

# 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. 4. 11	完成日期	2016. 4. 14

## 1. 实验目的

通过修改 pintos, 实现优先级的抢占机制。使当前执行线程的优先级被修改后, 如果就绪队列中有优先级更高的线程, 则当前进程应该让出 cpu。同时, 当我们创建一个新线程, 并将其加入就绪队列后, 如果其优先级高于当前执行中的线程, 线程应该主动地抢占 cpu。

## 2. 实验过程

(1) test 解释:

priority-change:

首先我们来看源代码:

```
12 void
13 test_priority_change (void)
14 {
15     /* This test does not work with the MLFQS. */
16     ASSERT (!thread_mlfqs);
17
18     msg ("Creating a high-priority thread 2.");
19     thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL);
20     msg ("Thread 2 should have just lowered its priority.");
21     thread_set_priority (PRI_DEFAULT - 2);
22     msg ("Thread 2 should have just exited.");
23 }
24
25 static void
26 changing_thread (void *aux UNUSED)
27 {
28     msg ("Thread 2 now lowering priority.");
29     thread_set_priority (PRI_DEFAULT - 1);
30     msg ("Thread 2 exiting.");
31 }
```

从测试代码中可以看出, 在主线程执行到 test\_priority\_change 函数时, 创建了一个子线程, 且该子线程的优先级是高于当前正在运行中的主线程的, 因此创建的子线程会成功抢占 cpu。创建后的子线程会执行 changing\_thread 函数, 在该函数中, 子线程的优先级被修改, 修改后的优先级会小于主线程的优先级, 因此主线程会重新抢占回 cpu。主线程在重新抢占回 cpu 继续执行 test\_priority\_change 函数中剩余的内容。然而在执行剩余内容时, 主线程的优先级又被修改为小于子线程的优先级, 因此主线程又得让出 cpu 给子线程执行, 子线程此后顺利执行完 changing\_thread 函数, 就被杀死了, 只剩下主线程一个线程。最后主线程也顺利地执行完剩余内容, 整个测试结束。

因此, 按照我刚才的分析, 输出字符串的顺序应该为: 1->4->2->5->3, 其中数字代表的就是字符串所在行数的先后顺序。

priority-preempt:

首先我们来看源代码:

```
17 void
18 test_priority_preempt (void)
19 {
20     /* This test does not work with the MLFQS. */
21     ASSERT (!thread_mlfqs);
22
23     /* Make sure our priority is the default. */
24     ASSERT (thread_get_priority () == PRI_DEFAULT);
25
26     thread_create ("high-priority", PRI_DEFAULT + 1, simple_thread_func, NULL,
27     msg ("The high-priority thread should have already completed.));
28 }
29
30 static void
31 simple_thread_func (void *aux UNUSED)
32 {
33     int i;
34
35     for (i = 0; i < 5; i++)
36     {
37         msg ("Thread %s iteration %d", thread_name (), i);
38         thread_yield ();
39     }
40     msg ("Thread %s done!", thread_name ());
41 }
```

从测试代码中可以看出,在主线程执行到 test\_priority\_preempt 函数时,创建了一个子线程,且该子线程的优先级会比主线程的高。因此子线程会成功抢占 cpu,并执行 simple\_thread\_func 函数。在该函数中,会执行一个循环语句,该语句会执行 5 次 thread\_yield 函数,即让当前线程让出 cpu。但是由于子线程的优先级始终高于主线程的优先级,每次 yield 之后依然是由子线程优先抢占 cpu。因此,我们应该可以看到子线程顺序地执行输出五次循环。之后子线程执行完 simple\_thread\_func 函数后被杀死,主线程再重新执行 test\_priority\_preempt 函数中的剩余内容。

因此我们应该看到的输出顺序为:2->2->2->2->2->3->1,其中数字代表字符串所处行数的先后顺序。

alarm-simultaneous:

首先我们来看源代码:

```

17
18 struct simple_thread_data
19 {
20     int id; /* Sleeper ID. */
21     int iterations; /* Iterations so far. */
22     struct lock *lock; /* Lock on output. */
23     int **op; /* Output buffer position. */
24 };
25
26 #define THREAD_CNT 16
27 #define ITER_CNT 16
28
29 static thread_func simple_thread_func;
30
31 void
32 test_priority_fifo (void)
33 {
34     struct simple_thread_data data[THREAD_CNT];
35     struct lock lock;
36     int *output, *op;
37     int i, cnt;
38
39     /* This test does not work with the MLFQS. */
40     ASSERT (!thread_mlfqs);
41
42     /* Make sure our priority is the default. */
43     ASSERT (thread_get_priority () == PRI_DEFAULT);
44
45     msg ("%d threads will iterate %d times in the same order each time.",
46         THREAD_CNT, ITER_CNT);
47     msg ("If the order varies then there is a bug.");
48
49     output = op = malloc (sizeof *output * THREAD_CNT * ITER_CNT * 2);
50     ASSERT (output != NULL);
51     lock_init (&lock);
52
53     thread_set_priority (PRI_DEFAULT + 2);
54     for (i = 0; i < THREAD_CNT; i++)
55     {
56         char name[16];
57         struct simple_thread_data *d = data + i;
58         snprintf (name, sizeof name, "%d", i);
59         d->id = i;
60         d->iterations = 0;
61         d->lock = &lock;
62         d->op = &op;
63         thread_create (name, PRI_DEFAULT + 1, simple_thread_func, d);
64     }
65
66     thread_set_priority (PRI_DEFAULT);
67     /* All the other threads now run to termination here. */
68     ASSERT (lock.holder == NULL);

```



```

70     cnt = 0;
71     for (; output < op; output++)
72     {
73         struct simple_thread_data *d;
74
75         ASSERT (*output >= 0 && *output < THREAD_CNT);
76         d = data + *output;
77         if (cnt % THREAD_CNT == 0)
78             printf("(priority-fifo) iteration:");
79         printf(" %d", d->id);
80         if (++cnt % THREAD_CNT == 0)
81             printf("\n");
82         d->iterations++;
83     }
84 }
85
86 static void
87 simple_thread_func (void *data_)
88 {
89     struct simple_thread_data *data = data_;
90     int i;
91
92     for (i = 0; i < ITER_CNT; i++)
93     {
94         lock_acquire (data->lock);
95         *(&data->op)++ = data->id;
96         lock_release (data->lock);
97         thread_yield ();
98     }
99 }

```

从测试代码中可以看出，主线程先执行 test\_priority\_fifo 函数，一开始把主线程的优先级设置为 Default+2，以保证在创建子线程的过程中 cpu 不会被抢占。此后主线程创建了 16 个子线程，每个子线程的优先级相同，为 Default+1。创建完子线程后，将主线程的优先级调为 Default，因此主线程优先级较低让出了 cpu。此后，就绪队列中的第一个子线程开始占用 cpu，并执行 simple\_thread\_func 函数。

在函数中，第一个子线程将自己的 id 储存到结果的记录数组 op 中。此后，执行 thread\_yield 函数，该子线程让出 cpu。但是由于 16 个子线程都有着相同的优先级，因此第一个子线程无法再次抢占到 cpu，而是顺序地由第二个子线程执行。此后每个子线程都顺序地执行过一次，则我们的第一次迭代结束。此后第一个子线程再次抢占到了 cpu，并再次将自己的 id 输入到 op 数组中，这标志着第 2 轮迭代的开始。最后当每个子线程都执行完 16 次迭代后，则所有的子线程都跳出了循环并执行完毕 simple\_thread\_func 函数，被系统杀死。

子线程全部被杀死之后，主线程重新抢占到 cpu，并开始输出结果。由于之前子线程每次执行都把自己的 id 输入到了 op 数组中。因此我们需要循环地将 op 数组中的内容输出，每次输出完一轮迭代的内容，需要输出换行。因为我们将 16 个子线程的优先级设置为一样，因此他们的执行顺序应该满足先入先出的类似队列的顺序。因此期望的输出结果为，16 次迭代的输出序列完全相同。

### (2) 实验思路:

本次实验的思路主要修改两个 thread.c 中的函数。

第一个需要修改的函数是 `set_priority()`，该函数的作用是修改当前执行线程的优先级。因此在修改完当前线程的优先级后，我们应该将修改后的优先级与就绪队列中的队头线程的优先级比较，如果当前线程的优先级小于就绪队列中最大的优先级（也即队头元素的优先级），则我们应该调用 `thread_yield()` 函数，让当前线程让出 cpu。

第二个需要修改的函数是 `thread_create()`，该函数的作用是创建一个新线程，并将其加入就绪队列中。因此在修改完一个新的线程并加入优先队列后，我们应该将当前正在执行的线程优先级与新线程的优先级相比较。如果该线程的优先级大于当前正在执行线程的优先级，则应该调用 `thread_yield` 函数，使当前正在执行的线程让出 cpu。

### (3) 代码分析:

```
344  /* Sets the current thread's priority to NEW_PRIORITY. */
345  void
346  thread_set_priority (int new_priority)
347  {
348      thread_current ()->priority = new_priority ;
349      int max_priority = list_entry(list_begin(&ready_list),
350      struct thread, elem)->priority ;
351      if(new_priority < max_priority) thread_yield();
352  }
```

这里是修改后的 `set_priority()` 函数，首先我们将当前正在执行线程的优先级修改为传入的参数 `new_priority`。之后我们利用宏定义 `list_entry` 来获取到就绪队列中的队头元素，并将其优先级保存在 `max_priority` 变量中。最后我们进行判断，如果当前执行线程的优先级小于就绪队列中最大的优先级，则我们应该执行 `thread_yield` 函数让出 cpu。

```
208
209  /* Add to run queue. */
210  int cur_priority = thread_current()->priority ;
211  thread_unblock (t);
212  if(priority > cur_priority) thread_yield () ;
213
214  return tid;
215  }
```

以上是修改后的 `thread_create()` 函数的一部分，我们首先获取当前正在执行线程的优先级，并将其保存在 `cur_priority` 变量中。接下来，我们把创建的新线程加入到就绪队列中，再执行判断，如果当前正在执行线程的优先级小于我们创建的线程的优先级，则我们应该执行 `thread_create` 函数，使当前正在执行的线程让出 cpu。

## 3. 实验结果

各 test 通过情况:



```
zhu@zed14353435: ~/pintos/src/threads/build
zhu@zed14353435:~$ cd pintos/src/threads/build
zhu@zed14353435:~/pintos/src/threads/build$ make check
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
pass 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
pass tests/threads/priority-fifo
pass 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
16 of 27 tests failed.
```

priority-change 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads/build
Constant subroutine SIGVTALRM redefined at /home/zhu/pintos/src/utils/pintos line 927.
squish-pty bochs -q
=====
Bochs x86 Emulator 2.6.2
Built from SVN snapshot on May 26, 2013
Compiled on Mar 25 2016 at 20:40:44
=====
000000000000i[      ] reading configuration from bochsrc.txt
000000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyboard' option.
000000000000i[      ] installing nogui module as the Bochs GUI
000000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: run priority-change
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 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 now lowering priority.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 exiting.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.
```

priority-preempt 运行情况:



```
zhu@zed14353435: ~/pintos/src/threads/build 20:15
squish-pty bochs -q
=====
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 priority-preempt
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 'priority-preempt':
(priority-preempt) begin
(priority-preempt) Thread high-priority iteration 0
(priority-preempt) Thread high-priority iteration 1
(priority-preempt) Thread high-priority iteration 2
(priority-preempt) Thread high-priority iteration 3
(priority-preempt) Thread high-priority iteration 4
(priority-preempt) Thread high-priority done!
(priority-preempt) The high-priority thread should have already completed.
(priority-preempt) end
Execution of 'priority-preempt' complete.
```

priority-fifo 运行情况:

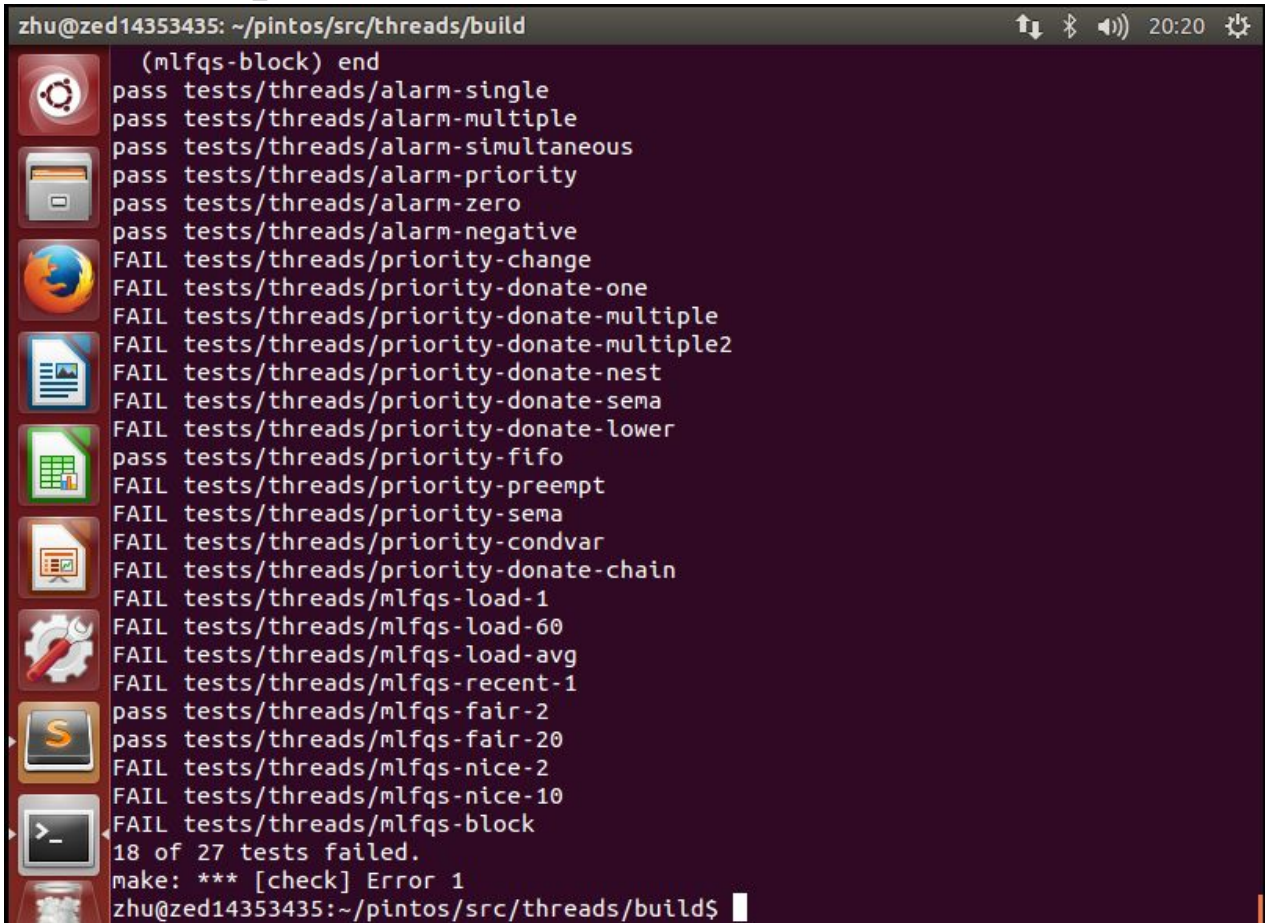
```
zhu@zed14353435: ~/pintos/src/threads/build 20:16
PiLo hda1
Loading.....
Kernel command line: run priority-fifo
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'priority-fifo':
(priority-fifo) begin
(priority-fifo) 16 threads will iterate 16 times in the same order each time.
(priority-fifo) If the order varies then there is a bug.
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) end
Execution of 'priority-fifo' complete.
```



## 4. 回答问题

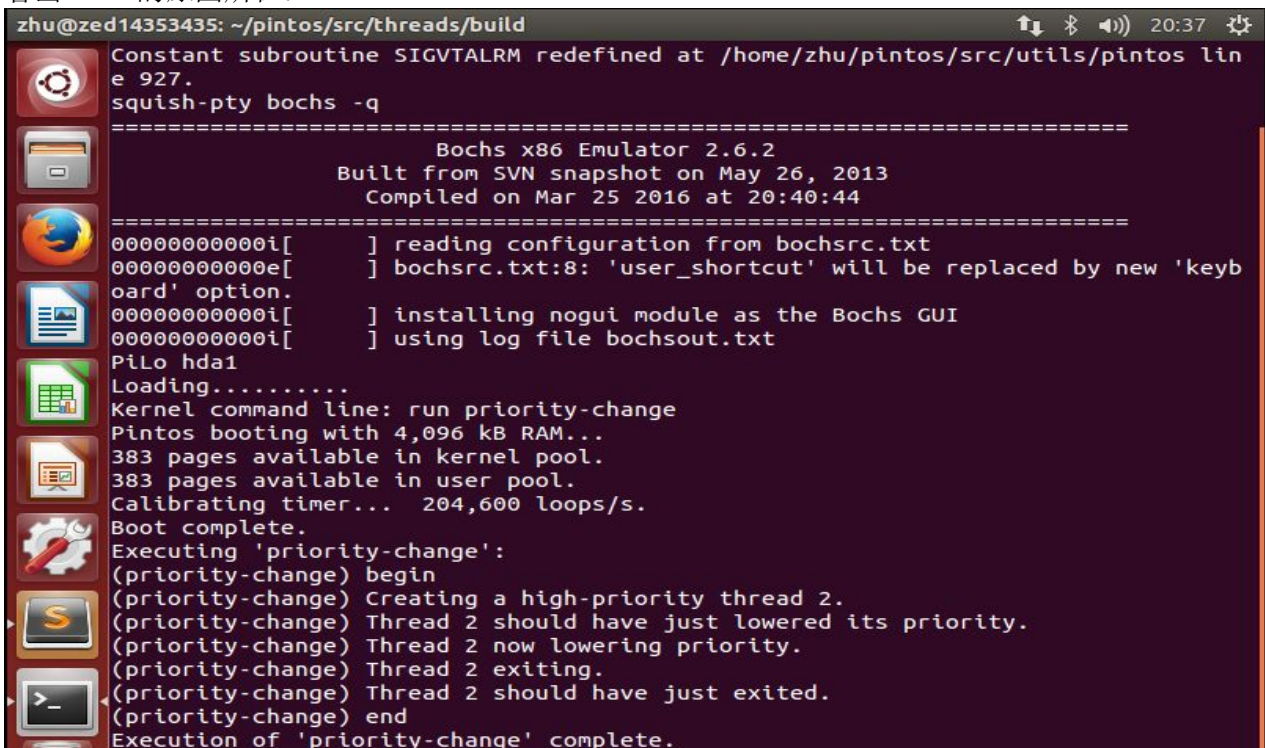
(1)

不考虑 `thread_create()` 的情况，是无法通过测试的，其结果如下图：



```
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
pass 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
18 of 27 tests failed.
make: *** [check] Error 1
zhu@zed14353435:~/pintos/src/threads/build$
```

可以看到 FAIL 了 `priority-change` 和 `priority-preempt`，我们不妨再单独运行这两个测试来看出 FAIL 的原因所在：



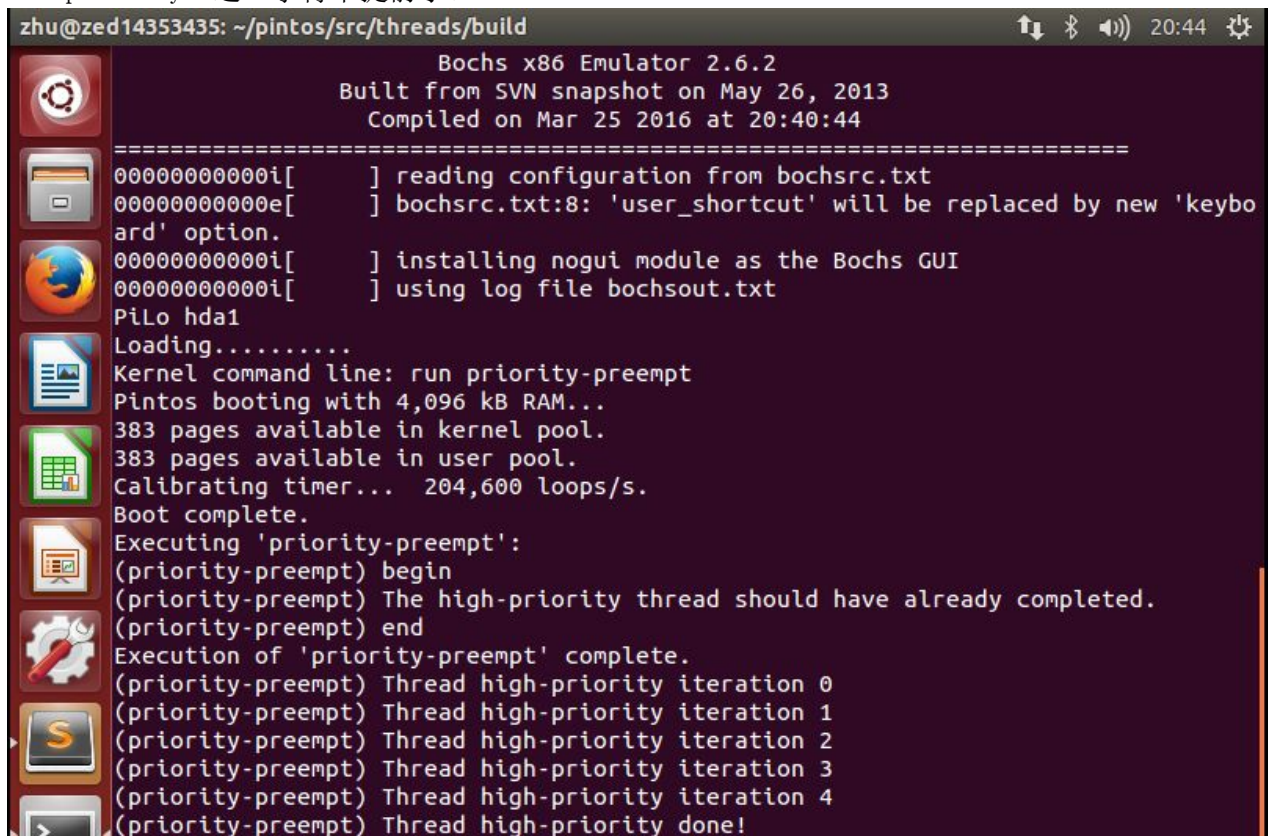
```
zhu@zed14353435: ~/pintos/src/threads/build
Constant subroutine SIGVTALRM redefined at /home/zhu/pintos/src/utils/pintos line 927.
squish-pty bochs -q
=====
Bochs x86 Emulator 2.6.2
Built from SVN snapshot on May 26, 2013
Compiled on Mar 25 2016 at 20:40:44
=====
00000000000i[      ] reading configuration from bochsrc.txt
00000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyboard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: run priority-change
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 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 now lowering priority.
(priority-change) Thread 2 exiting.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.
```



这是 priority-change 的错误运行结果，我们可以看到运行输出字符串的顺序错误了。结合我们之前分析过的 priority-change 的源代码：

```
12 void
13 test_priority_change (void)
14 {
15     /* This test does not work with the MLFQS. */
16     ASSERT (!thread_mlfqs);
17
18     msg ("Creating a high-priority thread 2.");
19     thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL);
20     msg ("Thread 2 should have just lowered its priority.");
21     thread_set_priority (PRI_DEFAULT - 2);
22     msg ("Thread 2 should have just exited.");
23 }
24
25 static void
26 changing_thread (void *aux UNUSED)
27 {
28     msg ("Thread 2 now lowering priority.");
29     thread_set_priority (PRI_DEFAULT - 1);
30     msg ("Thread 2 exiting.");
31 }
```

我们发现错误的原因在于如果不考虑 thread\_create 的情况，那么主线程在 thread\_create 创建完一个子线程之后并不会让出 cpu，而是继续顺序地执行，一直到 thread\_set\_priority 之后才会让出 cpu 给子线程执行，因此我们会发现输出顺序错误，“Thread 2 should have just lowered its priority”这一字符串提前了。



```
zhu@zed14353435: ~/pintos/src/threads/build
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 priority-preempt
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 'priority-preempt':
(priority-preempt) begin
(priority-preempt) The high-priority thread should have already completed.
(priority-preempt) end
Execution of 'priority-preempt' complete.
(priority-preempt) Thread high-priority iteration 0
(priority-preempt) Thread high-priority iteration 1
(priority-preempt) Thread high-priority iteration 2
(priority-preempt) Thread high-priority iteration 3
(priority-preempt) Thread high-priority iteration 4
(priority-preempt) Thread high-priority done!
```

这是 priority-preempt 的错误运行结果，我们可以看到运行输出字符串的顺序相比正确结果错误了。结合我们之前分析过的 priority-change 的源代码：

```

17 void
18 test_priority_preempt (void)
19 {
20     /* This test does not work with the MLFQS. */
21     ASSERT (!thread_mlfqs);
22
23     /* Make sure our priority is the default. */
24     ASSERT (thread_get_priority () == PRI_DEFAULT);
25
26     thread_create ("high-priority", PRI_DEFAULT + 1, simple_thread_func, NULL);
27     msg ("The high-priority thread should have already completed.");
28 }
29
30 static void
31 simple_thread_func (void *aux UNUSED)
32 {
33     int i;
34
35     for (i = 0; i < 5; i++)
36     {
37         msg ("Thread %s iteration %d", thread_name (), i);
38         thread_yield ();
39     }
40     msg ("Thread %s done!", thread_name ());
41 }

```

我们发现错误原因在于如果不考虑 `thread_create` 的情况，则在主线程创建完子线程之后并不会让出 cpu，将一直执行直到主线程结束。因此子线程只有在主线程执行结束之后才可以开始执行，并输出迭代的信息。因此我们在输出结果会看到的输出序列其实是主线程先执行完毕后，再执行子线程而输出的字符串序列。

(2)

在我的理解中，信号量是用来解决互斥问题而引入的一个概念。所谓的互斥问题常常出现于原子操作中，即在多线程系统下，对于某些代码段，我们需要保证每次只可以有有限个线程在执行该代码段，否则就会造成错误（比如：生产者与消费者问题）。

我觉得信号量的概念可以类比现实生活中的车位问题来理解，比如说在一个停车场中一共只有 2 个空位，但是同时有多辆车需要停车，那么每次就只能每次由两辆车进行停车，剩余的车辆需要进行等待，直到有车离开，让出车位之后，才可以再有车进入。在这个例子中，车位数就相当于信号量，每有一个进程尝试执行代码，则信号量需要-1，而一段代码可以被执行的前提条件是信号量必须大于 0，否则就需要进行等待。

因此 P 操作与 V 操作就是为了实现上述的功能而实现的。P 操作先对信号量进行-1，再判断如果判断信号量小于 0 则必须让该进程等待，知道信号量大于等于 0 才可以执行。而 V 操作则是为了释放信号量，对信号量进行+1。然而，直接按照这样的朴素方法实现的话，会有多个进程在忙等待，因而降低了 cpu 的效率。这时，我们可以借鉴上次实验的休眠与唤醒的思想，将 P 操作中的忙等待改为使线程休眠，再在 V 操作中判断如果信号量等于 0 的话，再从休眠队列中唤醒一个线程。这样我们就可以不必进行忙等待的操作了，可以有效地提高效率。

## 5. 实验感想

本次实验是关于线程的优先级调度。

本次实验中遇到的一个问题是在一开始编写代码的时候，我觉得在进行 `thread_yield` 之前可以不加以判断，也即每次在 `set_priority()` 和 `thread_create()` 中都进行一次 `thread_yield()` 操作。我的想法是因为我们上一次实验已经实现了保证就绪队列中始终维护着按照优先级排序，因此我们



进行一次 `thread_yield()` 之后，仍然是由当前最大优先级的线程抢占到 `cpu`。后来我进行了测试，发现也是一样可以通过测试的。

但是后来我仔细思考了一下，发现这种实现方式还是存在 `bug` 的。这种错误出现在有多个相同优先级的情况。因为，按照我们的规定，如果当前正在执行的线程的优先级与就绪队列中的优先级相同的情况下，该线程是不应该让出 `cpu` 的，应该优先级抢占仅仅发生在就绪队列中优先级更大的情况。但是，按照此前的那种实现方式，就会发生错误，这一点与 `thread_yield` 的实现方式是有关的：

```
311 void
312 thread_yield (void)
313 {
314     struct thread *cur = thread_current ();
315     enum intr_level old_level;
316
317     ASSERT (!intr_context ());
318
319     old_level = intr_disable ();
320     if (cur != idle_thread)
321         list_insert_ordered(&ready_list, &cur->elem, cmp_elem, NULL);
322     cur->status = THREAD_READY;
323     schedule ();
324     intr_set_level (old_level);
325 }
```

我们可以看到，`thread_yield` 的实现方式是先把当前正在运行的线程插入到有序队列当中，再进行调度使得一个新的线程使用 `cpu`。因此考虑刚才说到的那种情况，假如我们将当前正在运行的线程的优先级调整为与当前就绪队列中的最大优先级一样，如果不加以判断，将直接进行一次 `thread_yield` 操作，这将使得当前正在运行的线程被插入到最大优先级元素的后面，而将 `cpu` 让给原就绪队列中最大优先级的线程执行，这显然是不符合我们的规定的，因此这样的实现方式不可取，尽管可以通过此次测试，但是只是因为测试样例中没有检测出这种 `bug`。此外这种实现方式还有一个弊端，就是使得 `cpu` 的重新调度次数增多，降低了 `cpu` 的运行效率。