

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	2016 信计 2 班	专业 (方向)	信息与计算科学
学号	16339038	姓名	舒展

1. 实验目的

- 了解 pintos 的休眠唤醒机制, 并进行优化修改, 避免忙等待现象。
- 实现按照优先级进行调度的线程调度方式
- 阅读 test 部分源码, 掌握其测试内容,

2. 实验过程

(一) Test 源文件分析

- alarm-zero.c \ alarm-negative.c

测试目的:

测试传入 timer_sleep 的参数为非正数时, 系统是否还能正常运行。即希望传入非正数时, 这个两个测试能够通过。

过程分析:

timer_alarm_zero 的代码如下:

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

【分析】这个测试函数主要只做了一件事: 向 timer_sleep 函数传入一值为 0 的参数, 如果系统能够正常运行, 则通过测试。

进入 timer_sleep() 函数, 其代码如下:

```
int64_t start = timer_ticks ();

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

【分析】这是没有修改之前的 timer_sleep 代码, 可以看到, timer_elapsed(start) 是大于 0 的, 所以当 ticks 为非正数

时，循环体内的函数是不执行的。这也是为什么最开始 zero\negative 两个测试会 pass 的原因。

下面是修改过后的部分代码：

```
struct thread *current_thread = thread_current ();  
current_thread->ticks_blocked = ticks;  
thread_block ();
```

【分析】如图中所圈部分，这段代码对于 ticks 是非正数时的情况是没有处理的，直接把 ticks 赋值给 ticks_blocked，然后把线程 block 掉。这样做的后果是：该线程将永远不会被唤醒，因为在 blocked_thread_check 函数中，线程被唤醒的条件为下图：

```
t->ticks_blocked--;  
if (t->ticks_blocked == 0)  
{  
    thread_unblock(t);  
}
```

因为之前会先执行一条 ticks_blocked 语句，所以 if 中的判断将永远不会出现。

make check 一下，报错如下：

```
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ make check  
pintos -v -k -T 60 --bochs -- -q run alarm-negative < /dev/null 2> tests/threads/alarm-negative.errors > tests/threads/alarm-negative.output
```

【分析】通过测试的方法：在 timer_sleep 的开头添加如下判断即可：

```
if (ticks <= 0)  
{  
    return;  
}
```

结果分析：

```
Executing 'alarm-zero':  
(alarm-zero) begin  
(alarm-zero) PASS  
(alarm-zero) end  
Execution of 'alarm-zero' complete.  
[1]+ 已停止                  pintos run alarm-zero  
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$
```

【分析】如上图，alarm-zero 测试通过。

alarm-negative 分析过程与 alarm-zero 一样，在此省略。

- alarm-wait.c

测试目的：

创建 N 个线程，每个线程都休眠固定且不相同的时间 M 次。记录线程被唤醒的顺序，检查其是否是按照预期的顺序唤醒。如果完全符合则通过测试，如果不一样则测试失败。

过程分析：

进入该文件，首先声明了一个 test_sleep 函数，进入该函数：

首先该函数定义了几个变量，一行行分析：

首先创建了一个 sleep_test 的对象：

```
struct sleep_test test;
```

进入 sleep_test 结构体，其定义如下：

```
/* 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. */
};
```

【分析】结构体中定义了四个成员变量。start 记录测试开始时的系统时间；iteration 存储的是每个线程休眠的次数；output_pos 为一个 int 型指针，指向输出信息在输出缓冲区中的位置。

回到 test_sleep 函数，定义了一个指向 sleep_thread 类型对象的指针：

```
struct sleep_thread *threads;
```

进入 sleep_thread 结构体，其定义如下：

```
/* 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. */
};
```

【分析】结构体首先定义了一个指向 sleep_test 类型对象的指针，目的是为了使所有的 sleep_thread 都可以共享 sleep_test 中的信息；定义了一个 id 表示休眠线程的 id；duration 表示该线程要休眠的 ticks 数；iterations 用于储存该线程目前已经休眠的次数。

回到 test_sleep 函数，接下来的几行调用 msg 打印测试的相关信息。然后调用 malloc 函数为线程和输出信息动态分配内存：

```

/* Allocate memory. */
threads = malloc (sizeof *threads * thread_cnt);
output = malloc (sizeof *output * iterations * thread_cnt * 2);

```

接下来是初始化 test 变量的信息：

```

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

```

接下来是一个循环，用于创建 thread_cnt 个 thread，并将它们依次放入 ready_list 中，下面进行仔细分析：

```

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);
}

```

【分析】每次循环，创建了一个指向 sleep_thread 类型变量的指针，指针指向的位置按照 threads 数组被分配到的内存递增；中间连续的四行代码为初始化信息，id 从 0 递增，duration 每次递增 10 个 ticks。最后一行代码调用 thread_create 函数来创建一个内核线程，并将它放进 ready_list 中，这个新建的线程被分配到 cpu 时会执行 sleeper 函数。**注意：**传进去的优先级为默认优先级，即 PRI_DEFAULT，也就是说这个 thread_cnt 个线程的优先级是一样的。

下面进入 thread_create 函数进行分析：

```

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

```

【分析】thread_create 函数主要是为线程申请内存，调用 init_thread 初始化线程的信息，为其分配 tid 等，这里不再一一分析。注意：调用 init_thread 函数，线程的是被 BLOCKED 了的，如下图：

```
t->status = THREAD_BLOCKED;
```

所以在 thread_create 函数的最后，要调用一次 thread_unblock 函数将其 unblock，并放入 read_list：

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

返回到 sleep_test 函数的第一个循环中，分析一下被调用的 sleeper 函数：

```
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);  
}
```

【分析】函数的主要内容为一个 for 循环。每次循环调用 timer_sleep 函数，使线程休眠对应的 duration 个 ticks。然后将其 id 存放在 output 数组中。

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

【分析】第一个循环将创建的线程全部放入 ready_list 后，调用此语句，使主线程能休眠够长的时间，使这些线程都能过完成休眠。

下面分析 sleep_test 的第二个循环，该循环的作用是打印出实际的唤醒序列：

```
for (op = output; op < test.output_pos; op++)
```

【分析】遍历 output 数组，数组的每个元素指向一个线程，每次打印出该线程已经休眠的时间，如下图：

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

其中 new_prod 计算的便是 t 指向的线程的已休眠时间。

```

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

```

这个语句保证了线程的唤醒迅速是满足要求的，否则便会 fail。

sleep_test 的最后一个循环用于检测每个线程被唤醒的次数是否与其休眠的次数相等。

最后分析一下两个 test 函数：

```

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

```

```

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

```

【分析】test_alarm_single 创建 5 个线程，每个线程休眠 1 次；test_alarm_multiple 创建 5 个线程，每个线程休眠 7 次。

结果分析：

```

(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.

[1]+ 已停止                  pintos run alarm-multiple
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$

```

【分析】上图为 alarm-multiple 测试的输出结果，每个线程按照规则休眠 $10(i+1)$ 个 ticks，休眠次数与休眠时间的乘积等于 product 总的休眠时间。alarm-single 的测试结果与之类似，不再截图分析。

• alarm-priority.c

测试目的：

这个测试用于测试当多个不同优先级的线程同时被唤醒时，更高优先级的线程是否比低优先级的线程先进入 running 状态。即这个测试按照 $\text{priority} = \text{PRI_EDFALUT} - (i + 5) \% 10 - 1$ ($i:0 \rightarrow 9$) 的规则先后创建了 10 个线程，将它们 block 掉，并

设置同时间唤醒。若被唤醒后进入 running 状态的顺序是由优先级从高到低，则通过测试；若不然则测试失败。

过程分析：

进入 test_alarm_priority 函数，它首先定义了一个为 wake_time 变量进行赋值，wake_time 用来确定休眠线程被唤醒的时间：

```
wake_time = timer_ticks () + 5 * TIMER_FREQ;
```

接下来为一个 for 循环，用来创建 10 个线程：

```
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);
}
```

【分析】创建的线程按照 PRI_DEFAULT-(i+5)%10-1 的规则设定优先级（即 25, 24, 23, 22, 21, 20, 19, 18, 17, 16），然后调用 thread_create 初始化线程，并将之放入 ready_list 中。thread_create 函数前面已经分析过，这里不再深入分析。传入 thread_create 函数的第三个参数是 alarm_priority_thread，即当 cpu 分配给该线程时，将执行 alarm_priority_thread 函数。

最后又是一个 for 循环：

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

循环调用 sema_down 函数，让我们进入该函数一探究竟：

```
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);
```

【分析】sema_down 函数的主要部分是这个 while 循环，这是一个原子操作。判断当 value 为 0 的时候，调用 list_push_back 函数将正在 running 的线程放入 waiters 链表的最尾端，并调用 thread_block() 函数将其阻塞掉。注意这个正在 running 的线程正是 sema_down 函数，当它被阻塞时，

test_alarm_priority 函数中的这个 for 循环也就会被阻塞掉。那么这个线程什么时候会被唤醒呢？

当 cpu 分配给前面创建好的线程时，就会调用 alarm_priority_thread 函数。下面进入该函数进行分析：

```
timer_sleep (wake_time - timer_ticks ());
```

【分析】可以看到，每个线程休眠的时间均为 wake_time - timer_ticks 的形式。由于 wake_time 的值是固定的，但 timer_ticks 的返回值是一直在递增的，所以尽管这 10 个线程休眠的时间长度不同，但是唤醒的时间点是相同的。

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

【分析】接下来是打印出被唤醒的线程的优先级信息。当该线程结束休眠，重新回到 ready_list 中，再次被分配到 cpu 时，该语句就会被执行。

```
sema_up (&wait_sema);
```

【分析】最后一条语句是极为关键的，它的作用就是唤醒之前被 block 了的 sema_down 函数。

进入 sema_up 函数进一步分析：

```
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);
```

【分析】取 waiters 链表中的第一个元素（也就是之前被 block 了的 sema_down 线程），调用 thread_unblock 函数将其唤醒。之后继续执行 sema->value++ 语句，这时候 value 的值应该为 1。于是 sema_down 线程就被唤醒了。

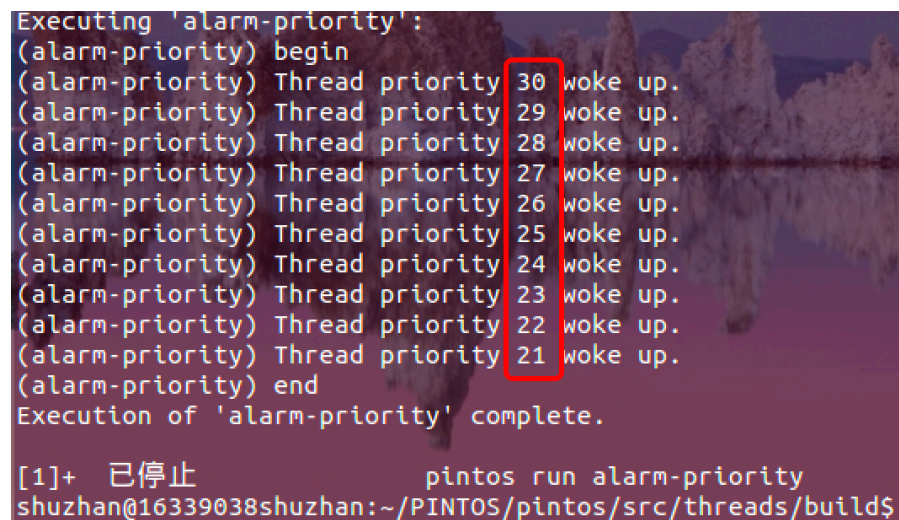
那么就让我们回到 sema_down 函数，继续进行分析。

```
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);
```


【分析】当该线程被唤醒后，while 循环继续执行，回到最初的判断，这时候 value 的值为 1，不满足条件，退出循环。然后继续执行 sema->value—语句，value 的值恢复为 0，sema_down 函数执行完毕。之后回到 test_alarm_priority 中的 for 循环，重复之前的步骤，被阻塞掉，再次等待被 sema_up 函数唤醒，如此往复，直到十次循环完毕，test_alarm_priority 主线程也就结束了。

结果分析：

对 alarm-priority.c 文件进行单个测试，输出如下：



```
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.

[1]+ 已停止                  pintos run alarm-priority
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$
```

【分析】可以看到，被唤醒的线程的确是按照优先级高低进入 running 状态的，测试通过。

• alarm-simultaneous.c

测试目的：

创建 N 个线程，每个线程都休眠固定且不相同的时间 M 次。记录线程被唤醒的顺序，然后与预期的正确结果相比较。如果完全符合则通过测试，如果不一样则测试失败。注意：这里与 alarm-priority 不同的地方在于，每次循环所有的线程是同时被唤醒的。

过程分析：

首先定义了一个结构体 sleep_test, 里面的成员变量的含义与 alarm-priority 中的相同，不再分析。

进入 test_sleep 函数，前半部分是一些打印基本信息、初始化相关变量的操作；下面主要分析一下中间的循环部分：

```

/* Start threads. */
ASSERT (output != NULL);
for (i = 0; i < thread_cnt; i++)
{
    char name[16];
    snprintf (name, sizeof name, "thread %d", i);
    thread_create (name, PRI_DEFAULT, sleeper, &test);
}

```

【分析】通过 for 循环，创建 thread_cnt 个 thread，并将它们依次放入 ready_list 中。循环中第三行传入 thread_create 中的优先级参数为 PRI_DEFAULT，即这个 thread_cnt 个线程的优先级都是相同的。当 cpu 被分配给这些线程时，调用 sleeper 函数。

下面进入 sleeper 函数进行分析：

```

for (i = 1; i <= test->iterations; i++)
{
    int64_t sleep_until = test->start + i * 10;
    timer_sleep (sleep_until - timer_ticks ());
    *test->output_pos++ = timer_ticks () - test->start;
    thread_yield ();
}

```

【分析】在同一个 iteration 内，不同的线程休眠的时间为 sleep_until-timer_ticks() 个 ticks, timer_ticks 是不断增加的，这保证了：每个线程休眠的时间不同，但是唤醒的时间点是相同的；而不同的 iteration 之间，又相差了 10 个 ticks。即，同一循环内，每个线程休眠不同的时间，在同一时间被唤醒，而下一循环线程被唤醒间隔了 10 个 ticks。当线程被重新唤醒，回到 ready_list 中，并被分配到 cpu 后，执行循环内第三行语句，即将 timer_ticks()-test->start 按顺序放入到 output 数组中。然后调用 thread_yield() 将该线程扔回到 ready_list 中。在同一循环内，所有线程的第 3、4 条语句是在同一个 tick 内执行的。

结果分析：

```

(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.

[2]+ 已停止                  pintos run alarm-simultaneous
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$

```

【分析】每个循环内，所有线程在同一时间被唤醒，并且间隔 10 个 ticks，所有线程又再次被唤醒。

(二) 实验思路与代码分析

• 休眠:

本次实验需要从 timer_sleep 开始修改，进入 timer_sleep 函数，进行分析：

```

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

```

【分析】这个函数主要是通过一个 while 循环来达到休眠目的的。当休眠时间少于 ticks 时，就调用 thread_yield() 函数，将其扔回到 ready_list 中。这样做的弊端是：休眠的线程仍然可能会在休眠时间占用 cpu，浪费了资源。因为休眠线程被 thread_yield 扔回了 ready_list，所以线程并没有真正地休眠。解决思路：将需要休眠的线程真正地 BLOCK 掉，即不将其放回到 ready_list 中，然其在休眠时间没有机会拿到 cpu，从而达到节省资源的目的。修改后的 timer_sleep 代码如下：

```

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);

```

【分析】如上图，这段代码做的事情是：拿到当前正在 running 的线程，为其设置要休眠的时间 ticks，并调用 thread_block 函数使其真正地处于休眠状态。thread_block 函数将该线程状态设置为 THREAD_BLOCKED，并调用 schedule 方法调度下一个线程。这里不

再深入分析。（注：timer_sleep 函数开头需要有一个判断，是为了通过 alarm_zero 和 alarm_negative 两个测试）

- 唤醒：

那么休眠的函数如何被再次唤醒呢？我们知道，在 pintos 中，每隔 10ms 就会调用一次 timer_interrupt 进行强制中断，以获取所有线程的状态信息。所以解决思路是：在每次强制中断时，调用 thread_foreach 函数来遍历 all_list，检查休眠函数的休眠时间是否已经结束，若满足要求，则调用 thread_unblock 函数使其结束休眠，重回 ready_list。

下面进入 timer_interrupt 函数进行分析：

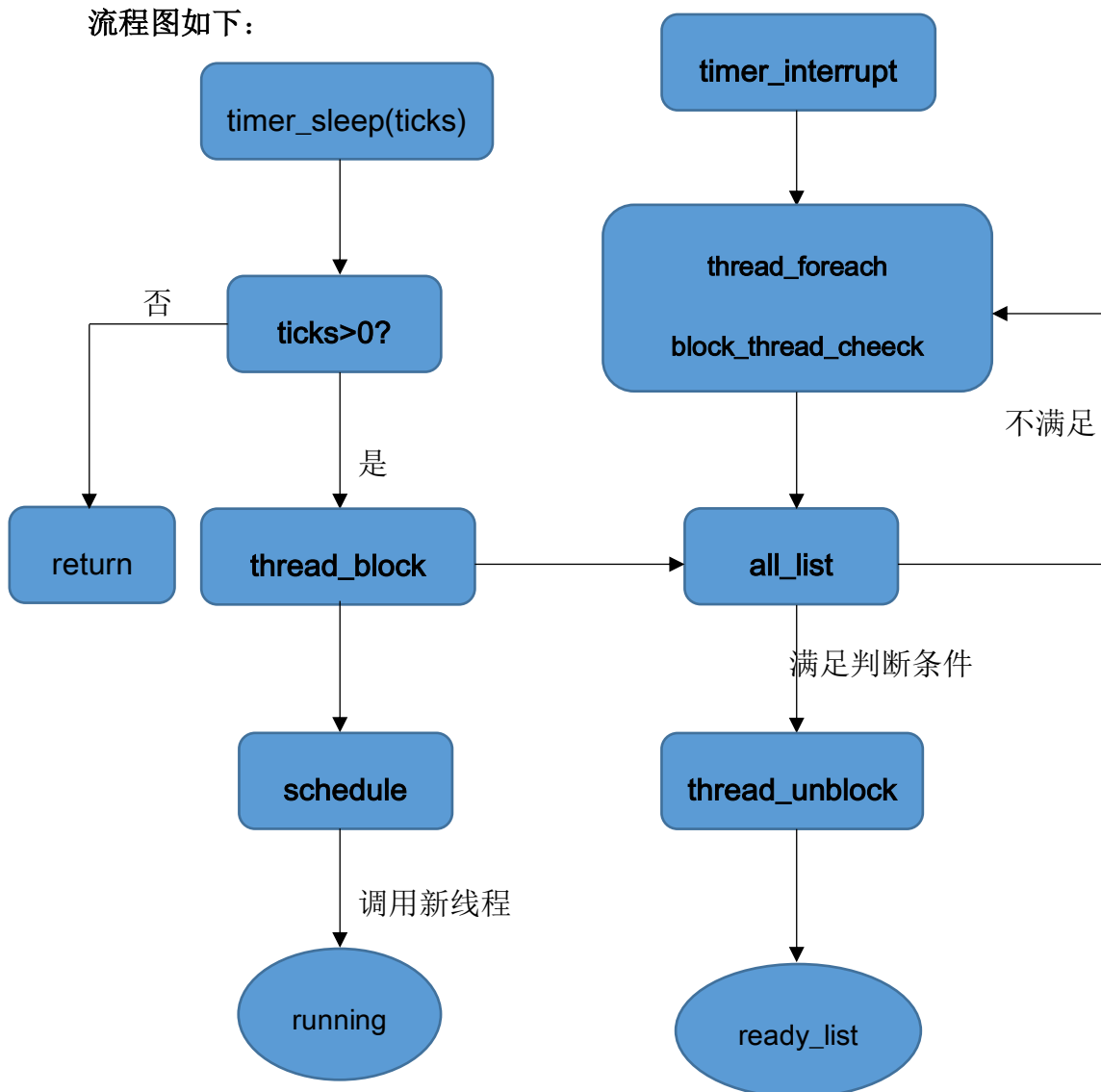
```
ticks++;  
thread_tick ();  
thread_foreach (blocked_thread_check, NULL);
```

【分析】最后添加了一条 thread_foreach 语句，传入的参数为 blocked_thread_check 函数，用来检查处于休眠状态的线程休眠时间是否已结束。这个函数是自己添加的，我将其放在 thread.c 文件中，进入该函数进行分析：

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

【分析】先进行一个判断，即该线程是否是处于阻塞状态，并且其剩余时间是否大于 0。若满足条件，则将其剩余休眠时间减少 1 个 tick；接着继续判断剩余休眠时间是否为 0，若满足，则调用 thread_unblock 结束休眠。代码中的 ticks_blocked 为 int64_t 型变量，添加中 thread 结构体中；其初始化是在 thread_create 函数中进行：当一个线程被创建时，将其初始化为 0；最后，在调用 timer_sleep 函数时，这个变量将被赋予传进来的参数 ticks 的值。以上便是休眠与唤醒的整体思路。

流程图如下：



- 优先级调度：

原始的代码是按照先进先出的规则来将线程放入 ready_list 的，并且每次调度线程都是调度 ready_list 的第一个元素，这就违背了优先级高的先调度的原则，无法通过 alarm-priority 测试。修改的思路有很多，如：每次调度是选取 ready_list 中最高优先级的进行调度；每次插入按照优先级的大小插入到正确的位置，维护其顺序；每次直接插入尾部，然后对整个 ready_list 进行排序。我的修改方法是每次在插入时调用 list_insert_ordered，与插入后调用 sort 进行排序相比，要节约不少时间。

list_insert_ordered 的声明如下：

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

需要传入一个 list_less_func 进行优先级大小比较，这个函数是需要自己定义的。我将其定义在 thread.c 文件中：

```
bool
cmpac(const struct list_elem *thread1, const struct list_elem *thread2, void *aux UNUSED)
{
    if (list_entry(thread1, struct thread, elem)->priority >
        list_entry(thread2, struct thread, elem)->priority)
        return true;
    else
        return false;
}
```

【分析】这个函数实现非常简单，调用 `list_entry` 方法，比较两个线程的优先级，大于则返回真，否则返回假。

那么哪些地方需要修改代码呢？按照思路，修改方法是将先进先出改为按照优先级排列，所以将所有 `list_push_back` 改为 `list_insert_ordered` 即可。在 `thread.c` 文件中 `ctr+f` 进行快速查找，搜索到有三处地方出现了 `list_push_back` 函数，如下：
`thread_unblock` 中：

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

`threads_yield` 中

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

`init_thread` 中：

```
list_push_back (&all_list, &t->allelem);
```

将它们全部替换为 `list_insert_ordered` 即可。

3. 实验结果

每个测试的实验结果已经在 `test` 文件分析部分体现和分析了，这里不再分析。下面为 `make check` 后的总的测试结果：

```
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
19 of 27 tests failed.
make: *** [check] 错误 1
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$
```

```
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
```

如上面两幅图所示，与最初相比，多通过了一个测试，这个测试正是 `alarm-priority`。

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?

- **alarm-zero\alarm-negative:**

这两个 test 在没有修改时能够过的原因:

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

如图中判断, timer_elapsed 的返回值永远是非负数, 若 ticks 的值是非正数, 那么循环体就不会执行, 程序也就不会出错。但修改代码后, 如下图:

```
current_thread->ticks_blocked = ticks;
thread_block ();
```

不会检查 ticks 是否为正数, 直接赋值给 ticks_blocked, 然后被阻塞掉。这样子休眠的函数将永远无法被唤醒! 所以修改后, 要在 timer_sleep 最开头加如下判断:

```
if (ticks <= 0)
{
    return;
}
```

- **alarm-single\alarm-multiple\alarm-simultaneous:**

这三个测试在代码没修改前能过的本质原因都是一样的, 即在为修改代码之前, ready_list 是按照先进先出的方式实现的, 所以无论创建的线程优先级是否都一样, 最终的结果都是相同的。因为先进先出的实现方式是不看优先级的, 只与这些线程被创建的顺序有关, 所以测试可以 pass。

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

```
/* Disables interrupts and returns the previous interrupt status. */
enum intr_level
intr_disable (void)
{
```

如图, 返回的是中断关闭之前的中断状态。

调用 intr_set_level() 函数的原因:

因为在进行完原子操作后, 若不调用此函数恢复到原来的中断状态, (例如原子操作前中断是打开的) 则中断状态将一直保持关闭, 操作系统将无法进行强制中断操作, 从而可能造成严重的后果。所以需要调用此语句恢复到原来的中断状态。

- 3. 什么情况下 schedule 调度的还是当前线程?

当只有一个线程在跑而 ready_list 为空或者该线程的优先级在 ready_list 中为最高的时候, schedule 调度的还是当前线程。

● 4. 为什么线程休眠要保证中断打开？

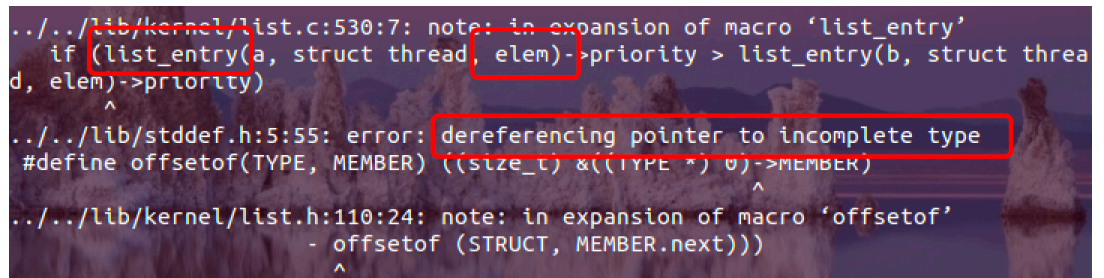
因为，每一个 tick，操作系统会产生一次强制中断，用以得知全部线程的情况；也用以得知休眠线程的休眠时间是否已经结束，是否应该被唤醒。若中断是关闭的，操作系统将无法得知休眠线程的情况，休眠线程也就无法再被唤醒。

5. 实验感想

- 对实验过程中遇到的小问题、困难与疑问的总结：

本次实验虽说跑出 19/27 的结果不是太困难，但在实验已经分析代码过程中还是存在着不少疑惑，下面一一列出：

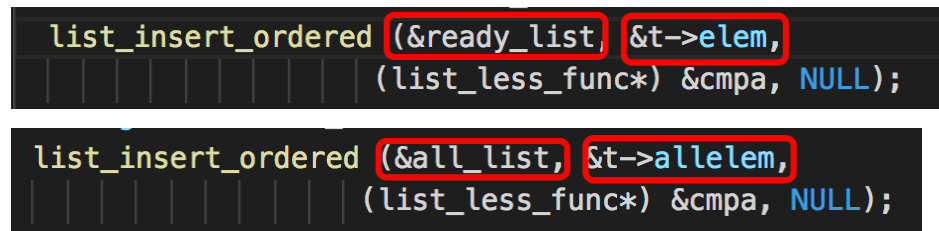
- 1、我实现 list_insert_ordered 需要的比较函数时，第一次是下意识地将它定义在 list.c 函数中，因为它的调用者 list_insert_ordered 函数是定义在 list.c 文件中的，但是会报如下错误：



```
../../lib/kernel/list.c:530:7: note: in expansion of macro 'list_entry'
  if (list_entry(a, struct thread, elem)->priority > list_entry(b, struct thread, elem)->priority)
      ^
../../lib/stddef.h:5:55: error: dereferencing pointer to incomplete type
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *) 0)->MEMBER)
                                  ^
../../lib/kernel/list.h:110:24: note: in expansion of macro 'offsetof'
  - offsetof (STRUCT, MEMBER.next))
    ^
```

仔细分析，原来是 list_entry 中含有参数 elem，这个参数定义在 thread.h 文件中的 thread 结构体中，而 list.h 文件并没有 include thread.c 文件。所以将这个函数直接放在 thread.c 中，就不会出现报错了。

- 2、有一个细节要特别注意，如下图：



```
list_insert_ordered (&ready_list, &t->elem,
                    (list_less_func*) &cempa, NULL);

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

向 ready_list 中插入线程一定要使用 elem，向 all_list 中插入线程一定要使用 allelem，不要弄混了。

- 3、最后有一个比较大的疑惑，对于 priority 测试，它的 10 个线程创建的顺序是 25\24\23\22\21\30\29\28\27\26，并且在创建的时候是依次将它们放入 ready_list 的。所以当最开始没有修改代码时，按照先进先出的规则，它们唤醒的顺序应该也是如此，但是查看其 output 文件，真实唤醒顺序如下，即出现了乱序：

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

目前还未能分析出产生此结果的原因，希望在今后的学习中可以解决。