中山大学本科生实验报告

（2018 学年春季学期）

课程名称：Operationg System　　任课教师：饶洋辉

年级+班级	16 级 + 2 班	专业（方向）	信息与计算科学
学号	16343065	姓名	桑娜

1. 实验目的

- 了解 27 个 tests 中前 6 个 test 的测试目的、运行机制、结果分析；
- 通过修改休眠函数保证 pintos 不会在线程休眠时忙等待；
- 通过修改线程 ready_list 的排队方式，使得线程能够按优先级获得 CPU；
- 了解系统中断的作用和执行方法。

2. 实验过程

(一) Test 源文件分析

alarm-single

测试目的：

创建 5 个线程命名为“thread i”（i=0,1,2,3,4），让 thread i 休眠直到自测试启动后(i+1)*10 个 ticks 被唤醒，记录它们被唤醒的顺序，看是否与预想的一致。

如果测试成功，那么输出的唤醒次序与 product 值的非降序应该是一致的。其中 poduct 定义如下：

product = sleep duration(单次休眠时间) * iteration count(休眠次数)

也就是这个线程被唤醒的时刻与测试启动时之间的 ticks 数，product 非降序，就是被唤醒的顺序是正确的。

过程分析：

首先在 pintos/src/test/threads/alarm_wait.c 中查看 test_alarm_single 是如何进行测试的：

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

该测试调用了 test_sleep 函数，用于测试线程的休眠与唤醒。

在同一文件中查看 test_sleep() 的定义，主要可以分为以下几个部分：

（1）结构体相关的变量

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

首先是 sleep_test 结构体变量 ***test***，声明及相关成员解释如下：

```
truct sleep_test
 {
   int64_t start;              /* 测试开始时的时间 */
   int iterations;             /* 每个线程的休眠次数 */

   /* Output. */
   struct lock output_lock;    /* 保护输出buffer的lock */
   int *output_pos;            /* 当前处于的输出buffer中的位置 */
 };
```

这里用到了结构体 lock，它定义在 src/threads/synch.h 中，它是为了控制线程能否获取到某一资源(在这里，这个资源是输出 buffer，也就是 output 所指向的用来记录线程唤醒顺序的区域)。

```
struct lock
  {
    struct thread *holder;
    struct semaphore semaphore;
  };
```

holder 是某一时刻拥有这个 lock 的线程，在 lock 被 holder 拥有的时间内，其他线程无法访问指定的资源。Semaphore 定义在同一个文件中：使用 lock 时，value 的值为 1，waiters 是等待拥有 lock 的 list。

```
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
```

指向 sleep_thread 结构体类型的指针变量 ***threads***，声明及相关成员解释如下：

```
struct sleep_thread
  {
    struct sleep_test *test;     /* 所有线程共享的关于某一测试(sleep_test)的信息 */
    int id;                     /* 休眠线程 ID. */
    int duration;               /* 单次休眠时间 */
    int iterations;             /* 到目前已经休眠的次数 */
  };
```

（2）初始化

- 开辟空间：

```
threads = malloc (sizeof *threads * thread_cnt);
output = malloc (sizeof *output * iterations * thread_cnt * 2);
```

threads 所指向的空间大小为 thread_cnt 个原本的 *thread 的大小，也就是现在有 thread_cnt 个指向 sleep_tread 的指针了。

output 所指向的空间大小为 2*(iterations*thread_cnt) 个 *output 的大小，这个空间是用来按被唤醒的顺序存放相应的线程 id 的（具体见 sleeper 函数）。

- 测试的初始化：

```
test.start = timer_ticks () + 100;
// test的起始时间为 当前时刻+100 ticks

test.iterations = iterations;
// 休眠次数为test_sleep的第2个输入参数

lock_init (&test.output_lock);
// 初始化lock

test.output_pos = output;
// 输出buffuer的起始位置设为output的起始位置
```

lock_init 定义在 synch.c 中：先确认 lock 指针是指向一个实体的，然后把 lock 的拥有者设为 NULL。

```
void
lock_init (struct lock *lock)
{
  ASSERT (lock != NULL);

  lock->holder = NULL;
  sema_init (&lock->semaphore, 1);
}
```

```
void
sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema != NULL);

  sema->value = value;
  list_init (&sema->waiters);
}
```

用 1 初始化 lock 的 semaphore 成员的 value，初始化 smeaphore 的等待队列。

- 线程的初始化：

```
for (i = 0; i < thread_cnt; i++)
  {
    struct sleep_thread *t = threads + i; // 指向第i个线程的指针为t
    char name[16];

    t->test = &test;                      // 共享本test
    t->id = i;                            // 线程 id 为 i
    t->duration = (i + 1) * 10;           // 线程休眠时间为 (i+1)*10
    t->iterations = 0;                    // 已经休眠次数为 0

    snprintf (name, sizeof name, "thread %d", i);
    // 第i个线程命名为thread i
    thread_create (name, PRI_DEFAULT, sleeper, t);
    //thread i, 优先级为31(默认), 执行sleeper函数(休眠)
  }
```

在同一文件中查看 sleeper() 的定义，详细解释如下：

```
147  static void
148  sleeper (void *t_)
149  {
150    struct sleep_thread *t = t_;          // 获取线程指针
151    struct sleep_test *test = t->test;    // 获取线程所在的test
152    int i;                                // 迭代次数
153
154    for (i = 1; i <= test->iterations; i++)
155      {
156        int64_t sleep_until = test->start + i * t->duration;
157        /* test->start 是测试启动时的时刻, 线程 &t 不断地被休眠、唤醒,
158           这里规定了线程t第i次被唤醒的时刻是测试启动后 i*t->duration
159        */
160        timer_sleep (sleep_until - timer_ticks ());
161        // 被唤醒时的时刻 - 当前的时刻 = 应当休眠的时间
162        lock_acquire (&test->output_lock);  // 上锁
163        *test->output_pos++ = t->id; // 将被唤醒的线程的id加入输出buffer
164        lock_release (&test->output_lock);  // 解锁
165      }
166  }
```

为了便于区分某个时间点，与经历的一段时间，作以下说明：

timer_ticks() 返回的是自系统被引导，经历的 ticks 数，我把这个当作系统的“时刻”，也就是从系统被引导的那一刻开始计时，每个时间点都有它对应的“时刻”。

而“时间”指的是从某一时刻至另外一个时刻之间所经历的 ticks 数。
实际上，不同线程在被创建、执行，或者中途的操作中是会消耗 ticks 的，因此线程 t 每一次真正“休眠”的时间并不是 t->duration，而是先规定好它被唤醒的时刻 sleep_until，在执行到 timer_sleep 的时候通过 timer_ticks 获取到当前时刻，sleep_until 减去 timer_ticks 就是线程真正应该休眠的时间。

```
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
}
```

在 synch.c 中查看 lock_acquire：断言 lock 没有被正在运行的线程拥有，因为 pintos 中的 lock 不是“递归的“，一个正在拥有某一 lock 的线程不可以再去获得这个 lock 的拥有权。
再在同一文件中查看 sema_down。

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

在关闭中断功能后，因为 sema 的 value 初始值是 1，所以直接让 value 减 1，这就变为了 0。修改正在运行的线程(假设为 A)为 lock 的拥有者。这个时候，如果获得 CPU 的线程变为了 B，并且它执行了 lock_acquire，因为 value 为 0，那么线程 B 的 elem 就会到 sema 的等待队列中排队，同时线程会被阻塞，schedule 函数调度下一个线程。

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

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

同一文件中，查看 lock_release，首先要确认正在占用 CPU 的线程是 lock 的拥有者，然后把 lock 的拥有者设为 NULL，最后调用 sema_up。
再在同一文件中查看 sema_up。

```
old_level = intr_disable ();
if (!list_empty (&sema->waiters))
  thread_unblock (list_entry (list_pop_front (&sema->waiters),
                              struct thread, elem));
sema->value++;
intr_set_level (old_level);
```

在关闭中断功能后，将排在 sema 的等待队列中的第 1 个 elem 所代表的线程 unblock，也就是把它的状态设为 READY，并放入 ready_list 中。最后要把 sema 的 value 加 1。
在 test_sleep 函数中，有这样一条语句：
这条语句是让主线程等待所有的子线程完成。

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

线程 B 在从 sema 的等待队列中 unblock 之后，可以直接获得 CPU(因为此时就绪队列中只有线程 B，其他的线程要么已经执行完毕，如 A；要么被 block 了，如 C，D，...)。那么线程 B 就恢复上次的状态，sema 的 value 又减为 0，B 成为 lock 的拥有者。
整理一下逻辑：因为 pintos 切换线程的速度很快，为了保证在输出 buffer 中记录被唤醒的顺序时不

被打扰，使用了 lock。线程休眠后，它们的 elem 是按照唤醒顺序排到 ready_list 中的，而当获得了 CPU 的线程要写 buffer 的时候，lock 会让后面请求拥有 lock 的线程的 elem 依次排在另一个队伍 sema 的等待队列中，并且阻塞掉它们。直到之前的线程写好了它的 id，排在 sema 等待队列中的 elem 所代表的线程才会被依次唤醒，写它们的 id 到输出 buffer 中。

（3）打印唤醒顺序

```
product = 0;
for (op = output; op < test.output_pos; op++)
  { // op 指向 output 起始地址，遍历整个输出buffer
    struct sleep_thread *t;
    int new_prod;

    ASSERT (*op >= 0 && *op < thread_cnt);
    t = threads + *op;  // *op 的值为按唤醒顺序获取到的线程id

    new_prod = ++t->iterations * t->duration;
    // 计算新的 product

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

    if (new_prod >= product)  // 确认 product 是否按非降序排列
      product = new_prod;
    else
      fail ("thread %d woke up out of order (%d > %d)!",
            t->id, product, new_prod);
  }
```

（4）确认 thread i (i=0,1,2,3,4) 被唤醒了 iterations 次。 （代码略）

（5）释放动态分配的内存与解锁。 （代码略）

结果分析：

如下图所示，在 alarm-single 测试中，创建了 5 个线程，每个线程休眠 1 次。第 i 个线程被唤醒的时刻为 test->start 后 (i + 1) * 10 个 ticks (i=0,1,2,3,4)。这里由于 iteration count 均为 1，所以 product 的值就是 sleep duration 的值。依次为 10，20，30，40，50。

```
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.
sangna@16343065sang:~/pintos/src/threads/build$
```

alarm-multiple

测试目的：

创建 5 个线程命名为“thread i”(i=0,1,2,3,4)，对于每个 thread i 不断地休眠、唤醒 7 次，第 j 次被唤醒的时刻设为 test->start 后 j*t-> (i+1)*10 个 ticks，记录所有的线程被唤醒的顺序，看是否与预想的一致。如果测试成功，那么输出的唤醒次序与 product 值的非降序应该是一致的。其中

poduct 定义与 alarm-single 相同，product 非降序，就是被唤醒的顺序是正确的。

过程分析：

先在 pintos/src/test/threads/alarm_wait.c 中查看 test_alarm_multiple 是如何进行测试的：5 个线程，每个线程休眠 7 次。

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

该测试调用了 test_sleep 函数，用于测试线程的休眠与唤醒。这个函数已在 alarm-single 中详细分析过，这里只是每个线程休眠的次数由 1 变为了 7。

结果分析：

如下图所示，在 alarm-multiple 测试中，创建了 5 个线程，每个线程休眠 7 次。

```
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
```

与 alarm-single 一样，唤醒次序与 product 值的非降序是一致的（部分截图）：

```
(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.
sangna@16343065sang:~/pintos/src/threads/build$
```

alarm-simultaneous

测试目的：

创建 3 个线程命名为“thread i”（i=0,1,2），对于每个 thread i 不断地休眠、唤醒 5 次，所有线程第 i 次被唤醒的时刻设为 test->start 之后 i*10 个 ticks。正确的情况下，在每次迭代中，不同线程被唤醒的时刻之差为 0 ticks；而下一次迭代与上一次迭代之差为 10 ticks。

过程分析：

先在 pintos/src/test/threads/alarm_simultaneous.c 中查看 test_alarm_simultaneous 是如何进行测试的：3 个线程，每个线程休眠 5 次。

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

该测试调用了 test_sleep 函数，用于测试线程的休眠与唤醒。

在同一文件中查看 test_sleep() 的定义，发现其大部分内容与 alarm-wait.c 中的内容是相似的。主要的区别在于：

1. alarm-simultaneous 中不同线程的 duration 是相同的，均为 10 ticks；
2. 输出 buffuer（output 所指向的空间）存放的内容不是按被唤醒的次序排列的相应的线程 id，而是线程被唤醒的时刻距离测试开始时刻的时间。

创建 thread_cnt 个线程，默认优先级，执行 sleeper 函数。

```
thread_create (name, PRI_DEFAULT, sleeper, &test);
```

在同一文件查看创建线程使用的函数 sleeper 核心部分是什么样子的。

```
for (i = 1; i <= test->iterations; i++)
  {
    int64_t sleep_until = test->start + i * 10;
    // 第i次迭代被唤醒的时刻为 测试开始的时刻 + i*10
    timer_sleep (sleep_until - timer_ticks ());
    *test->output_pos++ = timer_ticks () - test->start;
    /* output 所指向的空间存放的是
    当前时刻 - 测试开始时刻 = 当前距测试开始的时间 */
    thread_yield ();
    // 把当前线程放入ready_list,并调度下一线程
  }
```

在打印结果的时候，代码是这样的：

```
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]);
```

变量 i 遍历整个输出 buffer，迭代序号为 i / thread_cnt，线程 id 为 i % thread_cnt，当前线程与上一个线程被唤醒的时刻之差为 output[i] – output[i-1]。

结果分析：

如下图所示，在 alarm-multiple 测试中，创建了 3 个线程，每个线程休眠 5 次。

可以看到在每次迭代中，不同线程被唤醒的时刻之差为 0 ticks；而下一次迭代与上一次迭代之差为 0 ticks，这是正确的。

```
(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.
sangna@16343065sang:~/pintos/src/threads/build$
```

alarm-priority

测试目的：

创建 10 个线程命名为“prirority p(i) “，(i=0,1,...,9)。其中，p(i)为线程 prirority p(i) 的优先级，且 p(i) = 30 – (i + 5) % 10。

过程分析：

先在 pintos/src/test/threads/alarm_priority.c 中查看 test_alarm_priority 是如何进行测试的。

- 主线程 test_alarm_priority。

```
wake_time = timer_ticks () + 5 * TIMER_FREQ;
// 设置线程被唤醒时刻为当前时间 + 5 * 100
sema_init (&wait_sema, 0);
// 初始化semaphore结构体wait_sema
```

首先设置了线程应该被唤醒的时间 wake_time。

```
for (i = 0; i < 10; i++)
  {
    int priority = PRI_DEFAULT - (i + 5) % 10 - 1;
    /*     i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    priority = 25,24,23,22,21,30,29,28,27,26 */
    char name[16];
    snprintf (name, sizeof name, "priority %d", priority);
    thread_create (name, priority, alarm_priority_thread, NULL);
  }
```

接下来函数创建了 10 个线程，命名为 priority j(j=21,22,23,...,30)，它们的优先级和名字一致，执行 alarm_priority_thread。

- 子线程 alarm_priority_thread

```
  /* 忙等待直到当前的时刻改变 */
  int64_t start_time = timer_ticks ();
  while (timer_elapsed (start_time) == 0)
    continue;
```

其实这里包括下面的注释我并没有完全理解，为什么从 1 个 tick 的最初开始可以避免冲突。

```
/* 现在在一个timer tick的最开端，可以调用timer_sleep
而不用担心检查时间和系统中断之间的竞争 */
timer_sleep (wake_time - timer_ticks ());
msg ("Thread %s woke up.", thread_name ());
sema_up (&wait_sema);
```

这里 timer_sleep 在修改后，将线程 block，在 wake_time 被 unblock，按照优先级有序地排到 ready_list 中。

再回到 test_alarm_prirority 这个主线程。

- 主线程 test_alarm_priority

```
thread_set_priority (PRI_MIN);

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

在执行创建 10 个线程的 for 循环之后，将主线程的优先级设为最低，这样所有的子线程在被唤醒后在 ready_list 中都将排在它之前。

再回顾一下 sema_up 和 sema_down。

  - sema_up：若 sema 的等待队列不是空的，就 unblock 位于其等待队列第一位的 elem 所代表的的线程；无论是否为空，都将 value 加 1。
  - sema_down：若 value 为 0，则调用 sema_down 的线程将被 block，它的 elem 会被放到 sema 等待队列的末尾；当 vlue 不为 0 时，value 就减 1。

根据我的理解，这里就是通过这两个函数，让主线程等待所有子线程完成使命。

结果分析：

- 失败样例

```
Executing 'alarm-priority':
(alarm-priority) begin
(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) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
sangna@16343065sang:~/pintos/src/threads/build$
```

因为此时还没有修改 thread_yield 和 thread_unblock 中的 elem 插入方式，所以 ready_list 中 elem 的排列顺序与它们所代表的线程的优先级高低不是一致的。

- 成功样例

```
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.
sangna@16343065sang:~/pintos/src/threads/build$
```

经过修改后的成功测试样例，具体已在实验报告 Task 2 中写出。

alarm-zero

测试目的：

测试 timer_sleep(0)，结果应当立即返回。

过程分析：

先在 pintos/src/test/threads/alarm_zero.c 中查看 test_alarm_zero 是如何进行测试的：

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

只有两条语句，第一条表示让线程休眠 0 tick，timer_sleep 函数将在下一部分中详细分析。第二条 pass ( ) 定义在 pintos/src/test/threads/test.c 中，具体如下：

```
 97  void
 98  pass (void)
 99  {
100    printf ("(%s) PASS\n", test_name);
101  }
```

可以看到就是打印表示测试通过了的信息。

结果分析：

如下图所示，成功打印出 “(alarm-zero) PASS”的信息，表示测试通过。

```
Executing 'alarm-zero':
(alarm-zero) begin
(alarm-zero) PASS
(alarm-zero) end
Execution of 'alarm-zero' complete.
sangna@16343065sang:~/pintos/src/threads/build$
```

alarm-negative

测试目的：

测试 timer_sleep(-100)，唯一的要求就是它不会崩溃。

过程分析：

如下图所示，alarm-negative 与 alarm-zero 基本一样，唯一的区别在于休眠的时间为 -100 ticks 而不是 0。

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

结果分析：

成功打印出 “(alarm-negative) PASS”的信息，表示测试通过。

```
Executing 'alarm-negative':
(alarm-negative) begin
(alarm-negative) PASS
(alarm-negative) end
Execution of 'alarm-negative' complete.
sangna@16343065sang:~/pintos/src/threads/build$
```

（二）实验思路与代码分析

Task 1：通过修改 pintos 的线程休眠函数来保证 pintos 不会在一个线程休眠时忙等待。

- **问题分析**

忙等待的原因在于，在距离 start 的时间未达到 ticks 时，timer_sleep 函数只是不断地调用 thread_yield 函数。

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

thread_yield 核心部分如下：

```
if (cur != idle_thread)
  list_push_back (&ready_list, &cur->elem);
cur->status = THREAD_READY;
schedule ();
```

这个代码的含义是：将当前线程 cur 的状态设为 ready；如果 cur 不是 idle_thread，就把它放到 ready_list 的末尾。然后调用 schedule 函数。

schedule 函数主要分为以下几个部分：

(1) running_thread ( ) 返回当前线程起始指针位置，赋给 cur（这里获取到的应该就是上一步被放入 ready_list 尾部的线程指针）；

next_thread_to_run ( ) 返回 ready_list 中的第一个线程，若 ready_list 为空，则返回 idle_thread，赋给 next。（假若 ready_list 只有 1 个线程，那么 next 指向的也就是上一步被放入 ready_list 尾部的线程指针，此时 next==cur）。

```
struct thread *cur = running_thread ();
struct thread *next = next_thread_to_run ();
```

(2) 断言 cur 的状态不是 running（在 timer_sleep 这个情境下，应该是 ready），断言 next 是线程。

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

(3) 在 cur 和 next 指向的不是同一个线程的情况下，调用 switch_threads 函数。

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

switch_thread 的作用是保存当前线程 cur 的状态，赋给 prev；恢复新线程 next 之前保存的状态，赋给 cur。

(4) 最后一步调用了 thread_schedule_tail 函数完成收尾工作。

```
thread_schedule_tail (prev);
```

这部分除了“如果 prev 的状态是 dying，就清空资源”，其他地方都没有看(懂)。根据网上查到的信息，thread_schedule_tail 大概就是“获取当前线程， 分配恢复之前执行的状态”。

综上，原 timer_sleep 函数的实质是：在 ticks 时间内不断地调用 thread_yield。把当前线程状态设为 ready，并把它的 elem 放入 ready_list 的末尾，通过 schedule 函数切换到位于 ready_list 头部的 elem 所代表的线程。

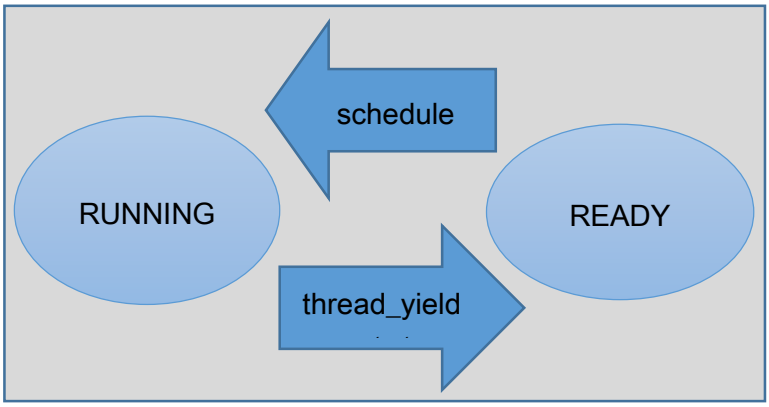


图 1 – 线程原“休眠”方式

如图 1 所示，在 ticks 时间内，所有线程都在 running 和 thread 状态之间不断切换，大大浪费 CPU 资源。

- **解决办法**

把需要休眠的线程切换为 block 状态，在 ticks 时间内不再加入 ready_list，也就没有可能获取 CPU，其他线程可以获取 CPU 正常运行；

在每一次系统中断的时候，查看 block 线程是否休眠完毕，若完毕则唤醒；

将要被唤醒的线程状态切换为 ready，并加入 ready_list，使其有可能重新获得 CPU。
步骤如下。

1. src/threads/thread.h，给 thread structure 增加数据成员如下

```
int64_t ticks_to_sleep; // *记录线程剩余的休眠时间
```

2. src/threads/thread.c，thread_create 函数内加入语句将 ticks_to_sleep 初始化为 0。

```
t->ticks_to_sleep = 0;
```

3. src/devices/timer.c，修改 timer_sleep 函数如下

```
void
timer_sleep (int64_t ticks) // 想让线程休眠ticks时间
{
  if(ticks <= 0) return;
  ASSERT (intr_get_level () == INTR_ON);
  enum intr_level old_level = intr_disable ();
  thread_current()->ticks_to_sleep = ticks;
  thread_block();
  intr_set_level (old_level);
}
```

将要休眠的时间 ticks 赋给 ticks_to_sleep，并调用 thread_block 函数。
thread_block 函数内容如下：

```
ASSERT (!intr_context ());
ASSERT (intr_get_level () == INTR_OFF);

thread_current ()->status = THREAD_BLOCKED;
schedule ();
```

断言不是外部中断，断言系统中断是关闭的，将 running 的线程状态切换为 blocked，调用 schedule 函数。

4. src/devices/timer.c 在 timer interrupt 函数中增加语句

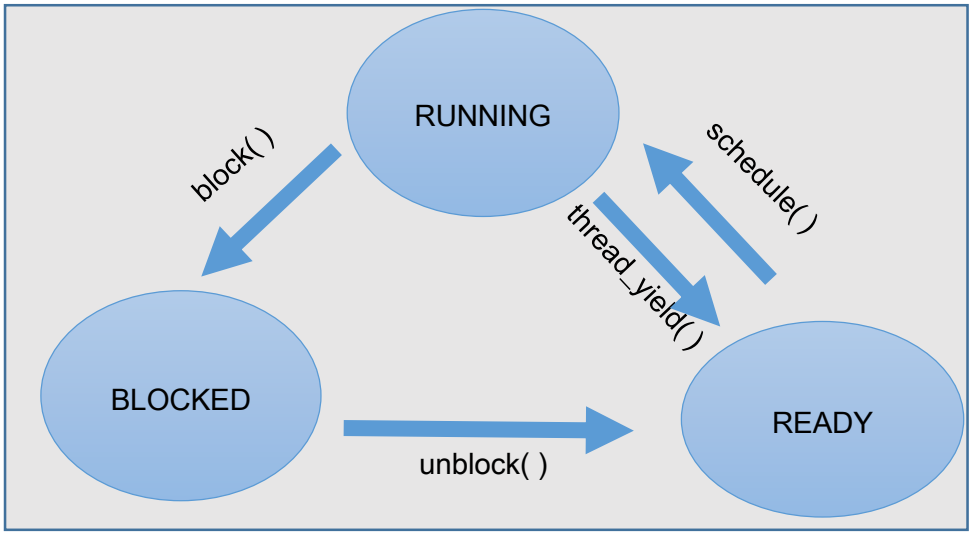
```
thread_foreach (blocked_thread_check, NULL); //add
```

thread_foreach 是用来遍历所有线程的，这条语句的意思是对所有线程执行 blocked_thread_check。
这个 blocked_thread_check 函数是自己定义在 thread.c（同时要在所有）中的，具体如下：

```
// 检查线程是否休眠完毕
void
blocked_thread_check(struct thread *t, void *aux UNUSED)
{
  if(t->status == THREAD_BLOCKED && t->ticks_to_sleep >0)
  {
     t->ticks_to_sleep--;
     if(t->ticks_to_sleep == 0)
     {
       thread_unblock(t);
     }
  }
}
```

如果线程的状态为 block，并且需要休眠的时间大于 0，就在每一次系统中断（每个 tick 发生一次）的时候，让 ticks_to_sleep 减 1。如果这个时间变为了 0，意味着线程不需要休眠了，那么

就调用 thread_unblock 函数将其唤醒。thread_unblock 将在 Task2 中详细解释。



如图 2 所示，当某一线程休眠时，将其状态设为 block，而其他在 ready_list 中的线程可以通过 schedule 获得 CPU，使 CPU 不必忙等待。

图 2 – 线程新休眠方式

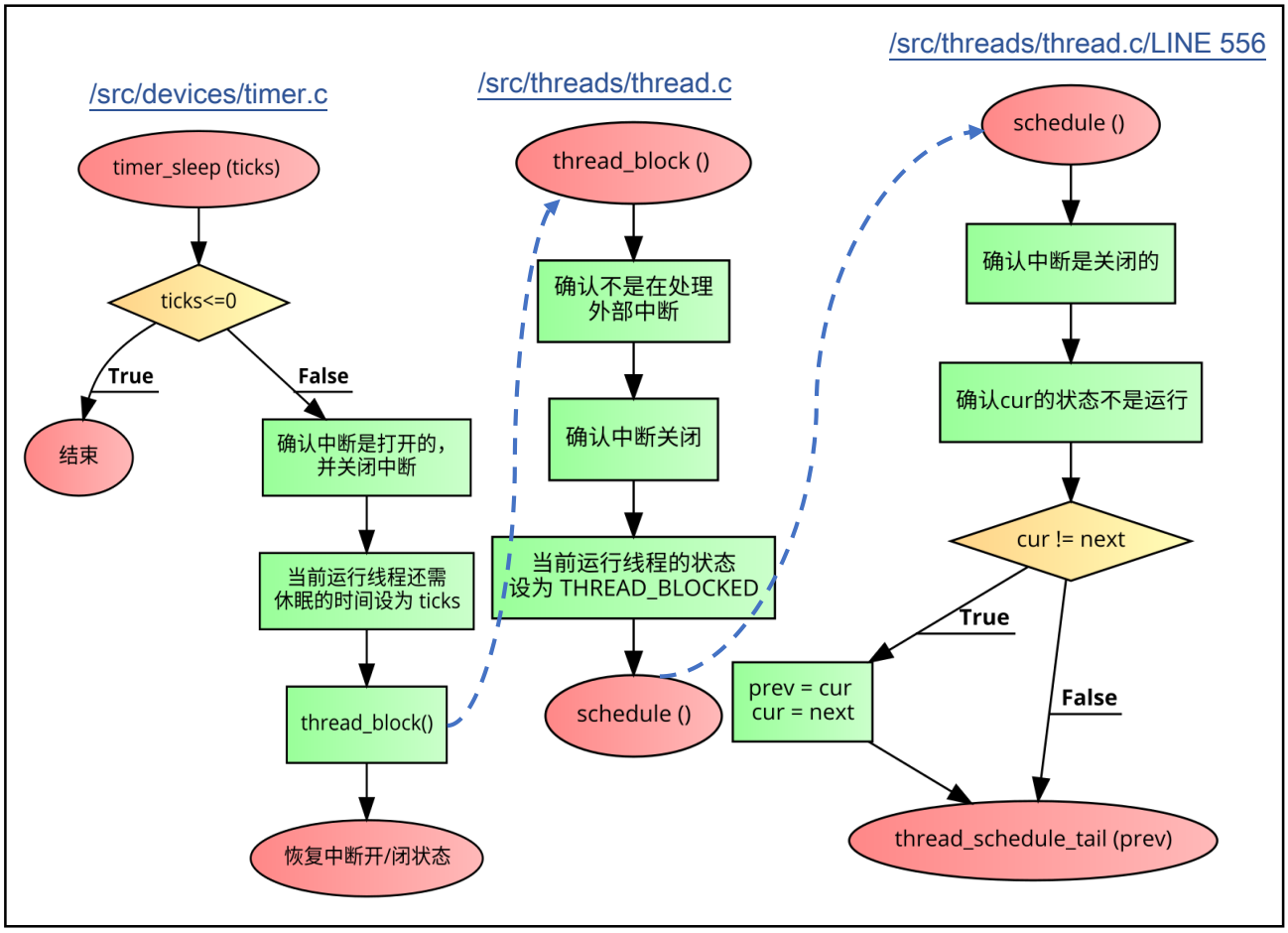


图 3 – 线程休眠函数流程图

Task 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。

- **问题分析**

根据前面的分析线程进入 ready_list 有 2 个途径：

(1) 通过 thread_yield 由 running 变为 ready；

```
if (cur != idle_thread)
  list_push_back (&ready_list, &cur->elem);
cur->status = THREAD_READY;
schedule ();
```

(2) 通过 unblock 由 block 变为 ready。

```
ASSERT (t->status == THREAD_BLOCKED);
list_push_back (&ready_list, &t->elem);
t->status = THREAD_READY;
```

这两个方法都是直接将需要被操作的线程的 elem 成员放入 ready_list 的尾部。

而在 schedule 函数中，获取下一个要运行的线程函数 next_thread_to_run 中，直接选择 ready_list 头部的 elem：

```
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

也就是说对于 ready_list 无论是入队还是出队都没有考虑线程的优先级。这显然是不对的。

需要修改源代码使得在 ready_list 中优先级最高的线程可以获得 CPU。

- **解决办法**

无须修改 schedule 函数，只要保证线程在加入 ready_list 时是有序的即可。具体步骤如下：

1. 使用 src/lib/kernel/list.h 中声明好的 list 有序插入函数：

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

在 list.c 中查看它的核心部分：

这里的 less 的宏定义 list_less_func 在 list.h 中：

```
for (e = list_begin (list); e != list_end (list); e = list_next (e))
  if (less (elem, e, aux))
    break;
return list_insert (e, elem);
```

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

如果 A<=B，函数返回 true；否则，函数返回 false。

list_insert 函数定义在 list.c 中

```
list_insert (struct list_elem *before, struct list_elem *elem)
{
  ASSERT (is_interior (before) || is_tail (before));
  ASSERT (elem != NULL);

  elem->prev = before->prev;
  elem->next = before;
  before->prev->next = elem;
  before->prev = elem;
}
```

显然是将第 2 个参数插入到第 1 个参数之前。

综上，list_insert_order 函数，在拿到要插入的元素 elem 时，从头部遍历 list，直到当 list 中的某一元素 e，elem 的某一属性和它的同一属性通过“less”函数进行比较，返回 true，便将 elem 插入到 e 之前。应用到优先级排列的情景，就是直到 elem 所代表的线程的优先级比 e 所代表的线程的优先级高的时候，便将 elem 插入到 e 之前。
那么在实现比较函数“less”的时候，返回 true 的情况，是 A. priority > B. priority。

2. 修改 thread_yield 函数和 unblock 函数
Running → Ready

```
if (cur != idle_thread)
{
  /* list_push_back (&ready_list, &cur->elem);*/
  list_insert_ordered (&ready_list, &cur->elem, greater, NULL); // add
}
cur->status = THREAD_READY;
schedule ();
```

Block → Ready

```
ASSERT (t->status == THREAD_BLOCKED);
/*list_push_back (&ready_list, &t->elem);*/
list_insert_ordered (&ready_list, &t->elem, greater, NULL); // add
t->status = THREAD_READY;
```

3. 在 thread.c 中定义比较函数 greater

```
234  static bool
235  greater(const struct list_elem *a,const struct list_elem *b, void *aux)
236  {
237    struct thread *ta = list_entry(a,struct thread,elem);
238    struct thread *tb = list_entry(b,struct thread,elem);
239    return ta->priority > tb->priority;
240  }
```

这里需要说明的是，线程本身并没有加入 ready_list，参与“排队”的是线程的一个数据成员“elem”，而通过函数 list_entry 可以获知某一 elem 代表的是哪个线程。
在 list.h 中找到 list_entry 的定义：

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

翻译一下注释： 将一个指向 list_elem 类型变量 LIST_ELEM 的指针转变为指向包含这个 LIST_ELEM 的结构体的指针。需要提供的参数：结构体的名字 STRUCT，LIST_ELEM 在结构体中的名字 MEMBER（struct thread 中有两个 struct list_elem 类型的成员：allelem 和 elem）。根据 thread_yield 和 unblock 原本的定义知道，负责“排队”的是成员 elem。

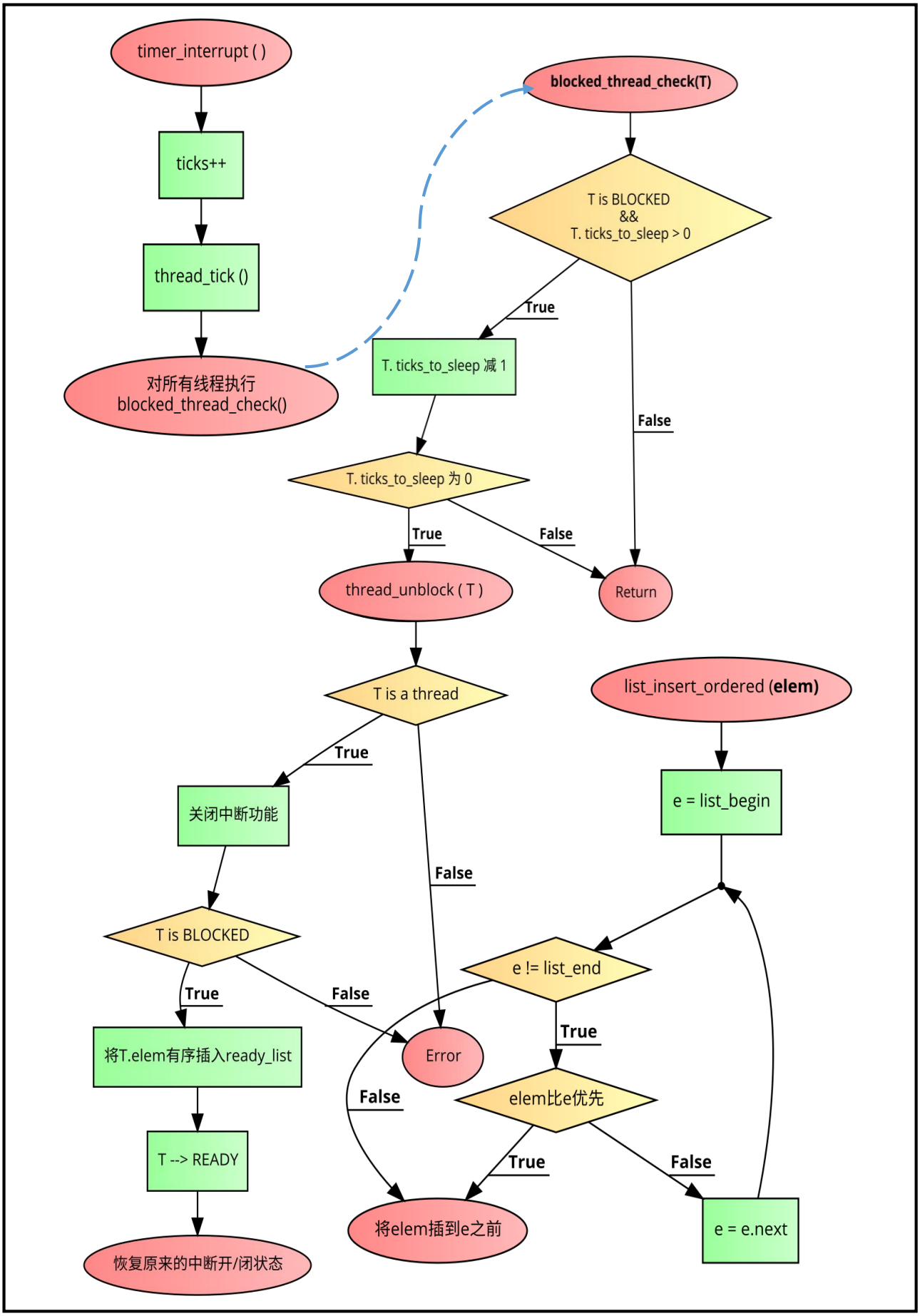
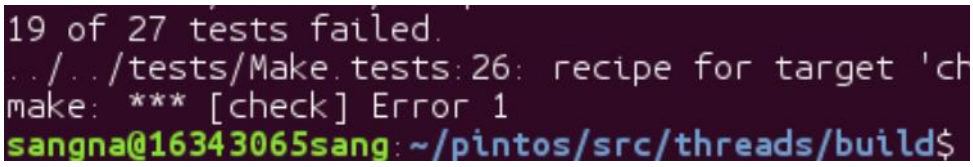


图 4 – 线程按优先级唤醒函数流程图

3. 实验结果



结果分析：之前是 20/27 test failed，也就是通过了 7 个测试，在本次实验修改完按优先级唤醒线程后，又多通过了 alarm-priority 这个 test，所以是 19/27 tests failed。至于如何修改代码，已在实验报告第二部分描述；alarm-priority 又是如何进行测试的，已经在实验报告第一部分进行分析。

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：(1) 20/27 是因为有些测试是需要我们实现或修改相关函数才可以通过的；

(2) 通过的那几个 test 中，都只用到了 timer_sleep 这个函数，而这个函数在未修改的时候是可以实现线程“休眠”的功能的，只不过它采用的忙等待方式造成了 CPU 的浪费。因此在什么都没有修改的时候仍然可以通过 7 个 test。

- 2. intr_disable()返回值是什么?为什么还要 intr_set_level()函数?

答：(1) intr_disable() 返回的是当前这一刻，系统中断这个功能是开启的还是关闭的；

(2) intr_set_level() 是为了是系统在改变中断功能开/闭状态，并且进行一定操作后，可以恢复之前的中断功能开/闭状态。

- 3. 什么情况下 schedule 调度的还是当前线程?

答：当前线程切换状态为 READY，并且 ready_list 中没有比当前线程优先级更高的其他线程的时候。一方面，当前线程切换为 READY，那它的 elem 就会被有序地插入 ready_list 中；另一方面，因为它的优先级最高，它的 elem 就会排到队头。在 scheduel 的时候，它本身就是 next_thread_to_run 的返回的线程。

- 4. 为什么线程休眠要保证中断打开？

答：如果线程休眠（调用 timer_sleep 时）中断功能是关闭的，在执行 block 函数后，中断功能仍然是关闭的，那么 timer_interrupt 就不会被调用，写在其中的 thread_foreach (blocked_thread_check, NULL)语句也就不会被执行。这样造成的后果就是没有函数来检查处于 block 状态的线程剩余的休眠时间，不会在每个 tick 给 ticks_to_sleep--。要么会造成休眠的线程不再被唤醒，要么休眠的时长不对。

5. 实验感想

我的本次实验感想如果可以用一句话来概括那就是“写代码容易，写报告难”。做完实验回过头来想想可能只写了几行代码，但是一点也不轻松。因为必须要对 test 文件进行阅读、分析，要把握整个框架，要知道每个函数在做什么，要把它们关联起来。而在阅读源码的过程中会发现很多函数都是层层嵌套的，即使用了 ctrl+shift+F 在整个 src 文件中进行查找，依然很费力。读到最后一个嵌套的函数的时候，往往已经忘了最外层的那个函数是干嘛的，又必须层层回溯。这个时候流程图就可以很好地表示出这些函数之间的调用关系和整体的框架。

通过这次实验，我总结了以下经验：

1. 写实验报告比预想的耗时多了，要预留一定的时间。同时要学会精简语言，这次实验报告差点就超过 20 页了，可是真正的“干货”我觉得没有多少。还有很多地方我没有理解。

2. 给代码截图的时候最好把相应的行号也截上，方便以后查找，尤其是自己添加的部分，很可能在以后的实验中还需要修改。分析代码的时候，最好不要以注释的形式给出。我最后生成 pdf 看的时候觉得有点头晕，希望 TA 能原谅我。

3. 对层层嵌套的函数的追根溯源要适可而止，否则很可能子子孙孙无穷尽矣(当然还是可以看完的，只不过实在没那么多精力)。有些代码不用全部通读，可以只看注释，知道主要实现的功能是什么就可以了。

4. 流程图太详细了，这一点可以类比第 1 条，有一些断言以及中断状态的开闭在流程图中可以适当省略，否则单单一个流程图就要占据一个页面，造成流程图看起来很冗余，失去了它本来是为了简化说明的意义。

综上，本次实验报告存在一些不足，但是鉴于时间的关系，我就没有再做修改了。