中山大学移动信息工程学院本科生实验报告

（2017 学年春季学期）

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	1501	专业（方向）	移动信息工程
学号	15352006	姓名	蔡丽芝
电话	13538489980	Email	314749816@qq.com
开始日期	2017/4/20	完成日期	2017/4.27

1. 实验目的

（1）分析并解释 3 个 tests，priority-preempt，priority-change，priority-fifo

（2）实现优先级抢占调度

2. 实验过程

(一) Test 分析

Test 1: priority-preempt

测试目的：测试当创建一个优先级高于当前运行线程的线程时候，是否会优先调度高优先级的线程实现优先级抢占调度

过程分析：

```
void
test_priority_preempt (void)
{
  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

  /* Make sure our priority is the default. */
  ASSERT (thread_get_priority () == PRI_DEFAULT);

  thread_create ("high-priority", PRI_DEFAULT + 1, simple_thread_func, NULL);
  msg ("The high-priority thread should have already completed.");
}
```

测试线程创建了一个叫做”high-priority”的线程，并且赋给它一个优先级，使得其优先级高于测试线程。假设当代码改变后，能实现优先抢占调度，则在这个测试中，当创建了 high-priority 线程时，测试线程会从 CPU 中退出重新回到就绪队列中，high-priority 线程会抢占运行，执行 simple_thread_func 函数

```
static void
simple_thread_func (void *aux UNUSED)
{
  int i;

  for (i = 0; i < 5; i++)
    {
      msg ("Thread %s iteration %d", thread_name (), i);
      thread_yield ();
    }
  msg ("Thread %s done!", thread_name ());
}
```

simple_thread_func 会进入一个循环，第一次循环，打印出 thread high-priority iteration 0

后调用 thread_yield 函数，high-priority 从 CPU 中退出，重新回到到就绪队列中，但由于当前线程的优先级高于测试线程，所以从就绪队列中调度的还是 high-priority，high-priority 从上次中断的地方开始运行，第二次循环打印信息，high-priority 又从 cpu 中退出回到就绪队列中，在就绪队列中调度优先级高的 high-priority，直到循环结束，打印 Thread high-priority done，high-priority 线程执行结束，测试线程获得 cpu 时间，从上次中断的地方开始运行，打印信息 The high-priority thread should have already completed.

Test 2: priority-change

测试目的：检测当设置一个线程优先级，使其优先级发生改变的时候，是否会进行优先级抢占调度

过程分析：

```
void
test_priority_change (void)
{
  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

  msg ("Creating a high-priority thread 2.");
  thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL);
  msg ("Thread 2 should have just lowered its priority.");
  thread_set_priority (PRI_DEFAULT - 2);
  msg ("Thread 2 should have just exited.");
}
```

假设将测试线程命名为 thread1(便于分析)，thread1 打印了一串信息，thread1 创建了一个优先级为 PRI_DEFALUT+1 的 thread2，thread2 的优先级高于 thread1，thread1 会从 CPU 中退出回到就绪队列中，thread2 抢占 cpu 调度，执行 changing_thread 函数。

```
static void
changing_thread (void *aux UNUSED)
{
  msg ("Thread 2 now lowering priority.");
  thread_set_priority (PRI_DEFAULT - 1);
  msg ("Thread 2 exiting.");
}
```

changing_thread 函数中调用了 thread_set_priority 函数，将自身的优先级降低为 PRI_DEFAULT-1 由于 thread2 的优先级小于 thread1，所以 thread2 退出 cpu 回到就绪队列中，thread1 在上次中断的地方开始运行，执行 msg（‘Thread 2 should have just lowered its priority’），随后又再次调用 thread_set_priority 将自身的优先级降低为 PRI_DEFAULT-2，此时 thread1 的优先级小于 thread2，thread1 被退出 cpu 回到就绪队列，thread2 抢占调度，在上次中断的地方开始执行，执行 msg（“Thread 2 exiting”）；thread2 执行结束后，threa1 获得 cpu 时间，在上次中断的地方开始执行，执行 msg（“Thread 2 shouldhave just exied”

Test 3: priority-fifo

测试目的：测试线程是否按照 first in first out 的顺序执行。

过程分析：

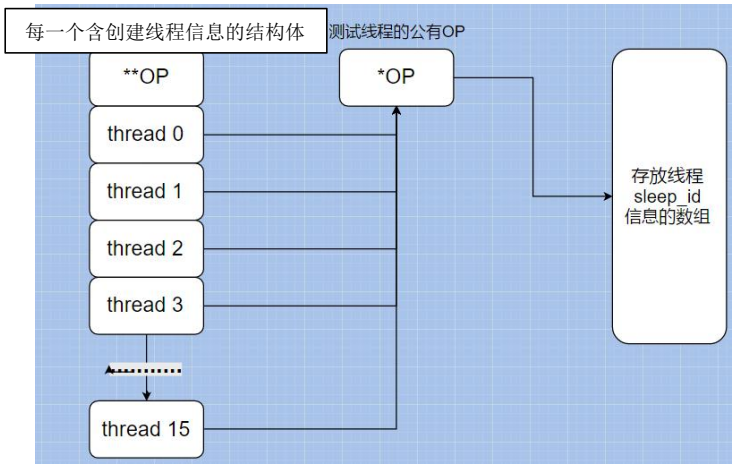
```
struct simple_thread_data
  {
    int id;
    int iterations;
    struct lock *lock;
    int **op;
  };
```

```
thread_set_priority (PRI_DEFAULT +
for (i = 0; i < THREAD_CNT; i++)
  {
    char name[16];
    struct simple_thread_data *d =
    snprintf (name, sizeof name, "
    d->id = i;
    d->iterations = 0;
    d->lock = &lock;
    d->op = &op;
```

```
simple_thread_func (void *data_)
{
  struct simple_thread_data *data = data_
  int i;

  for (i = 0; i < ITER_CNT; i++)
    {
      lock_acquire (data->lock);
      *(*data->op)++ = data->id;
      lock_release (data->lock);
      thread_yield ();
    }
}
```

这 3 段代码中都有指针 op



每个新创建的线程中的 op 都存放着当前测试线程的 op 地址，而测试线程公有的 op 地址会指向一个数组，该数组用来存放线程的 sleep_id 信息。

```
thread_set_priority (PRI_DEFAULT + 2);
for (i = 0; i < THREAD_CNT; i++)
  {
    char name[16];
    struct simple_thread_data *d = data + i;
    snprintf (name, sizeof name, "%d", i);
    d->id = i;
    d->iterations = 0;
    d->lock = &lock;
    d->op = &op;
    thread_create (name, PRI_DEFAULT + 1, simple_thread_func, d);
  }
```

在第一行代码中，调用了 thread_set_priority 函数，将当前测试线程的优先级设为 33，接下来创建了 16 个优先级都为 32 的线程，这里将创建的线程依次命名为 thread0，thread2....,方便下面分析。综合分析这段代码可知，一开始将测试线程的优先级设为 33 的目的是：使测试线程的优先级高于要创建线程的优先级，从而保证能成功创建出 16 个优先级为 32 的线程，并将他们依次放入就绪队列中而不被打断。

```
thread_set_priority (PRI_DEFAULT);
```

再次调用了 thread_set_priority 函数，将测试线程的优先级设为 31，此时测试线程的优先级小于线程 thread0,...thread15，所以测试线程会退出 cpu，thread 0 抢占 cpu，执行 simple_thread_func 函数

```
static void
simple_thread_func (void *data_)
{
  struct simple_thread_data *data = data_;
  int i;

  for (i = 0; i < ITER_CNT; i++)
    {
      lock_acquire (data->lock);
      *(*data->op)++ = data->id;
      lock_release (data->lock);
      thread_yield ();
    }
}
```

第一次进入循环，由于需要将线程的 id 信息赋值给 op 指向的数组，使用锁机制并记录其线程 id，从而确保每个内存位置只有一个线程 id 记录，且使操作不被打断，最后释放锁，避免死锁。接下来调用 thread_yield() 函数，thread 0 退出 cpu，调度就绪队列中的 thread 1 执行 simple_thread_fun 函数，重复上述操作。

在此过程中就绪队列的变化为，设测试线程为 a，thread0 等简称为 0，1，2

0 1 2 3 4 5 .... 15 a -> 0 从就绪队列中调出,执行第一次迭代，后回到就绪队列中。

1 2 3 4 5.....15 0 a -> 1 从就绪队列中调出,执行第一次迭代，后回到就绪队列中

2 3 4 5 6... 15 0 1 a -> 2 从就绪队列中调出,执行第一次迭代，后回到就绪队列中.
. . . . . .
15 0 1 2 3 4 5...14 a -> 15 从就绪队列中调出,执行第一次迭代，后回到就绪队列中.

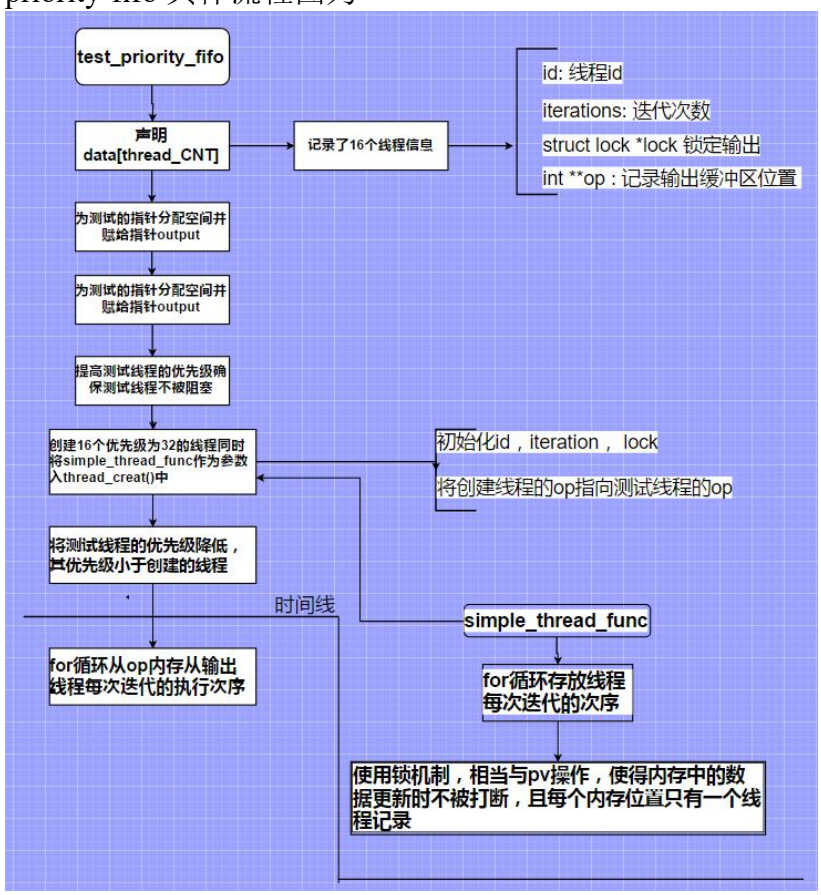
0 1 2 3 4 5.....15 a -> 0 从就绪队列中调出,从上次中断地方执行第二次迭代,后回到就绪队列
1 2 3 4 5 6 ...15 0 a -> 1 从就绪队列中调出,从上次中断地方执行第二次迭代,后回到就绪队列
2 3 4 5 6 7..15 0 1 a -> 2 从就绪队列中调出,从上次中断地方执行第二次迭代,后回就绪队列
. . . . . .
0 1 2 3 4 5 6...15 a-> 0 从就绪队列中调出,从上次中断地方执行第 16 次迭代,后回到就绪队列
1 2 3 4 5 6 7...15 a-> 1 从就绪队列中调出,从上次中断地方执行第 16 次迭代,后回到就绪队列
. . . .
直到所有的 16 个 thread 线程被迭代 16 次后，就绪队列里就只剩下测试线程，测试线程从上次中断的地方开始执行。

```
  cnt = 0;
  for (; output < op; output++)
    {
      struct simple_thread_data *d;

      ASSERT (*output >= 0 && *output < THREAD_CNT);
      d = data + *output;
      if (cnt % THREAD_CNT == 0)
        printf ("(priority-fifo) iteration:");
      printf (" %d", d->id);
      if (++cnt % THREAD_CNT == 0)
        printf ("\n");
      d->iterations++;
    }
output = op = malloc (sizeof *output * THREAD_CNT * ITER_CNT * 2);
```

打印出数组中存放的线程执行的 sleep_id, 当 cnt %THRED_CNT = 0, 的时候表明，所有 16 个线程的第 n 次迭代都完成，此时输出一个换行，进行第 n+1 次迭代的输出。

priority-fifo 具体流程图为



(2) 实验思路与代码分析

思路(参考 ppt):
解决优先级抢占调度的关键点就在于线程的优先级， 在任何地方，一旦发现了更高优先级的线程，就应该发生抢占调度， 抢占这个行为可以具体细分为：将当前正在运行的线程重新放回 ready 队列中，让更高优先级的线程进入 cpu 运行。这个过程其实就是 thread_yield()。

分析原 thread_set_priority 函数：

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
}
```

函数首先利用了 thread_current()函数返回了当前正在 cpu 中运行的线程的指针,然后将新的优先级赋值给了当前正在运行的线程

根据第一个 test: priority-preempty,可知，我们需要在设置函数时，即 thread_set_priority 函数中，检测新设置的优先级是否大于就绪队列中最高优先级的线程的优先级，如果小于，则当前的线程让出 cpu
代码：

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
  thread_yield ();
}
```

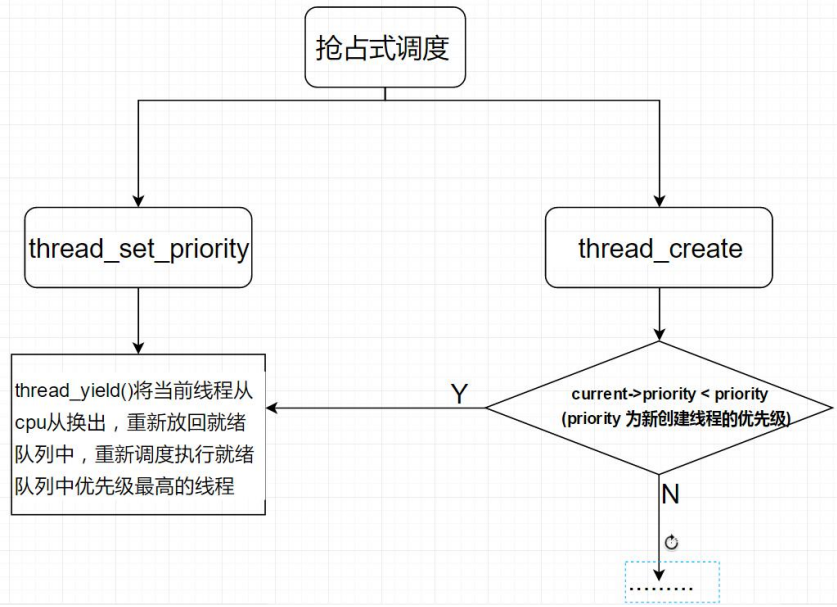
线程新设置了优先级，此时应该马上进行判断改变后的线程优先级是否小于就绪队列中最高优先级的线程，调用 thread_yield 函数，会将当前在 CPU 中运行的线程换出，重新将当前线程放回就绪队列中，由于插入到就绪队列中的线程已经按优先级大小排好队，因此，CPU 再次调度的永远是再就绪队列中优先级最高的线程，若仍然是更改后线程的优先级高的话，自然 cpu 调度运行的还是当前的线程,然而如果更改后的线程优先级变小了,则 cpu 调度运行是就绪队列中优先级最高的那个。

根据第二个 test: priority-change，线程在创建的时候发生了优先级的变化，此时也是要检测新创建的线程优先级是否大于当前的线程，若是，则当前线程必须退出 CPU，执行新创建的线程。
代码：在 thread_creat 函数中

```
// compare with the priority of the current thread
if(thread_current()->priority < priority)
{
   thread_yield();
}
```

通过 thread_current()函数返回了当前在 cpu 中运行的线程的指针，得到当前运行线程的优先级，并与创建线程的优先级进行比较，如果创建线程的优先级大于当前运行线程的优先级，则调用 thread_yield()函数，将当前在 cpu 中运行的线程抢占出来，放到就绪队列中，又因为在上次实验中，我们对 thread_yield 函数作出了修改，使得插入到就绪队列中的序列优先级是有序的，thread_yield()函数会调用 schedule()函数,把就绪队列头部的线程(该线程的优先级最高)调度到 cpu 中进行运行，从而起到了抢占式调度。

代码思路流程图：



3. 实验结果

```
simmon@15352006lizhicai: ~/pintos/src/threads/build
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.
make: *** [check] Error 1
simmon@15352006lizhicai:~/pintos/src/threads/build$
```

priority-preempt 测试结果

```
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-change 测试结果

```
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-fifo 测试结果

```
(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) 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. 如果没有考虑修改 thread_create 函数的情况，test 能通过吗？如果不能，会出现什么结果（请截图），解释为什么会出现这个结果。

答：test 不能通过，并且结果是 18 of 27 tests failed

```
simmon@15352006lizhicai: ~/pintos/src/threads/build
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
simmon@15352006lizhicai:~/pintos/src/threads/build$
```

当忽略了线程创建时出现的优先级变化，priority-change 和 priority-preempt 两个测试都 failed，只有 priority-fifo 测试 pass 了。

分析 priority-change 的测试结果：

```
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.
```

很明显出现上述的打印信息，与 result 里的不符，因此是 failed。

原因是：由于忽略了线程创建时的抢占式调度，运行 test_priority_change 时，当创建一个具有更高优先级的新线程的时候，当前测试线程并不会从 cpu 中退出，而是继续在 cpu 中运行所以会执行 msg("Thread 2 should have just lowered its priority")这行代码。而运行到 thread_set_priority 函数时，降低了自身的优先级，在 thread_set_priority 函数中我们是考虑到了优先级抢占，因此当前线程会被阻塞，线程切换到 thread2 运行，因此会又有上面的输出信息顺序。

分析 priority-preempt 的测试结果:

```
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!
```

对照 pass 时 priority-preempt 的测试结果，打印信息顺序明显不同，因此是 failed
原因是： 在 test_priority_preempt 函数中，创建了一个优先级更高的线程，按理来说测试线程应该从 cpu 中退出来，线程切换到新创建的线程，但是由于我们忽略了线程创建时的抢占式调度，导致当前线程继续运行直到结束，打印 The high-priority thread should have already completed. 而新创建的线程并不会在当前的测试线程中运行，出现上面的结果。
分析 priority-fifo 的测试结果:
priority-fifo 的结果时 pass，因为在 test_priority_fifo 中并没有在创建新线程时出现创建的新线程优先级高与当前运行线程的情况，因此忽略了创建线程时的抢占并不会影响结果。

2. 用自己的话阐述 Pintos 中的 semaphore 和 lock 的区别和联系
答：联系：在 lock 结构体内有一个成员变量 semaphore，因此，lock 可以说是一个特殊的信号量，这个信号量的初始值为 1
区别：a 较于 semaphore 的不同，lock 的 semaphore 的初始值为 1，表示只能有一个线程可进入临界区，一个 lock 只能被一个线程拥有，而 semaphore 的 value 可以不只是 1，可以大于 1，即在临界区内同时可进入多个线程。
b 一个信号量没有一个固定的拥有者，也就是说，pv 操作可以不是同一个线程，一个线程执行 p 操作，可以另外一个线程执行 v 操作，而 lock 不同，一个拥有 lock 的线程，必须在同一个当前线程里进行加锁，解锁，也就是 pv 操作，因此在 lock 结构体中有一个线程变量指针 holder，用于存放锁的拥有者，在 lock_release 的时候里有一个断言 lock 的拥有者必须是当前的线程，否则就会报错。

3. 考虑优先级抢占调度后，重新分析 alarm-priority 测试

```
wake_time = timer_ticks () + 5 * TIMER_FREQ;
sema_init (&wait_sema, 0);
```

设置唤醒时间，设置一个信号量 wait_sema 并把它初始化为 0

```
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);
  }

thread_set_priority (PRI_MIN);
```

调用了 10 次 thread_creat 函数创建了 10 个优先级分别为 25，24，23，22，21，20，30，29，28，27，26 的线程，这些线程会按照优先级排好的顺序放入到就绪队列中，循环结束后测试线程调用了 thread_set_priority 函数把当前自身的线程优先级降到最低，由于测试线程的优先级最低小于前面所创建的 10 个线程，所以测试线程会从 cpu 中退出回到就绪队列。然后调度就绪队列中优先级为 30 的线程执行 alarm_priority_thread 函数。

```
static void
alarm_priority_thread (void *aux UNUSED)
{
  /* Busy-wait until the current time changes. */
  int64_t start_time = timer_ticks ();
  while (timer_elapsed (start_time) == 0)
    continue;

  /* Now we know we're at the very beginning of a timer tick, so
     we can call timer_sleep() without worrying about races
     between checking the time and a timer interrupt. */
  timer_sleep (wake_time - timer_ticks ());

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

  sema_up (&wait_sema);
}
```

在没有实现抢占调度之前，while 循环通过使用忙等待的方式确保时间已经发生变化，目的是为了保证当前操作 P 操作已经完成，这个时候线程进行 timer_sleep()函数运行的时候是不会被系统中断这里涉及到 P 操作，谈谈在没有实现抢占调度前，测试线程是如何能够被阻塞然后先去调度运行之前新创建的线程的，就是通过 P 操作(因为在上一次的实验报告中这部分我没有解释好，所以这次会详细解释没有抢占时 PV 实现的作用)

```
thread_set_priority (PRI_MIN);

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

将测试线程的优先级降到最低，保证最后才被唤醒。进入循环中会执行 10 次 p 操作，第一次执行 P 操作，wait_sema->value = 0; 在 sema_down 函数中

```
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);
```

所以测试线程会被阻塞，放入到等待队列中，新创建的线程获得 cpu，会执行，也就回到 alarm_priority_thread 的分析中，首先执行的优先级为 30 线程进入休眠，调用就绪队列中下一个优先级最高的线程 29，依次到 10 个线程都都休眠结束，到了唤醒时间，由于每个线程的唤醒时间都是一样的，但 cpu 只能给一个线程，因此按照就绪队列里的顺序，依次执行线程，每个线程的末尾都会执行进行 v 即 sema_up（）操作，

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

将放在等待队列中的测试线程重新放回就绪队列中，并把 wait_sema->value++，所有被创建的线程完成后，value = 10，且测试线程在就绪队列中，此时测试线程被调度，回到测试线程中阻塞的位置继续执行，由于 value ！=0 ，所以测试线程不会被阻塞，一直进行 p 操作，value 进行减减操作直到推出循环后，value = 0;好了这是我上次实验没分析好的地方，花了很大篇章，但是我觉得对于这次抢占调度的 alarm-priority 分析有很大的帮助，所以我重分析了一下。

实现了抢占式调度后 alarm-priority 的分析：

在分析没实现抢占式调度时 alarm-prioriry 里讲到了，测试线程在创建完 10 个优先级大小不等的线程后，调用了 thread_set_priority 函数将自身的优先级降为最低，通过上面的分析，可知道之前没实现抢占式调度时，测试线程时是通过调用 sema_down 函数阻塞当前的测试线程去调度之前创建的线程，而实现了抢占式调度后，在测试线程将自身优先级降低的时候，测试线

程就从 cpu 中退出回到就绪队列中 (之前调用 sema_down 函数的时候，测试线程被阻塞，是回到等待队列中，不是回到就绪队列中) 去调度之前创建的线程，执行 alarm_priority_thread 函数的分析和上面的一样，其中由于执行了 10 次 v 操作，所以 wait_sema->value = 10, 10 个线程都执行完后，测试线程获得 cup 时间在上次中断的地方开始重新执行，进入一个 for 循环

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

wait_sema->value--, 当执行到第 10 次后，退出循环，此时 wait_sema->value = 0;
至此，所有线程执行完毕。

5. 实验感想

这次实验在分析 test 的时候，尤其是分析 priority-fifo 的时候，发现了前两次实验很多理解上的误区，原来我一直理解错 thread_create 函数，之前我以为，在创建了线程将线程放入就绪队列中后，新创建的线程就会被立即调用，导致我很多代码都无法理解其中的含义，在分析 priority-fifo 时，终于意识到了自己之前的错误，原来新创建的线程放到就绪队列中时，还需要调度。只有优先级高的才能先获得 CPU 时间从而运行。还有一个误区，原来我之前一直弄混了 thread_yield()函数和 thread_block 函数，调用了 thread_yield 函数的线程，会从 cpu 中退出回到就绪队列，而调用了 thread_block 函数会把线程阻塞，线程不是回到就绪队列中，而是去到等待队列中，thread_unblock 则是把线程从等待队列中放回就绪队列。感觉之前很多代码理解都不彻底，也又很多误区，我觉得是因为在分析 tests 的时候我没有很认真地分析每一个过程，总是逃避一些困难的地方，比如一些信号量等，存在一种大致读懂就可以的侥幸心理，这一次受打击了，所以很仔细地分析，同时在操作系统上主页放上的一些同学写的优先报告也帮助了我很多，看了他们的报告发现自己的报告和理解真的存在很多不足，总之以后还是好好做实验，写报告吧