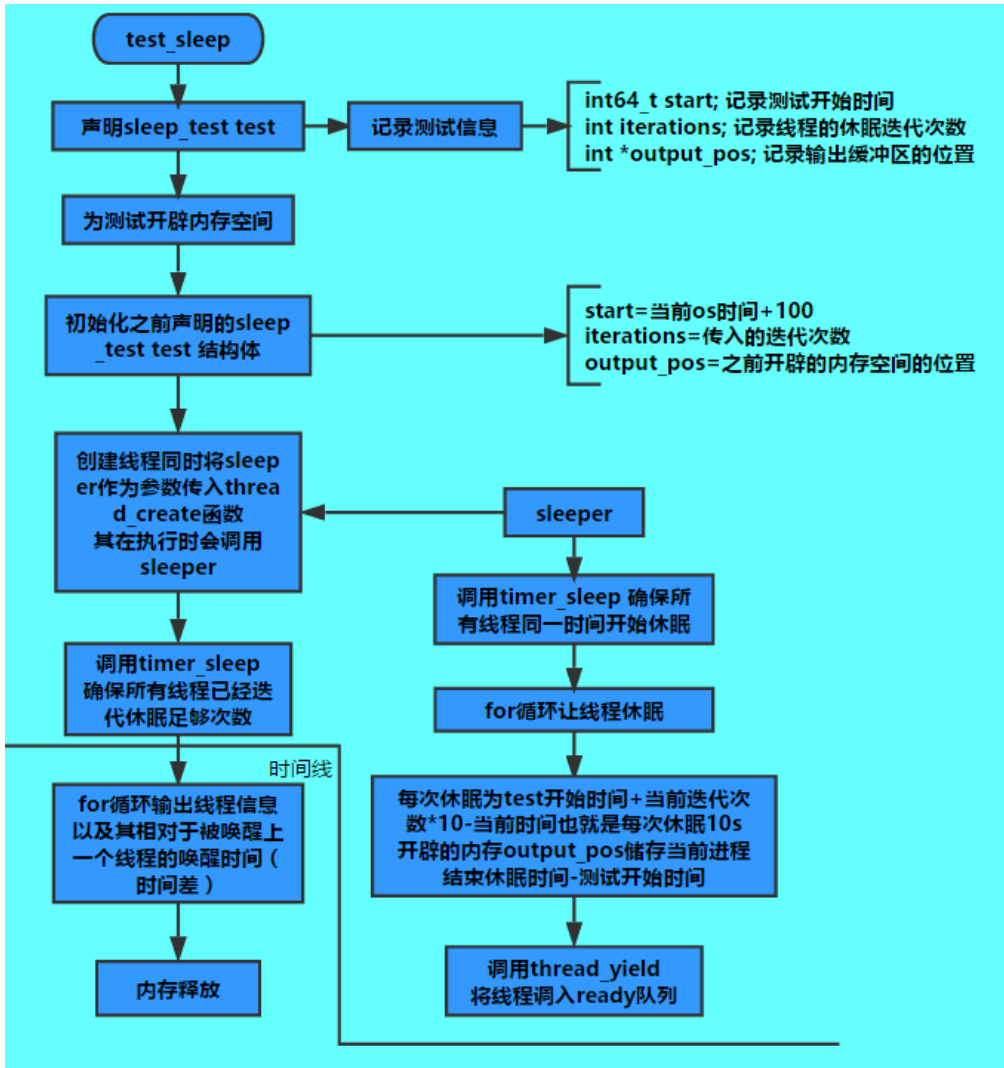
```
void
test_alarm_simultaneous (void)
{
  test_sleep (3, 5);
}
```

测试时调用了 test_sleep 传入了两个参数 thread_cnt, iterations，分别是线程数量以及休眠次数。

```
int64_t sleep_until = test->start + i * 10;
timer_sleep (sleep_until - timer_ticks ());
*test->output_pos++ = timer_ticks () - test->start;
thread_yield ();
```

先分析一下最重要的 sleeper 其在休眠时间的设置上用 sleep_until - timer_ticks 是一个随时间减少的值，这样每个线程休眠后是同时被唤醒的。也就是说即使开始休眠时间不同但是唤醒时间也是相同的。

以下 test_sleep 流程



3. 结果分析

每次休眠为 test 开始时间+当前迭代次数*10-当前时间也就是每次休眠 10s。输出的结果是当先线程的唤醒时间相对于上一被唤醒线程的唤醒时间（也就是前后唤醒两个进程的时间差）。所有线程同时休眠同时唤醒，也就是说同一迭代次数的线程其唤醒时间是相同的，其唤醒时间都与线程 0 相同时间差为 0，所以线程 1,2 会出现 woke up 0 ticks later。因为线程每次休眠 10s 所以线程 0 与前一线程的唤醒时间相差 10s。因为迭代次数为 5 所以会输出 5 次迭代信息。此处未做任何修

改就能通过测试的原因也是只要线程的休眠与唤醒直至正常就能通过测试。

```
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.

root@15352008蔡荣裕:~/pintos/src/threads/build# ========================================
```

3. Test：alarm-priority

1. 测试目的

测试线程能否按照优先级从大到小的顺序唤醒

2. 过程分析

```
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);
  }
```

创建线程时定义线程优先级，为 PRI_DEFAULT - (i + 5) % 10 – 1，也就是非单调递增的优先级。

```
thread_set_priority (PRI_MIN);

for (i = 0; i < 10; i++)
  sema_down (&wait_sema);
```

创建完成后 10 个线程后，先降低测试线程优先级，确保最后才唤醒，之后 for 循环使用 10 次 P 操作。作用如下图在第一次使用 P 操作后因为 value 初始为 0，所有在这个 while 循环中会把当前进程也就是测试进程阻塞掉。

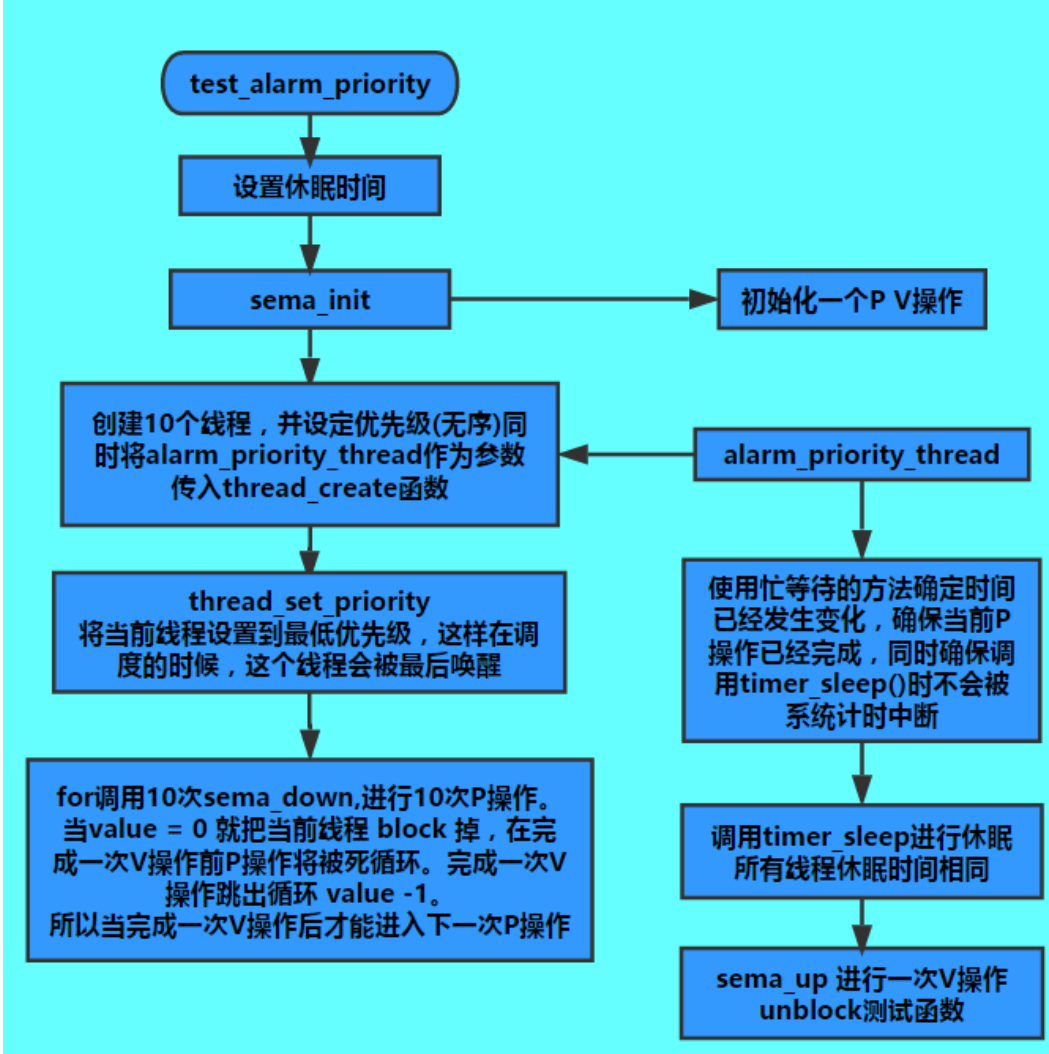
```
  while (sema->value == 0)
    {
      list_push_back (&sema->waiters, &thread_current ()->elem);
      thread_block ();
    }
```

当 10 个线程同时休眠结束时，进行 V 操作，因为 PV 操作原因每次只有一个线程被解除阻塞并执行，也就是解除了测试进程的阻塞。但是之后测试进程又会被下次 P 操作阻塞。

```
if (!list_empty (&sema->waiters))
  thread_unblock (list_entry (list_pop_front (&sema->waiters),
                              struct thread, elem));
```

直到 10 个线程都结束，测试函数才会结束。这里做的法目的就是确保测试函数最后结束，测试函数必须最后退出。当然这里也可以用休眠的方法来使测试函数最后退出，但是休眠时间就要经过计算了（结果已测试确认可行删 PV 操作增加休眠函数），确保其退出时所有线程已经休眠结束。

因为线程同时唤醒所以会同时进入 ready 队列，在没改动之前进入 ready 是按创建顺序进入的。改动后进入 ready 为排序进入。



3. 结果分析

当 10 个线程会以从大到小的优先级有序的插入 ready 队列运行。所以出是有序的按照优先级从大到小输出。(如不改动就会按照创建顺序进入，按照创建顺序输出)

```
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.

root@15352008蔡荣裕:~/pintos/src/threads/build# =
```

4. Test：alarm-zero & alarm-negative

1. 测试目的

测试休眠时间=0。当休眠时间=0 时应该立即返回
测试休眠时间<0。要求不会崩溃。

2. 过程分析

休眠时间为 0 立即返回输出 pass。

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

休眠时间为负数，立即结束休眠输出 pass。

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

3. 测试结果

这是两个特殊的测试，测试休眠函数参数非法输入时线程休眠是否正常，通过这个测试只需要休眠之前加上对参数的判断即可。

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

root@15352008蔡荣裕:~/pintos/src/threads/build#
Boot complete.
Executing 'alarm-negative':
(alarm-negative) begin
(alarm-negative) PASS
(alarm-negative) end
Execution of 'alarm-negative' complete.

root@15352008蔡荣裕:~/pintos/src/threads/build#
```

(三) 实验思路与代码分析

1. 要求通过修改 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。之前分析了 timer_sleep，其原本的休眠机制在休眠的时候不断把 running 中进程扔进就绪队列，这样占用了很多 CPU 资源。

```
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;
  thread_block ();
  intr_set_level (old_level);
}
```

解决方法重写 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 thread_unblock 之前该线程都不会被调度执行。此处还要注意一点之前分析的 alarm-zero / alarm-negative 这两个测试说明了对于非法的休眠 ticks 必须直接返回所以加了一句 if 判断。

因为要给线程赋值阻塞时间所以需要在线程结构体中增加一个参数 `int64_t ticks_blocked;  /* Blocked time. */` 并在创建线程时初始化为零,在 init_thread 中初始化即可。

之后我们有了线程的休眠时间，那么就要检测线程什么时候被唤醒了，所以在系统 ticks++的时候需要减少所有被阻塞线程的 tick_blocked。于是就需要增加一个函数 blocked_thread_check，判断是否被阻塞，是否可以被唤醒，同时在系统时间中断时调用这个函数。

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

Thread_foreach 在每次中断的时候遍历所有线程，传入 blocked_thread_check 对所有被 block 的线程减少休眠时间，对需要唤醒的线程唤醒。

```
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);
    }
  }
}
```

2. 要求修改 pintos 排队的方式来使得所有线程按优先级正确地被唤醒。首先看到 thread_unblock 函数其主要作用就是把当前线程直接扔到 ready 队列尾部。

```
list_push_back (&ready_list, &t->elem);
```

解决方法：要按照优先级正确地被唤醒那么就不应该直接直接放到 ready 队列尾部应该按照优先级排序。

```
/* Operations on lists with ordered elements. */
void list_sort (struct list *,
                list_less_func *, void *aux);
void list_insert_ordered (struct list *, struct list_elem *,
                          list_less_func *, void *aux);
```

在 list.h 中我们可以看到如上对于 list 的操作，可以选择插入后 sort 也可以选择直接使用 list_insert_ordered，但是前提都是我们现在需要构建一个 list_less_func，

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

所以我们现在需要写一个 cmp 函数比较优先级，实现如下。

```
bool
thread_cmp_priority(const struct list_elem *elem1,
                    const struct list_elem *elem2,void *aux)
{
  return list_entry(elem1,struct thread,elem)->priority
         > list_entry(elem2,struct thread,elem)->priority;
}
```

此处使用 list_entry 这个宏定义能返回一个线程的指针。

实现 cmp 函数之后只需要将 thread_unblock 改为如下。

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

同时发现在 thread_yield 和 init_thread 都有此函数,于是全改了。

说一下这里选择 list_insert_ordered 原因，主要还是时间复杂度问题因为在插入线程后对 ready 队列 sort，会造成 n^2lgn 的复杂度，而有序插入复杂度为 n^2。这个改造完成之后 tests alarm-priority 会通过。

3. 实验结果

以 test alarm-multiple 为例在修改休眠方式前这个测试的运行时间如下，可以看出 CPU 忙等待。

```
Timer: 842 ticks
Thread: 0 idle ticks, 844 kernel ticks, 0 user ticks
```

修改休眠方式后其花费时间如下

```
Timer: 848 ticks
Thread: 550 idle ticks, 301 kernel ticks, 0 user ticks
```

```
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/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
19 of 27 tests failed.
make: *** [check] Error 1
root@15352008蔡荣裕:~/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？**

  原来的休眠算法，用循环的方式将进程从 running 队列调到 ready 队列。原理上是一种忙等待的机制是实现休眠和唤醒。通过一个循环判断是否达到需休眠的时间，如果达到休眠时间，则退出循环，否则把线程丢到 ready 队列末尾，然后交由调度算法处理。

  这样休眠唤醒实际上是没有问题的，只是 CPU 一直处于忙等待，虽然线程休眠了，但是 CPU 不出在空闲状态，一直在进行线程切换。

  通过之前分析的 alarm-single 和 alarm-mutiple 测试，只要睡眠唤醒机制正常且唤醒的顺序符合睡眠时间长的后唤醒即可通过，alarm- simultaneous 只要能做到同时唤醒即可通过，所以原来的休眠唤醒机制一样能通过测试。而 alarm-zero 和 alarm-negative 两个测试中，原来的 timer_sleep 会直接退出循环，所以也会通过。

- **2. 为什么休眠时要保证中断打开？**

  因为之前的休眠方式是一种忙等待的机制，虽然线程休眠但是 CPU 并没有处于空闲状态，如果此时中断关闭，那么可能会造成其他所有需要 CPU 的非休眠线程 “饿死”因为当前 CPU 处于一直在调度休眠进程的状态，这个休眠线程还是拥有 CPU 控制权，使得其他需要 CPU 的线程无法运行。但是在我们修改完休眠方式之后其实可以不用保证中断打开了因为这种休眠方式 CPU 还是有空闲时间的。

- **3. 中断和轮询的区别？**

  CPU 要和外部设备进行通信，可以采用轮询和中断两种方式。

  轮询是由 CPU 定时发出询问，依序询问每一个周边设备是否需要其服务，有即给予服务，服务结束后再问下一个周边，接着不断周而复始。轮询实现容易，操作简单，但效率偏低。

  中断，就是打断正在进行中的工作。中断不需要处理器轮询设备的状态，设备在自己发生状态改变时将主动发送一个信号给 CPU，后者在接收到这一通知信号时，会挂起当前正在执行的任务转而去处理响应外设的中断请求。

  总的来说，轮询和中断一个是 CPU 主动访问外部设备，一个是外部设备访问 CPU。虽然说中断更节约系统资源，能更快的感知设备状态变化，在中断的实现与控制比轮询复杂得多。实际用中还是以中断轮询结合出现的多。