

# 中山大学移动信息工程学院本科生实验报告

(2016 学年秋季学期)

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	1403	专业(方向)	移动互联网
学号	14353435	姓名	朱恩东
电话	15919154005	Email	<a href="mailto:562870140@qq.com">562870140@qq.com</a>
开始日期	2016. 3. 30	完成日期	2016. 4. 3

## 1. 实验目的

通过修改 pintos 休眠的实现机制, 使线程在休眠过程中不会占用 cpu 资源。同时修改就绪队列的插入方式, 使得就绪队列中的线程始终按照优先级排序。

## 2. 实验过程

(1) test 解释:

alarm-zero:

首先我们来看源代码:

```
9
10 void
11 test_alarm_zero (void)
12 {
13     timer_sleep (0);
14     pass ();
15 }
16
```

不难看出alarm-zero只调用一次timer\_sleep函数, 其中传入的参数为0, 使线程休眠0个tick, 也即不进行休眠, 应该直接返回。

alarm-negative:

首先我们来看源代码:

```
9
10 void
11 test_alarm_negative (void)
12 {
13     timer_sleep (-100);
14     pass ();
15 }
```

不难看出该测试与alarm-zero十分类似, 都只调用一次timer\_sleep函数, 其中传入的参数为-100, 显然线程的休眠时间不可能为负值, 对此类非法值应该不进行休眠, 直接返回。

alarm-simultaneous:

首先我们来看源代码:

```
15 void
16 test_alarm_simultaneous (void)
17 {
18     test_sleep (3, 5);
19 }
20
```

而test\_sleep函数定义如下:

```
/* Runs THREAD_CNT threads thread sleep ITERATIONS times each. */
static void
test_sleep (int thread_cnt, int iterations)
{
```

它接受两个参数，一个是创造的线程数 `thread_cnt`，另一个是每个线程的迭代次数 `iterations`。因此该测试会创建 3 个子线程，其中每个线程会进行五次迭代休眠，每次迭代休眠 10 个 ticks。由于创建的三个线程是同时进入休眠且休眠相同时间，因此在每一次迭代中，三个线程都应该是在同一个 tick 被唤醒。从输出结果中应该要能够，一旦第一个线程苏醒后，其余两个线程在经过 0 个 ticks 后就应该苏醒。

alarm-single:

首先我们来看源代码：

```
15 void
16 test_alarm_single (void)
17 {
18     test_sleep (5, 1);
19 }
```

而与上一个测试相似的，`test_sleep` 定义如下：

```
/* Runs THREAD_CNT threads thread sleep ITERATIONS times each.
static void
test_sleep (int thread_cnt, int iterations)
{
    struct sleep_test test;
```

而 `alarm-single` 传入 5, 1 两个参数，代表会创建 5 个线程，每个线程休眠一次。而与之之前不同的是，这 5 个线程依然同时被创建，但有着不同的休眠时间，由 `thread0` 到 `thread1` 依次休眠 10, 20... 50 个 ticks。因此，预计的输出结果应该是在 10, 20... 50 个 ticks 的时刻，分别为 `thread0`-`thread1` 依次醒来。

alarm-multiple:

首先我们来看源代码：

```
21 void
22 test_alarm_multiple (void)
23 {
24     test_sleep (5, 7);
25 }
```

`test_sleep` 定义如下：

```
/* Runs THREAD_CNT threads thread sleep ITERATIONS times each.
static void
test_sleep (int thread_cnt, int iterations)
{
    struct sleep_test test;
```

而 `alarm-multiple` 传入 5, 7 两个参数，代表着将会创建 5 个线程，每个线程休眠 7 次。因此 `alarm-multiple` 与 `alarm-single` 最大的区别在于 `multiple` 中的线程在被唤醒一次后会继续进入休眠状态，而又由于不同线程的不同休眠时间影响，各线程不再是按顺序依次醒来。预计的输出结果应该是 ticks=10 时刻 `thread0` 醒来，ticks=20 时刻 `thread0` 和 `thread1` 同时醒来，依次类推..... 最终 `thread0` 最先完成 7 次休眠后停止执行。

alarm-priority:

这部分代码比较长，首先我们来看源代码：

```

16 void
17 test_alarm_priority (void)
18 {
19     int i;
20
21     /* This test does not work with the MLFQS. */
22     ASSERT (!thread_mlfqs);
23
24     wake_time = timer_ticks () + 5 * TIMER_FREQ;
25     sema_init (&wait_sema, 0);
26
27     for (i = 0; i < 10; i++)
28     {
29         int priority = PRI_DEFAULT - (i + 5) % 10 - 1;
30         char name[16];
31         snprintf (name, sizeof name, "priority %d", priority);
32         thread_create (name, priority, alarm_priority_thread, NULL);
33     }
34
35     thread_set_priority (PRI_MIN);
36
37     for (i = 0; i < 10; i++)
38         sema_down (&wait_sema);
39 }

```

这个函数是在创建线程，首先这个函数创建了一个变量 `wake_time`，用来存放线程的苏醒时间，这在之后会用到。`TIMER_FREQ` 的定义在 `timer.h` 中可以找到，为 100，至于为什么要 `5 * TIMER_FREQ`，我认为是因为要设置一个较长的时间来使得之后的操作不会超过线程的苏醒时间。

之后的 `for` 循环是在创建线程，同时给不同的线程以不同的优先级，优先级的计算由公式给出：

```
int priority = PRI_DEFAULT - (i + 5) % 10 - 1;
```

其中 `PRI_DEFAULT` 定义在 `thread.h` 中，值为 31，因此可以知道线程的优先级分别为：25, 24, ... 21, 30, 29, ... , 26。

```

41 static void
42 alarm_priority_thread (void *aux UNUSED)
43 {
44     /* Busy-wait until the current time changes. */
45     int64_t start_time = timer_ticks ();
46     while (timer_elapsed (start_time) == 0)
47         continue;
48
49     /* Now we know we're at the very beginning of a timer tick, so
50      * we can call timer_sleep() without worrying about races
51      * between checking the time and a timer interrupt. */
52     timer_sleep (wake_time - timer_ticks ());
53
54     /* Print a message on wake-up. */
55     msg ("Thread %s woke up.", thread_name ());
56
57     sema_up (&wait_sema);
58 }
59

```

该函数用于执行线程休眠。值得注意的是 `timer_sleep` 中的参数是 `wake_time` 与当前 tick 的差值，也就是说所有的线程都将在同一时刻苏醒。而苏醒后的线程应该按照优先级的先后顺序来执行。因此预计的运行结果应该是优先级为 30-21 的线程按照降序排列。

### (2) 实验思路:

本次实验思路主要分为两部分: 一个是修改休眠函数, 使其一个进程在休眠的时候不会处于忙等待的状态; 另一个是修改线程队列的插入过程, 使得其中的线程始终按照优先级的大小进行排序。

对于第一个改进思路, 我们的方法是不再使用 while 语句进行轮询的操作来确定一个线程是否休眠完毕, 而是在每一次中断时判断该线程是否休眠完毕。因此我们可以在原本 thread 的定义中增加一个变量来记录线程应该休眠多长时间, 线程在非 blocked 情况下该变量为 0。我们接着修改中断函数, 在每一次中断中遍历当前线程链表, 如果有正在休眠的线程, 则判断是否已达到设定的休眠时间。

对于第二个改进思路, 我们的方法是使用内置的 list\_insert\_ordered 函数, 通过我们自定义的 cmp 函数, 保证每次插入链表元素时是按照优先级顺序插入的。这里吧 thread\_yield 和 thread\_unblock 这两个函数中原本直接插入的函数改为我们定以后的 list\_insert\_ordered 函数。

### (3) 代码分析:

```
83  struct thread
84  {
85      /* Owned by thread.c. */
86      tid_t tid;
87      enum thread_status status;
88      char name[16];
89      uint8_t *stack;
90      int priority;
91      struct list_elem allelem;
92      /* Shared between thread.c and userprog.c */
93      struct list_elem elem;
94
95      #ifdef USERPROG
96      /* Owned by userprog/process.c */
97      uint32_t *pagedir;
98      #endif
99
100     int ticks_blocked;
101     /* Owned by thread.c. */
102     unsigned magic;
103 };
```

这里是修改后的 thread 结构体的定义, 增加了一个 ticks\_blocked 变量, 所以要记得在 thread\_create 中将其默认值置为 0 (因为不算复杂, 所以这里不再贴出修改的 thread\_create 函数)。

```
89  void
90  timer_sleep (int64_t ticks)
91  {
92      if (ticks <= 0) return ;
93
94      struct thread* cur = thread_current() ;
95      enum intr_level old ;
96      old = intr_disable() ;
97      cur->ticks_blocked = ticks ;
98      thread_block() ;
99      intr_set_level(old) ;
100 }
```

这里是修改后的 timer\_sleep 函数, 这里需要注意的是休眠时间的合法值应该是大于 0 的, 因此遇到 ticks 小于等于 0 的情况应该直接返回。若输入的 ticks 合法, 我们只需改变当前 thread 的 ticks\_blocked 为需要休眠的时间 ticks, 再将当前 thread 的状态更改为 blocked 即可。这里要注意再对 thread 操作时中断必须处于关闭状态。



```

173 static void
174 timer_interrupt (struct intr_frame *args UNUSED)
175 {
176     ticks++;
177     enum intr_level old = intr_disable() ;
178     thread_foreach(blocked_thread_check, NULL) ;
179     intr_set_level(old) ;
180     thread_tick () ;
181 }

```

这里是修改后的中断函数，在一次中断中，我们除了常规地增加系统时间 ticks 之外，还需遍历整个线程链表并用 blocked\_thread\_check 函数检查是否有应该唤醒的休眠线程。这里同样需要注意对链表遍历时必须保证中断关闭。

```

590 void blocked_thread_check(struct thread* t, void *aux UNUSED)
591 {
592     if(t->status == THREAD_BLOCKED && t->ticks_blocked != 0)
593     {
594         t->ticks_blocked-- ;
595         if(t->ticks_blocked == 0)
596         {
597             thread_unblock(t) ;
598         }
599     }
600 }

```

这里是在中断中使用的 blocked\_thread\_check 函数的定义，可以看到该函数先判断当前线程是否处于休眠状态，且休眠时间是否为合法值大于 0，当满足条件时，先对该线程的剩余休眠时间减一，然后看该线程是否已该被唤醒，若是的话就将该线程 unblock 掉。

以上是修改休眠与唤醒方式的部分。接下来我们看一下修改后保证按照优先级顺序插入的代码部分：

```

601
602 bool cmp_elem(struct list_elem *x, struct list_elem *y)
603 {
604     struct thread *thread_a = list_entry(x, struct thread, elem);
605     struct thread *thread_b = list_entry(y, struct thread, elem);
606     return thread_a->priority > thread_b->priority;
607 }

```

以上是我自定义的 cmp\_elem 函数，它接收两个 list\_elem 类型的结构体 x 和 y，同时通过宏定义的 list\_entry 函数得到 x 和 y 所处的 thread 的地址，分别储存在两个指针中。最后我们根据两个 thread 之间的优先级返回两者先后顺序。

```

239 void
240 thread_unblock (struct thread *t)
241 {
242     enum intr_level old_level;
243
244     ASSERT (is_thread (t));
245
246     old_level = intr_disable ();
247     ASSERT (t->status == THREAD_BLOCKED);
248     list_insert_ordered(&ready_list, &t->elem, cmp_elem, NULL);
249     t->status = THREAD_READY;
250     intr_set_level (old_level);
251 }

```

```

309 void
310 thread_yield (void)
311 {
312     struct thread *cur = thread_current ();
313     enum intr_level old_level;
314
315     ASSERT (!intr_context ());
316
317     old_level = intr_disable ();
318     if (cur != idle_thread)
319         list_insert_ordered(&ready_list, &cur->elem, cmp_elem, NULL);
320     cur->status = THREAD_READY;
321     schedule ();
322     intr_set_level (old_level);
323 }

```

以上是修改后的 thread\_unblock 和 thread\_yield 函数,因为这两个函数中涉及到了插入操作,因此我们必须保证插入链表是有序的。因此要使用 list\_insert\_ordered,同样要注意插入时应该保证中断关闭。

### 3. 实验结果

各 test 通过情况:

```

zhu@zed14353435: ~/pintos/src/threads/build
(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/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
zhu@zed14353435:~/pintos/src/threads/build$

```

alarm-zero 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads/build 00:25

=====
Bochs x86 Emulator 2.6.2
Built from SVN snapshot on May 26, 2013
Compiled on Mar 25 2016 at 20:40:44
=====

00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keybo
ard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
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.

zhu@zed14353435:~/pintos/src/threads/build$ =====
Bochs is exiting with the following message:
[      ] SIGNAL 2 caught
=====

zhu@zed14353435:~/pintos/src/threads/build$
```

alarm-negative 运行情况:



```
zhu@zed14353435: ~/pintos/src/threads/build 00:27

=====
Bochs x86 Emulator 2.6.2
Built from SVN snapshot on May 26, 2013
Compiled on Mar 25 2016 at 20:40:44
=====

00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keybo
ard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
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.

zhu@zed14353435:~/pintos/src/threads/build$ =====
Bochs is exiting with the following message:
[      ] SIGNAL 2 caught
=====

zhu@zed14353435:~/pintos/src/threads/build$
```

alarm-simultaneous 运行情况:



```
zhu@zed14353435: ~/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.

zhu@zed14353435:~/pintos/src/threads/build$ =====
=====
Bochs is exiting with the following message:
[      ] SIGNAL 2 caught
=====

zhu@zed14353435:~/pintos/src/threads/build$
```

alarm-single 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads/build
00000000000i[      ] using log file bochsout.txt
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.

zhu@zed14353435:~/pintos/src/threads/build$ =====
=====
Bochs is exiting with the following message:
[      ] SIGNAL 2 caught
=====

zhu@zed14353435:~/pintos/src/threads/build$
```

alarm-multiple 运行情况:



```
zhu@zed14353435: ~/pintos/src/threads/build 00:31
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
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

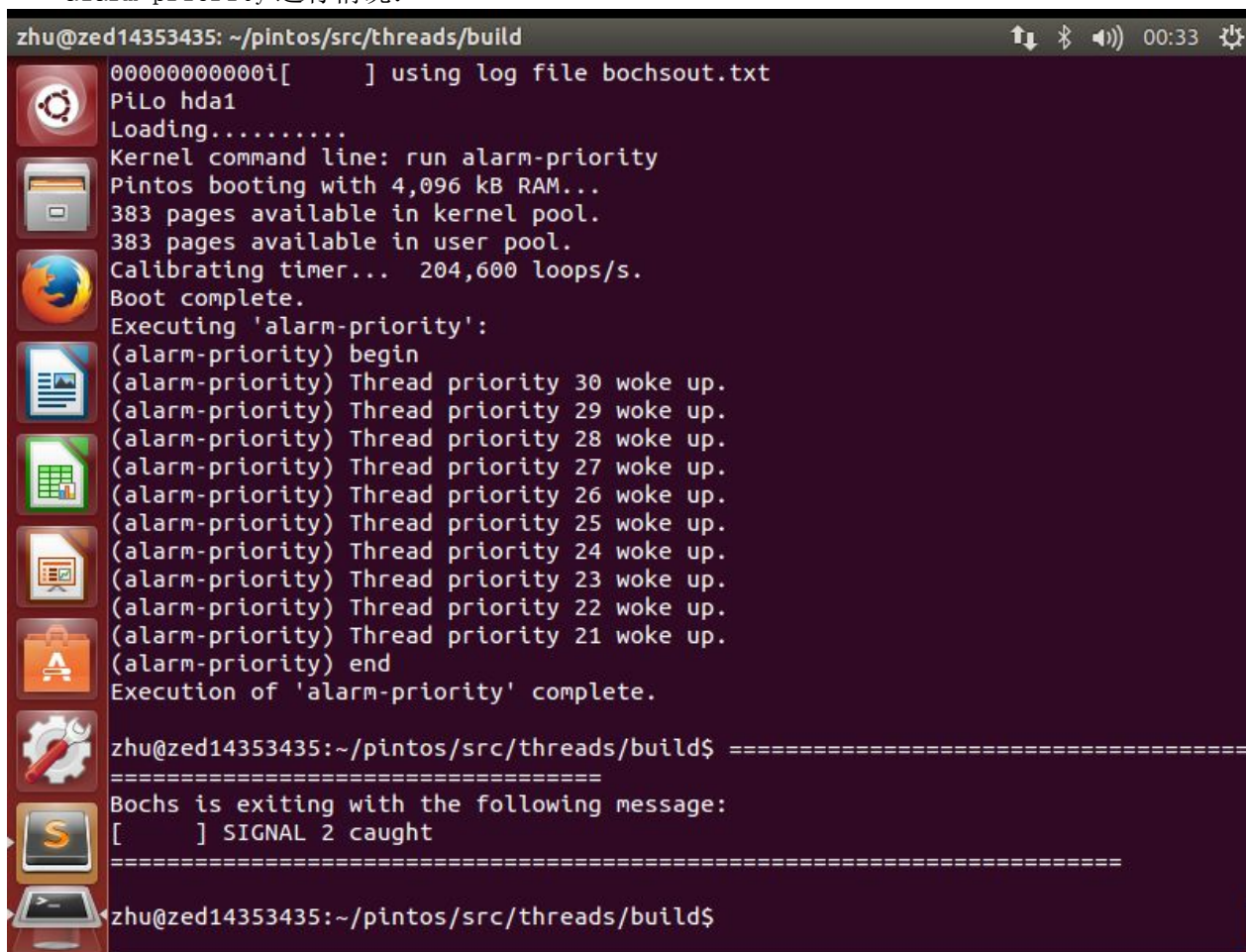
zhu@zed14353435: ~/pintos/src/threads/build 00:32
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(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.

zhu@zed14353435:~/pintos/src/threads/build$ =====
=====
Bochs is exiting with the following message:
[    ] SIGNAL 2 caught
=====

zhu@zed14353435:~/pintos/src/threads/build$
```



alarm-priority 运行情况:



```
zhu@zed14353435: ~/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.

zhu@zed14353435:~/pintos/src/threads/build$ =====
=====
Bochs is exiting with the following message:
[      ] SIGNAL 2 caught
=====

zhu@zed14353435:~/pintos/src/threads/build$
```

## 4. 回答问题

(1)

之前结果为 20/27, 也即 alarm 中除 alarm-priority 之外的测试均可以正常通过。我觉得原因在于, 之前使用忙等待的轮询方式来实现休眠, 是可以正常完成休眠与唤醒的工作的, 只不过因为忙等待消耗了 CPU 多余的资源而被认为是不合理的。而本次 alarm-test 中其他几个 test 测试的分别为: 休眠 0 个 ticks, 休眠负数个 ticks, 休眠单次, 休眠多次, 多个进程同时休眠同时唤醒, 这些测试的均为休眠与唤醒的基本功能是否完整。而是用忙等待的方式是可以保证这些休眠与唤醒的基本功能的。因此, 之前不做修改也可以通过这些测试。而在 alarm-priority 中, 我们必须保证就绪队列中的进程按照优先级的顺序来插入, 这一点是之前的代码中没有实现的, 因此无法通过测试。

(2)

简单的来说, 进程是资源的拥有和分配者, 而线程则是指令的执行者。一个进程是正在运行中的程序, 它拥有着资源, 并且以线程为单位来分配这些资源。因此, 一个进程往往是可以拥有多个线程的。一个进程中的各个线程共享内存和进程资源。

(3)

我认为中断和轮询最大的一个区别在于, 轮询是消耗 CPU 的资源来主动去查看是否需要终止当前 CPU 进程来用于 IO 操作等; 而中断则是在需要进行 IO 等操作的时候来请求中断, 否则 CPU 是正常执行的。因此中断相比轮询来说, 提高了 CPU 的利用率。以我们本次的实验为例, 原本忙等待的实现方式是在 timer\_sleep 中使用一个 while 语句, 每次判断是否到达唤醒时间, 若达到了唤醒时

间才跳出循环，否则会一直卡在 while 循环中。而我们修改之后的实现方式是在每一次系统中断获取系统时间的时候再来判断是否达到唤醒时间，其他时刻 CPU 并没有被占用，因此运行效率更加高。

## 5. 实验感想

本次实验是关于线程的休眠与唤醒，需要修改原本基于轮询实现的忙等待休眠，实现基于中断的 block 休眠方式。

本次实验中遇到的一个问题是在修改完成后 make check 时陷入死循环的问题，而且也无法通过。经过检查才发现是在 timer\_sleep 中漏写了判断 ticks 是否小于等于 0 的情况：

```
89 void
90 timer_sleep (int64_t ticks)
91 {
92     if(ticks <= 0) return ;
93
94     struct thread* cur = thread_current() ;
95     enum intr_level old ;
96     old = intr_disable() ;
97     cur->ticks_blocked = ticks ;
98     thread_block() ;
99     intr_set_level(old) ;
100 }
```

也即没有写上面的 if 语句，我们将该进程 block 掉了，但是却把他剩余休眠时间赋值为和其他正常运行进程一样的 0，这样在进行 alarm-zero 的测试时会陷入死循环。这是因为我们在中断中判断线程是否被 block 掉的时候，如下：

```
590 void blocked_thread_check(struct thread* t, void *aux UNUSED)
591 {
592     if(t->status == THREAD_BLOCKED && t->ticks_blocked != 0)
593     {
594         t->ticks_blocked-- ;
595         if(t->ticks_blocked == 0)
596         {
597             thread_unblock(t) ;
598         }
599     }
600 }
```

如果 ticks\_blocked 为 0，则无法进入 if 语句，该线程将永远无法被 unblock 掉，陷入了永久休眠的死循环中。

这个 bug 提醒了我在编写操作系统的代码的时候对一些细节的边界情况需要考虑到位，否则很有可能对于极端的输入情况就会造成错误。

此外，本次实验中还遇到的一个问题是关于编写的比较函数 cmp 的时候，cmp 的存放位置。之前我就近把 cmp 存放在了 timer.c 文件中，结果发现编译一直无法通过，报错信息是 thread 等结构体没有定义。然后发现是因为 timer.c 中没有 thread.h 等头文件的包含，于是我想当然的给 timer.c 加上了这些头文件，结果发现编译之后依然无法通过，而且有了更多的报错信息，主要是各类冲突的报错。后来是把 cmp 函数写在了 thread.c 中才通过了编译。我觉得，在一个工程文件中各个文件是有层次关系的，因此并不是每一个文件都可以随便 include 头文件，否则很容易出现自己 include 自己这样的错误。比如本次实验中，thread.c 显然是较为顶层的文件，它需要包含各类的底层文件，而 timer.c 等文件主要是用来实现一些基本功能的，显然是输入较为底层的文件。因此我们在自定义函数的时候，也要弄清楚这一类的层次关系，尽量不要在一些底层文件中去实现一些高级的操作，否则会引起这些文件层次之间的冲突。