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	1506	专业（方向）	移动信息工程
学号	15352116	姓名	洪子淇
电话	13726205766	Email	1102229410@qq.com
开始日期	2017.4.21	完成日期	2017.4.27

1 实验目的

①进一步了解线程调度，切换以及睡眠的机制，进而发现 test 中的缺陷。

②了解 semaphore 与 lock 的区别和联系。

③重新设计 thread_set_priority（）函数，要求：每次修改当前线程的优先级，检测新的优先级是否小于 ready 队列中最高优先级的线程的优先级，如果小于，则当前线程让出 CPU

④每一次创建线程（thread_create（）函数），并加入 ready 队列后，检测这个加入 ready 队列的线程的优先级是否大于当前线程，如果大于，则当前线程退出 CPU

⑤实现抢占式调度并分析四个 test：priority-preempt，priority-change，priority-fifo 和 alarm-priority

2 实验过程

2.1 Test 分析

✧ priority-preempt 测试样例

✧ 测试目的：

这个测试的目的是确保一个高优先级的线程处于优先位置。创建一个线程，新建的线程优先级高于此时运行线程的优先级，如果线程执行切换到新建的线程，测试成功；否则就测试失败。

✧ 过程分析

①首先我们来看 test 中 priority-preempt 的代码：

```
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.");
}
```

创建一个 test 线程，优先级为 PRI_DEFAULT，然后调用 thread_create 函数新创建了一个优先级为 PRI_DEFAULT+1 的测试线程。其中 thread_create 函数又调用了 simple_thread_func 函数。

②接下来我们来看 simple_thread_func 函数：

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

调用 thread_yield 函数，在上一次实验了解 yield 函数先是把当前运行的线程放入就绪队列，再调用 schedule 函数，从就绪队列里拿下一个线程切换过来进行 run。

---修改之前失败原因：当创建了一个较高的优先级线程之后，该线程并没有抢占 test 线程的 CPU，所以当 test 线程执行完之后才会去执行先创建的测试线程。

---修改之后成功原因：在创建线程的时候增加了对当前运行的线程与新创建的线程的优先级的判断，如果优先级高就抢占当前线程的 CPU，把当前线程

放到就绪队列，而运行新的线程。

- 这个 for 循环一开始的时候误解了它的意思，以为会创建 5 个线程，最主要是对线程的创建还不是很了解。其实在这个测试一共就两个线程，一个是 test 线程，一个是在 test 线程新创建的子线程。而在这里进行了 5 次循环的意思是，子线程没有被中断。执行完子线程之后再执行 test 线程。

✧ 结果分析:

新创建的高优先级的线程可以抢占当前线程的 CPU，测试成功。

```
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.

seven@15352116:~/pintos/src/threads/build$ ================================
==================================
```

✧ **priority-change 测试样例**

✧ 测试目的:

这个测试目的是确保当前执行的是优先级最高的线程。在基于 priority-preempt 测试上，若当前执行的线程改变其优先级，并且低于就绪队列里的线程，如果能够正确切换到优先级高的线程，则测试成功；否则测试失败。

✧ 过程分析:

①首先我们来看 test_priority_change 函数:

```
msg ("Creating a high-priority thread 2.");
thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL);
```

先创建了一个 test 线程，然后创建了一个比 test 线程优先级高的 thread 2，由于新创建的 thread 2 优先级高于 test 线程，那么此时线程切换到 thread 2，然后又调用了 changing_thread 函数。

②我们现在来看 changing_thread 函数：

```
msg ("Thread 2 now lowering priority.");
thread_set_priority (PRI_DEFAULT - 1);
```

可以知道该函数调用了 thread_set_priority 函数将 thread 2 的优先级改为 PRI_DEFAULT-1。此时 thread 2 的优先级小于了 test 线程，所以把 CPU 让出给 test 线程。

③回到 test_priority_change 函数：

```
msg ("Thread 2 should have just lowered its priority.");
thread_set_priority (PRI_DEFAULT - 2);
```

当前的 test 线程又执行了 thread_set_priority 函数，优先级改为 PRI_DEFAULT-2，这个时候 thread 2 的优先级又高于 test 线程。

④回到 thread 2 线程的 changing_thread 函数：

```
msg ("Thread 2 exiting.");
```

执行完 msg，thread 2 线程结束。

⑤回到 test 线程的 test_priority_change 函数：

```
msg ("Thread 2 should have just exited.");
```

执行完 msg，test 线程结束。

回顾整个过程（省略其中 msg 的执行），线程切换顺序为：

✧ 结果分析：

测试结果如图，根据分析，msg 的输出按照了线程切换的顺序进行输出，即当线程改变了优先级之后可以保证高优先级的线程在 running 队列里面，测试成功。

```
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.

seven@15352116:~/pintos/src/threads/build$ ==============================
====================================
```

✧ **priority-fifo 测试样例**

✧ 测试目的：

这个测试的目的是：测试线程是否符合 fifo 原则。同时创建多个优先级相同的线程，如果他们是先进先出的，则测试成功，否则测试失败。

✧ 过程分析：

测试样例新建了一个结构体 simple_thread_data，包括了 ID，目前线程迭代执行的次数以及输出锁和输出数组的指针。

①首先我们分析 test_priority_fifo 函数：

```
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);
  }

thread_set_priority (PRI_DEFAULT);
/* All the other threads now run to termination here. */
```

test 线程的优先级设为 PRI_DEFAULT + 2，然后进入循环创建 THREAD_CNT 个优先级为 PRI_DEFAULT +1 的子线程，由于创建的子线程优先级低于当前 test 线程，test 线程继续执行。紧接着把 test 线程优先级设为 PRI_DEFAULT，优先级低于就绪队列队首线程，进行线程切换，运行子线程。

②子线程调用了 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 ();
    }
}
```

函数先将当前运行的线程的 ID 值赋给 op 所指向的数组元素，调用 thread_yield 切换线程，由于 THREAD_CNT 个子线程的优先级都相同，所以 thread_yield 函数将会切换到 ID+1 的线程运行。根据迭代次数，每个线程一共进行了 ITER_CNT 次的线程切换。op 数组中存储的将会是迭代次数为 ITER_CNT 的 0 到 THREAD_CNT-1 的整数序列，子线程结束，运行 test 线程：

```
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++;
  }
```

✧ 结果分析：

根据上述分析，每个线程都迭代了 16 次，而且每次迭代中相同的线程按照 fifo 的方式被切换，被切换的线程插入就绪队列里相同优先级线程的尾部，所以从 0 到 15 的序列总共出现了 16 次，结果如下：

```
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) 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.

seven@15352116:~/pintos/src/threads/build$ ====================================
==================================
```

2.2 实验思路与代码分析

✧ **实验思路**

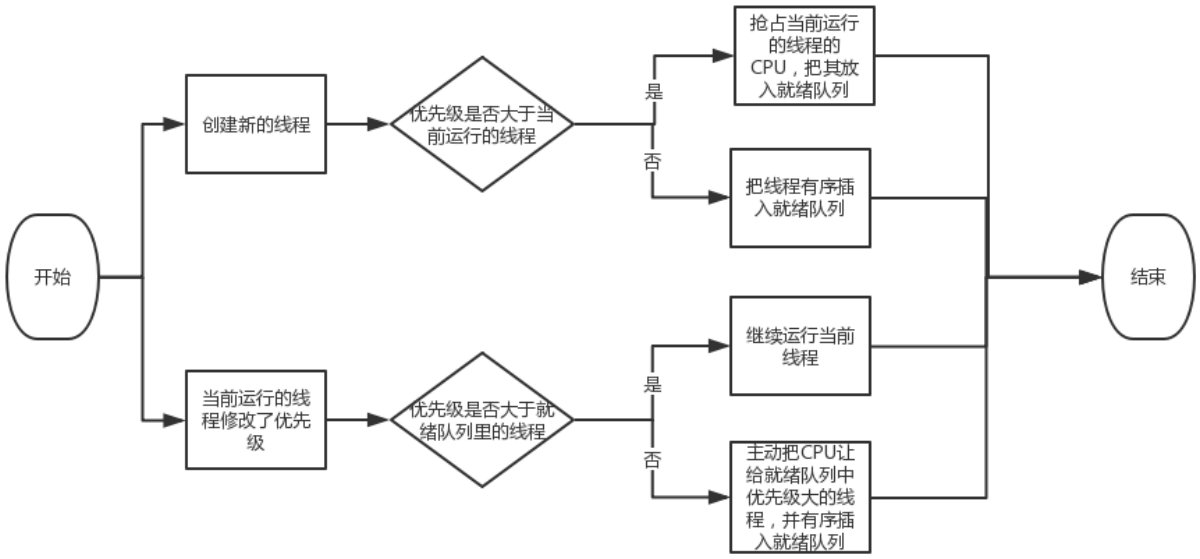
优先级抢占调度的关键问题是线程的优先级，一旦在当前时刻出现了更高优先级的线程，就应该发生抢占。抢占就是当前正在运行地线程重新回到就绪队列，让最高优先级地线程进入 CPU 运行。

具体分以下两种情况：

1. 当创建新线程的时候，判断其优先级与正在运行的线程的优先级，如果优先级高于正在运行的线程，那么就应该抢占当前线程的 CPU，并把当前的线程退回到就绪队列。
2. 当前运行的线程的优先级更改之后比就绪队列里面优先级最大的线程的优先级要低的时候，主动把 CPU 让出去，并且有序地插入到就绪队列里面。

✧ 流程图与伪代码

● 流程图：



- 伪代码

```
创建新线程:

    如果 新线程优先级 > 执行线程优先级

        抢占 CPU;
```

```
改变线程优先级:

    如果 线程优先级低于就绪队列队首优先级

        让出 CPU;
```

✧ 代码分析

在实验思路里面我们大概可以知道需要在 thread_create 函数和 thread_set_priority 函数加上对创建的和修改的线程增加优先级的判断，根据判断是否进行抢占。

①首先我们需要了解 thread_yield 函数：

```
/* Yields the CPU.  The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
```

从注释我们可以看出 yield 函数就是一个 CPU，表示当前的线程没有休眠也可能不会被马上切换掉。

该函数第一行：`struct thread *cur = thread_current ();`

②让我们跳转 thread_current 函数：

```
/* Returns the running thread.
   This is running_thread() plus a couple of sanity checks.
   See the big comment at the top of thread.h for details. */
struct thread *
thread_current (void)
{
  struct thread *t = running_thread ();

  /* Make sure T is really a thread.
     If either of these assertions fire, then your thread may
     have overflowed its stack.  Each thread has less than 4 kB
     of stack, so a few big automatic arrays or moderate
     recursion can cause stack overflow. */
  ASSERT (is_thread (t));
  ASSERT (t->status == THREAD_RUNNING);

  return t;
}
```

根据层次，首先是 thread_current 函数调用的 running_thread 函数，返回了正在运行的线程的指针。根据 pintos 的命名含义的去理解，接下来有一个断言 t 指针是一个线程，一个断言这个线程处于 THREAD_RUNNING（运行）状态 。总而言之，current 返回了当前线程的起始指针位置。(起始在这里主要是指线程这一个结构的开始位置的指针，不影响它的物理含义)

③回到 yield 函数

```
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    //list_push_back (&ready_list, &cur->elem);
    list_insert_ordered (&ready_list, &cur->elem, &cmp_priority, NULL);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

我们在上一次实验中了解到绿色框住的程序是为了保证原子性操作，接着判断当前的线程不是空闲线程，如果不是就把当前线程有序插入到就绪队列里面然后把当前的线程的状态改为 ready。（如果是空闲线程就证明当前没有线程在运行，所以也没有必要进行插入。）而 schedule 函数就是将就绪队列中的队首线程切换为运行状态。

总而言之，yield 函数就是把当前运行的线程扔回到就绪的队列。又知道 thread_current 是返回了当前运行的线程的指针。根据宏定义，可以访问当前线程的优先级。

④list_entry 函数，返回就绪队列队首线程。由于实现了有序插入，所以就绪队列队首元素就是优先级最大的元素。

✧ 在 create 函数增加线程切换，要加以判断；在改变优先级的时候进行线程切换，也要加以判断，代码如下：

create 函数：

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

//////////////     which is changed     ////////////
if (thread_current()->priority < priority)
  thread_yield();
```

set_priority 函数：

```
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;

  //////////////     which is changed     ////////////
  struct list_elem * e = list_begin(&ready_list);
  if (new_priority < list_entry(e, struct thread, elem)->priority)
    thread_yield();

}
```

3 实验结果

- 对比之前的实验结果可以发现，多过了 3 个 priority-preempt，priority-change，priority-fifo 的测试样例。

```
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.
make: *** [check] Error 1
seven@15352116:~/pintos/src/threads/build$
```

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：priority_preemp 和 priority_change 不能通过，priority_fifo 可以通过；

```
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
```

①首先分析 preempt 函数，新创键了一个优先级高于当前线程的线程。由于删掉了 thread_create 函数的修改，没有进行抢占式的调度。所以会先执行完 test 线程，再执行子线程。输出即会看到 end 变为了前面：

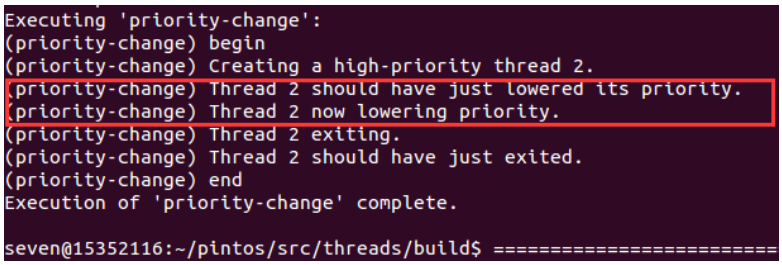
```
Executing 'priority-preempt':
(priority-preempt) begin
(priority-preempt) The high-priority thread should have already completed.
(priority-preempt) end
Execution of 'priority-preempt' complete.
(priority-preempt) Thread high-priority iteration 0
(priority-preempt) Thread high-priority iteration 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!

seven@15352116:~/pintos/src/threads/build$ ================================
```

结果符合猜想。

①priority-change：

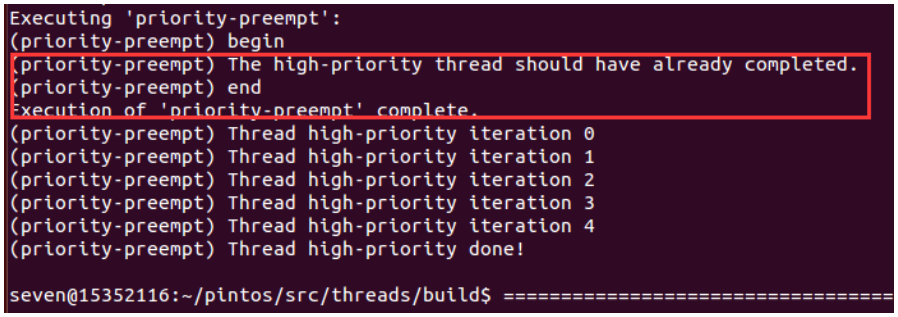
由于新创建的线程优先级要高于 test 线程，所以如果把创建线程时修改的程序删掉，该线程就不能进行抢占式调度，执行的顺序应该是下图：

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

  msg ("Creating a high-priority thread 2.");①
  thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL); ②
  msg ("Thread 2 should have just lowered its priority.");③
  thread_set_priority (PRI_DEFAULT - 2); ④
  msg ("Thread 2 should have just exited."); ⑧
}

static void
changing_thread (void *aux UNUSED)
{
  msg ("Thread 2 now lowering priority.");⑤
  thread_set_priority (PRI_DEFAULT - 1);⑥
  msg ("Thread 2 exiting."); ⑦
}
```



（测试结果符合猜想，和正确输出对比发现，红色框框住的两条 msg 顺序相反，测试失败。）

②priority-preempt：



由图中红色框可以看出新创建的线程并没有抢占了 test 线程的 CPU，当整个 test 线程结束之后才开始执行子线程。

③由于 priority_fifo 在创建新的线程的时候优先级低于正在执行的 test 线程，所以并不需要进行抢占，所以删掉对该测试并没有影响，只要更改线程的时候进行了线程切换，该测试就是成功的。

- 2.用自己的话阐述 Pintos 中的 semaphore 和 lock 的区别和联系。

答：semaphore 是一个非负整数类型的信号量，记录临界区资源的数量，可以被多个线程进行拥有，在 pintos 里面对应信号量的操作有：sema_down 和 sema_up。即通俗的 PV 操作，当线程获得资源时进行 sema_down（P）操作，当线程不再需要资源时，释放资源进行 sema_up（V）操作。如果资源数大于 0，表示可以被访问。如果当资源数为 0 时，线程就会进入等待状态，如果有多个线程进行申请，就会形成一个等待的队列，直到有线程释放了资源，信号

量增加。

而 lock 是一种二进制的信号量（也称互斥锁/互斥量），只有两个值，0 或 1。它是一种单值的信号量，也就是说它只能用于一个资源的互斥访问，而不能实现资源的多线程互斥和同步。在 pintos 对应的信号量操作，分别是 lock_acquire（P）操作和 lock_release（V）操作。Lock 首先会初始化为 1，表示这个资源可以被申请，当有线程申请了这个资源时，那么这个锁就会属于这个线程，并且信号量 1-1= 0。如果有其他线程申请这个资源就会被阻塞掉，直到这个锁被解开，即该线程释放资源，信号量 0+1=1 的时候。而阻塞的机制是 lock_acquire 通过 sema_down (&lock->semaphore)实现的。在线程进行释放资源的时候调用 lock_release 同时调用 sema_up (&lock->semaphore)对锁的对象进行解锁。

综上所述，lock 处理多线程的临界区问题：多个线程共享一个信号量（一种资源），并初始化为 1，并且锁具有唯一对象性，只有获得锁的那个线程才有解锁的功能且在未解锁的时候，对于其他申请该资源的线程进行了阻塞。而 semaphore 用来控制访问具有若干个实例的某种资源，并初始化为该资源的数量。每个线程需要使用资源是进行 sema_down 操作，释放资源时进行 sema_up 操作。当信号量的值为 0 时，所有资源被使用，之后需要使用资源的线程会被阻塞直到信号量计数大于 0。

- 3. 考虑优先级抢占调度后，重新分析 alarm-priority 测试。

  答：首先设定了 test 线程的唤醒时间，然后创建了 10 个线程，每个线程都会去执行 alarm_priority_thread 函数。由于当前 test 线程的优先级为 PRI_DEFAULT，根据子线程优先级的定义式可知，那 10 个线程的优先级

都比 test 线程的优先级要低，所以继续执行 test 线程：

```
thread_set_priority (PRI_MIN);
```

test 线程的优先级设为最低。由于抢占式的调度算法，test 线程立即被阻塞，子线程按照优先级的大小依次执行。

与上一次测试不一样在于，上一次没有实现抢占式的调度，执行完 thread_set_priority 函数之后，CPU 并没有给到子线程，而是继续运行 test 线程，进入 for 循环：

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

由于一开始把信号量设置为 0，执行当前线程操作时把 test 线程阻塞掉，把 CPU 让给子线程，从而实现了线程的切换。所以两次线程切换的机制是不一样的。

5 实验感想

5.1 实验准备阶段

首先我并没有一开始就做抢占式调度，根据我上一次的实验经历发现，首先得去思考抢占式调度的机制到底是什么，即是如何去实现抢占的。当了解到抢占可以分步执行的时候，先判断优先级大小，如果优先级小与就绪队列里的线程，就把当前线程扔回到就绪队列，利用线程切换函数 thread_yield，而我们在第一次实验中又实现了插入有序，那么显然。。。我们就并不需要再去实现其他内容了。

5.2 实验进行阶段

对实验内容有了一个清晰的想法之后就是实现了。就觉得第一步判断优先级

有点多余，因为无论优先级大还是小，我只需要调用 yield 函数，把正在运行的线程扔回就绪队列就好，有序插入已经实现了，按照就绪队列的顺序执行就可以了。

thread_create 和 thread_set_priority 仅加上 thread_yield 函数的实验结果：

```
16 of 27 tests failed.
make: *** [check] Error 1
seven@15352116:~/pintos/src/threads/build$
```

后来再仔细想想的时候，虽然结果和多写几行判断的结果是一样的，可是实现的机制毕竟不一样，少了一行判断就意味着：

①即使创建的线程的优先级**低于**正在 running 的线程，running 序列还是需要被阻塞，那么可能就会多进行一次保存现场和恢复现场的操作，在实验过程中没有深刻体会。可能在实际上当不断创建了新的线程，而且最新创建的线程的优先级都比就绪队列里面的优先级低。那么 CPU 就不断地对 running 线程做了中断，保存现场，恢复现场的工作。而这些都是没有必要的。白白浪费时间，降低 CPU 的效率。所以加多判断是很有必要的，即使对实验结果没有影响。

②当创建的线程的优先级与正在 running 的线程相等的时候，它也不具备抢占能力。可是依旧调用了 yield 函数，把正在运行的线程切换到就绪队列，此时它的位置就会在新创建的线程的后面，而不符合优先级相等的时候考虑先到先服务的策略了。

同理，对 set_priority 中最好也是加多一个判断。

5.3 实验收获阶段

第一次实验说真的，其实对于 test 的分析思路还是没有那么清楚。而这一次，可能是因为比较简单，然后对于线程切换的顺序，什么时候执行那一句话，什

么线程正在执行，什么线程在就绪队列，就绪队列的排序情况又是如何的，有了一个较为清晰的了解。这可能是我觉得最大的收获的地方。