

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 级	专业 (方向)	信息与计算科学
学号	16339038	姓名	舒展

1. 实验目的

2. 实验过程

(一) Test 源文件分析

• priority-preempt

测试目的:

通过输出中间过程信息来测试“高优先级的线程确实发生了抢占”。按照预期,“Thread high-priority iteration”的信息应该连续输出五次,紧接着输出信息“Thread high-priority done !”,最后输出“The high-priority thread should have already completed.”。若不然,则测试失败。

过程分析:

```
static thread_func simple_thread_func;
```

文件中首先定义了一个 simple_thread_func 函数。

simple_thread_func 的定义如下:

```
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 ());
}
```

这个函数做的事情很简单:循环输出五条语句,然后输出一条“Thread high-priority done!”语句。

接下来进入主线程: test_priority_preempt 进行分析:

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

这个断言保证了主线程（即这个测试）的优先级为 PRI_DEFAULT。这样做的目的是为了确保接下来创建的子线程的优先级（PRI_DEFAULT + 1）比主线程要高。

```
thread_create ("high-priority", PRI_DEFAULT + 1, simple_thread_func, NULL);
```

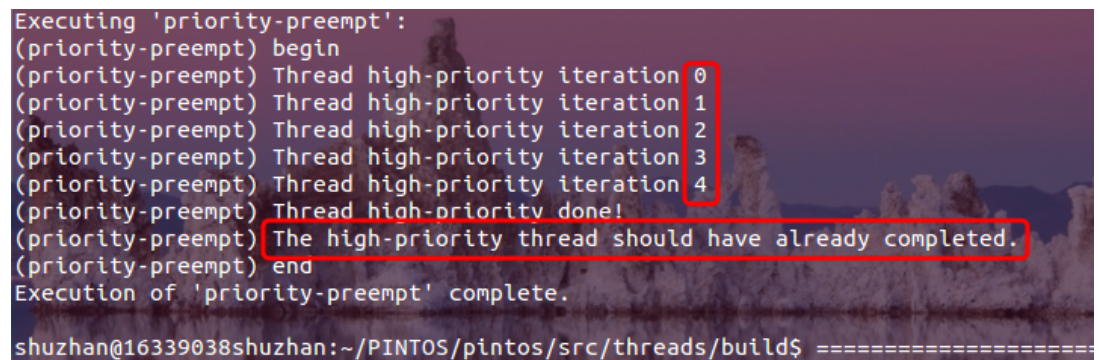
接着在主线程中创建一个子线程，优先级设置 PRI_DEFAULT + 1。由于此时已经实现了优先级抢占，所以该子线程在创建后，马上会抢占主线程的 cpu 资源，执行 simple_thread_func 函数。

```
msg ("The high-priority thread should have already completed.");
```

正如前面所分析的，子线程输出相应的信息、进行完后，主线程才有机会分配到 cpu，继续执行这条语句。

结果分析：

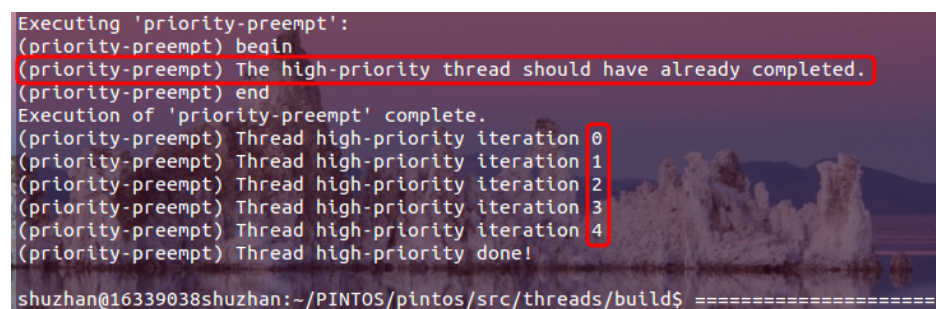
test-priority-preempt 的 output 如下图：



```
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.  
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ =====
```

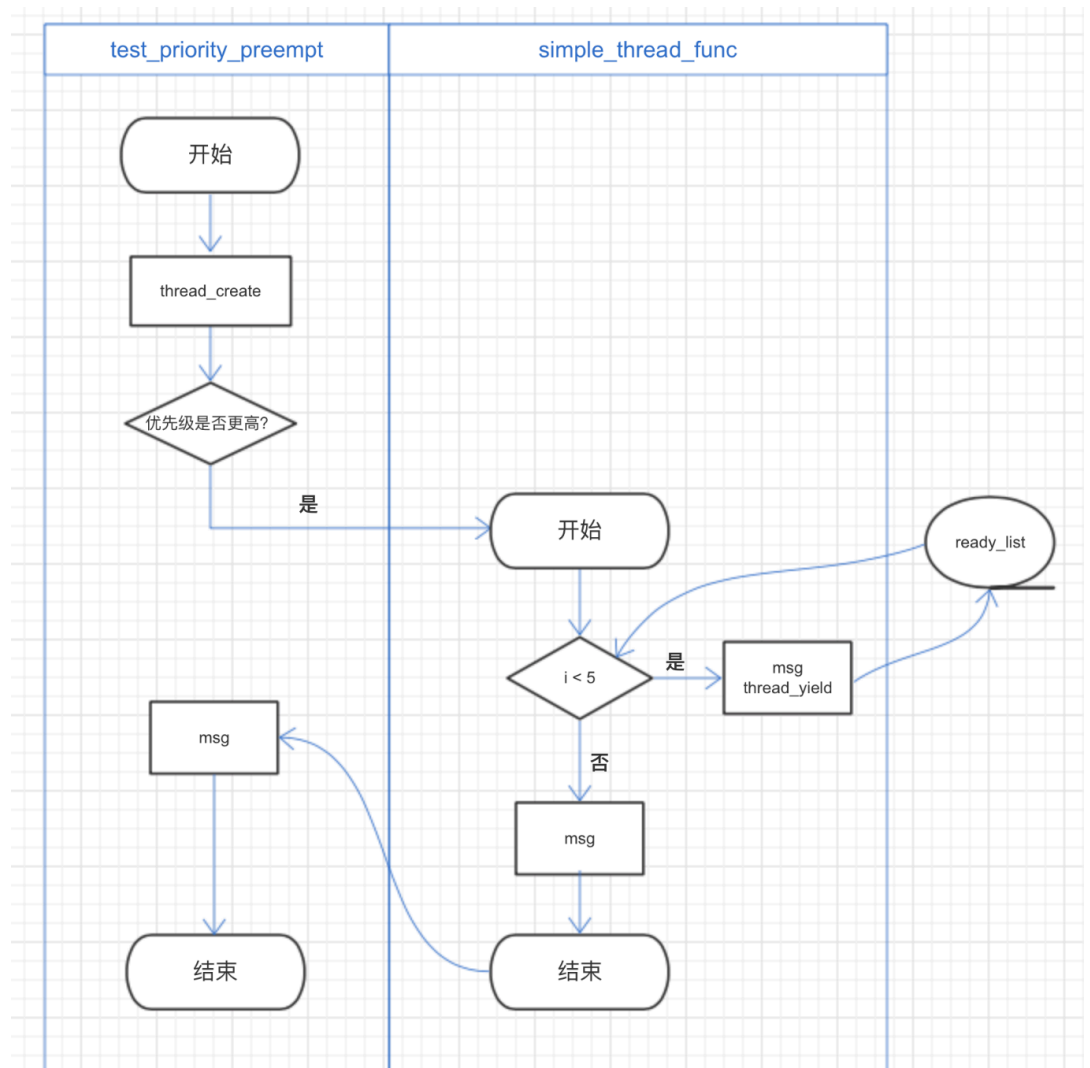
如上图所示，先依次输出五条“Thread high-priority iteration”语句，然后输出一条“Thread high-priority done!”语句，最后输出主线程中的“The high-priority thread should have already completed.”。如预测的一样，说明的确发生了高优先级的抢占。

另外，若没有实现优先级抢占，输出信息的顺序应该是调换过来的，如下图所示：



```
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!  
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ =====
```

流程图：



• priority-change

测试目的：

降低正在 running 的线程的优先级，使其的优先级变得并不是最高，以此来测试的确发生了优先级抢占。如果在测试过程中，输出信息的顺序按照预期的一样则通过测试，否则测试失败。

过程分析：

文件中首先定义了一个 changing_thread 函数，下面是其定义：

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

如上图这个函数先打印出第 1 条信息，然后调用 `thread_set_priority` 方法 `changing_thread` 的优先级降低为 `PRI_DEFAULT - 1`。这时由于优先级降低，该线程会被高优先级的线程抢占。等到再次分配到 `cpu` 时，打印第 2 条信息。

下面分析主线程 `test_priority_change` 测试函数：

```
msg ("Creating a high-priority thread 2.");
```

首先打印该条信息。注：主线程的优先级应该为默认优先级：`PRI_DEFAULT`。

```
thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL);
```

然后调用 `thread_create` 方法创建一个子线程，该线程的优先级设置为 `PRI_DEFAULT + 1`。应该该子线程的优先级比主线程的优先级更高，所以子线程抢占 `cpu` 资源，执行 `changing_thread` 函数。

切换到子线程，打印出第 1 条语句：“Thread 2 now lowering priority.”。将自己的优先级降低至 `PRI_DEFAULT - 1`。此时主线程的优先级又比主线程更低了，于是主线程抢占 `cpu` 资源。

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

主线程接着打印出该条语句。

```
thread_set_priority (PRI_DEFAULT - 2);
```

主线程又将自己的优先级降低为 `PRI_DEFAULT - 2`，低于子线程的优先级。

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

子线程打印出该条信息，然后结束线程。

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

子线程结束，主线程可以被分配到 `cpu` 资源，于是打印该条信息，结束主线程。

结果分析：

```

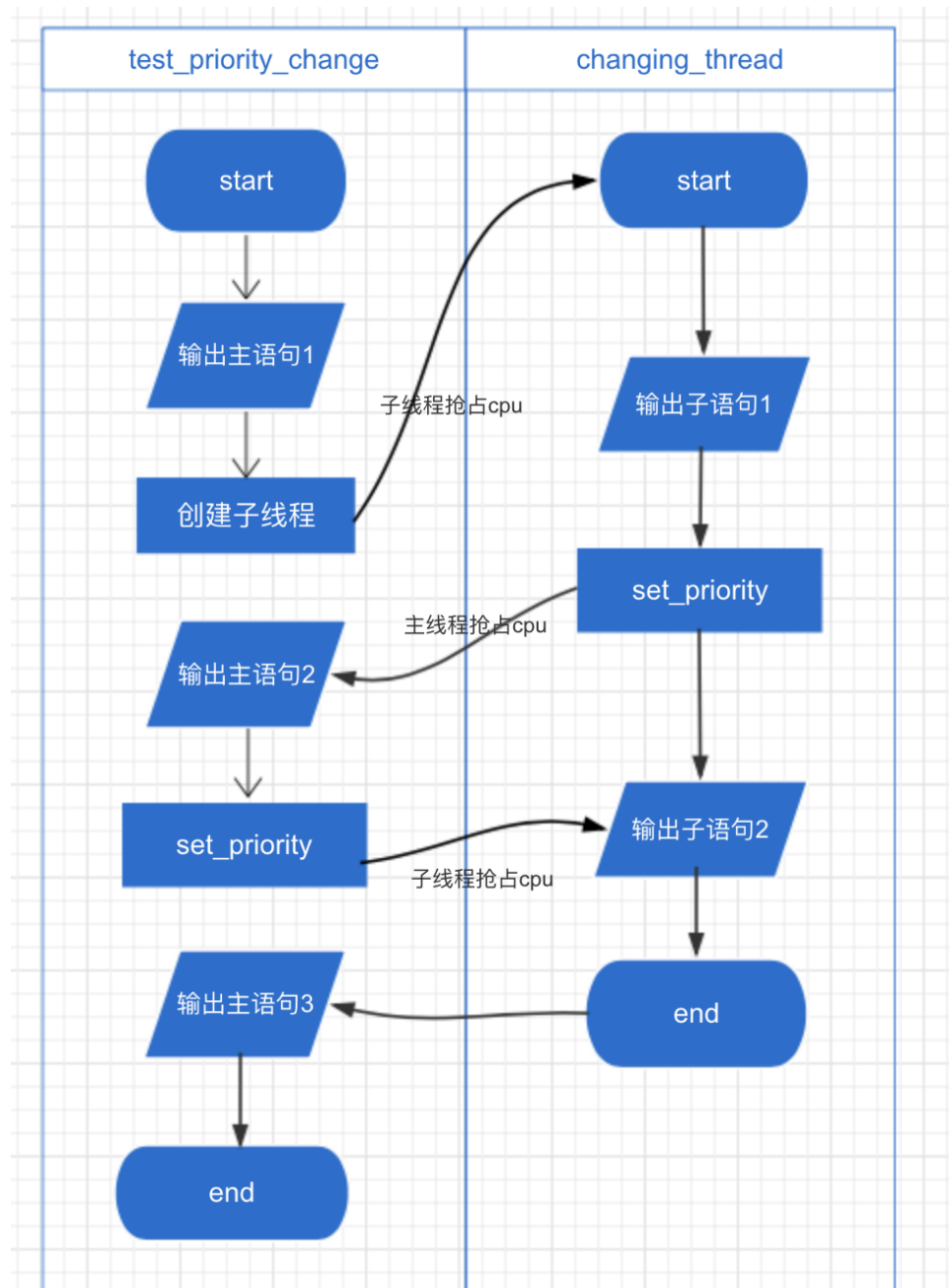
Executing 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 now lowering priority.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 exiting.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.

shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ =====

```

如图，相应信息的确按照期望的顺序打印出来，测试通过。

流程图：



- priority-fifo

测试目的：

按照 0-15 的编号依次创建 16 个线程，循环 16 次。对于每次循环，通过输出信息的方式，确保这 16 个线程是按照先进先出（0-15）的顺序来执行的。若 16 次循环顺序均正确则通过测试；否则测试失败。

过程分析：

```
struct simple_thread_data
{
    int id;                /* Sleeper ID. */
    int iterations;        /* Iterations so far. */
    struct lock *lock;     /* Lock on output. */
    int **op;              /* Output buffer position. */
};
```

创建了一个名为 simple_thread_data 的结构体，含有四个成员变量。
id: 线程的编号；iterations: 该线程目前循环的次数；lock: 这些线程使用的同一个锁；op: 指向输出缓冲区的位置指针。

子线程 simple_thread_func 函数分析：

```
struct simple_thread_data *data = data_;
```

定义一个结构体指针，指向该线程对应的 simple_thread_data 结构体。

接下来为一个 for 循环，循环 ITER_CNT(16) 次：

```
lock_acquire (data->lock);
*(*data->op)++ = data->id;
lock_release (data->lock);
thread_yield ();
```

首先调用 lock_acquire 方法，当前主线程请求获得锁的所有权。

进入 lock_acquire 函数：

```
ASSERT (lock != NULL);
ASSERT (!intr_context ());
ASSERT (!lock_held_by_current_thread (lock));
```

首先是三句断言：lock 不是指向 NULL；此时并不处于中断状态；锁并不被当前正在 running 的线程所拥有。

```
sema_down (&lock->semaphore);
```

调用 sema_down 方法。

下面进入 `sema_down` 函数（虽然上个实验已经分析过该函数，但为了将这个测试过程讲清楚，有必要再分析一遍）：

```
while (sema->value == 0)
{
    list_push_back (&sema->waiters, &thread_current ()->elem);
    thread_block ();
}
```

核心部分为该 `while` 循环。若信号量为 0，说明此时锁正在被占用，于是将当前线程放入 `waiters` 进行排队，并调用 `thread_block` 方法将其阻塞掉。（注意：因为信号量初始值为 1，所以第一个子线程不会进入此循环，后面也会分析到。）

这个线程再次被唤醒时是在其它线程调用 `lock_release` 方法后（后面会分析到），此时信号量 `sema` 的值不再为 0（为 1），退出循环。

```
sema->value--;
```

退出循环后接着执行这条语句，信号量的值减 1，变回 0。

回到 `lock_acquire` 函数：

```
lock->holder = thread_current ();
```

该函数最后执行这条语句，表示此时锁被当前正在运行的线程所拥有。

回到 `simple_thread_func` 函数，此时锁已被该子线程所拥有。

```
lock_acquire (data->lock);
*(*data->op)++ = data->id;
lock_release (data->lock);
thread_yield ();
```

- 该子线程执行这条语句，向输出缓冲区中写入当前子线程的 `id`，并将 `op` 指针向后移动一位。注意：因为此时已经上锁，所以无需担心其它子线程对缓冲区进行写入。

- 执行完上面那条语句后，目的已经达到，调用 `lock_release` 进行解锁。

进入 `lock_release` 函数：

```
ASSERT (lock != NULL);
ASSERT (lock_held_by_current_thread (lock));
```

首先是两个断言：`lock` 指向的不是空指针；锁的确是被当前正在运行的线程锁拥有的。

```
lock->holder = NULL;
sema_up (&lock->semaphore);
```

- 然后释放锁的拥有者。
- 然后调用 sema_up 方法：将 waiters 队列中的第一个线程唤醒（如果队列不为空）；将信号量的值加 1（有 0 变为 1）。

回到 simple_thread_func 子线程：

```
thread_yield ();
```

循环最后调度该函数，将当前子线程切出 cpu，并进行重新调度。于是该子线程进入 ready_list，等待再次分配到 cpu，进行下一次的循环。以上便是 simple_thread_func 的过程分析。

主线程 test_priority_fifo 函数分析：

```
output = op = malloc (sizeof *output * THREAD_CNT * ITER_CNT * 2);
```

调用 malloc 方法为输出缓冲区申请内存。output 储存缓冲区的首地址；op 起到移动光标的作用。

```
lock_init (&lock);
```

接着调用 lock_init 方法初始化锁。

进入 lock_init 函数：

```
void
lock_init (struct lock *lock)
{
    ASSERT (lock != NULL);

    lock->holder = NULL;
    sema_init (&lock->semaphore, 1);
}
```

- 初始化锁，holder 应该指向 NULL
- 然后调用 sema_init 初始化信号量。注意穿进去的参数为 1，即信号量的初始值为应该为 1。

这需要简单说明一下信号量的初始值为什么是 1 而不是 0：

若初始值为 0，则第一个子线程在调用 sema_down 后，将会进入 while 循环，如下图：


```
while (sema->value == 0)
{
    list_push_back (&sema->waiters, &thread_current ()->elem);
    thread_block ();
}
```

此时第一个子线程会调用 `thread_block` 方法将自己阻塞掉，进入 `waiters` 队列，后面的子线程也会进入 `waiters` 队列，`lock_release` 永远无法被调用，这些子线程也将在 `waiters` 队列中无法被唤醒。

回到主线程 `test_priority_fifo`:

```
thread_set_priority (PRI_DEFAULT + 2);
```

将主线程优先级调高至 `PRI_DEFAULT + 2`

接下来为一个循环体，循环 `THREAD_CNT` (16) 次:

```
d->id = i;
d->iterations = 0;
d->lock = &lock;
d->op = &op;
thread_create (name, PRI_DEFAULT + 1, simple_thread_func, d);
```

前面四句为子线程信息的初始化。然后调用 `thread_create` 函数创建 `simple_thread_func` 子线程，每个子线程的优先级均设置为 `PRI_DEFAULT + 1`，比此时主线程的优先级低，不用担心主线程被子线程抢占。

```
thread_set_priority (PRI_DEFAULT);
/* All the other threads now run to termination here. */
ASSERT (lock.holder == NULL);
```

- 然后将主线程的优先级调低至 `PRI_DEFAULT`，比子线程低，子线程会抢占主线程的 `cpu` 资源。
- 当主线程继续执行 `ASSERT` 语句时，子线程应该已经全部执行完毕。
- 创建的这 16 个子线程的调度顺序前面已经分析过了，即每个线程循环均循环 16 次，每次循环线程按照 0-15 的编号被 `cpu` 调度。

最后的 `for` 循环输出缓冲区所储存的这个测试的过程信息。

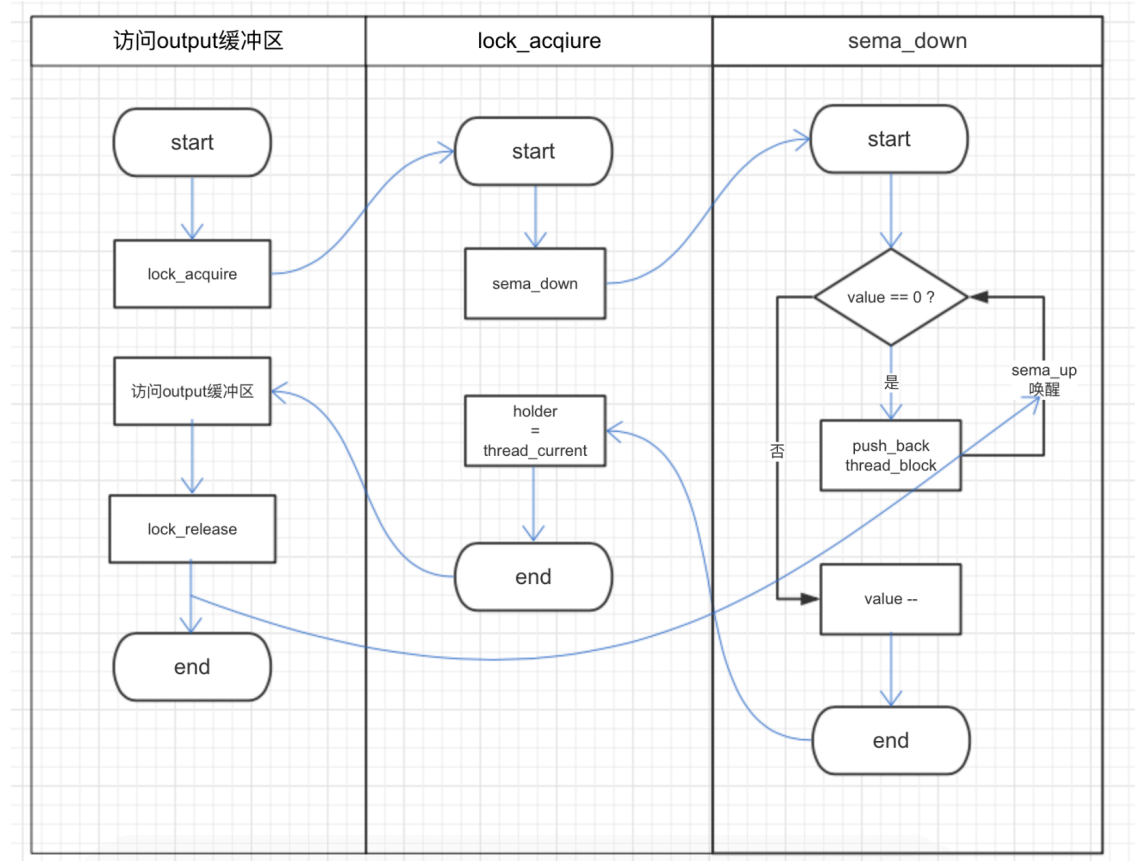
以上便是 `priority-fifo` 测试的详细分析。

结果分析:

```
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.
```

如图：与预期的一样，循环 16 次，每次子线程按照 0-15 的编号按顺序被 cpu 调度。测试通过。

流程图:



加锁与解锁流程图

(二) 实验思路与代码分析

实现优先级调度，是为了提高系统的响应特性。“抢占”顾名思义，是一个线程强行抢夺另一个线程的 cpu 资源。当发生优先级抢占时，正在 cpu 运行的线程即使在逻辑上还可以继续运行（即时间片还没有结束），也会被更高优先级的线程立即抢占资源。

要达到目的，我们只需要考虑这种情况：没有正在运行的线程，其优先级，比正在运行的线程的优先级更高。那么何时会出现这种情况呢？略加思考不难得出答案：

- 1) 正在运行的线程其优先级主动降低，ready_list 中可能有优先级更高的线程。
- 2) 当创建一个新的线程，该线程的优先级比正在 cpu 中运行的线程的优先级更高。

第一种情况对应与 thread.c 中的 thread_set_priority 函数，只要在设置完当前线程的优先级后，检查 ready_list 中是否有更高优先级的线程即可。若有，则调用 thread_yield 函数将其切出 cpu，并重新进行调度。由于上一个实验已经实现了 ready_list 中的元素按照优先级排列，所以重新调度的线程一定是优先级最高的，代码如下：

```
thread_current ()->priority = new_priority;
if (new_priority <
    list_entry (list_max (&ready_list, cmpa, NULL), struct thread, elem)->priority)
    thread_yield();
```

第二种情况出现在新建一个线程的时候，即 thread_create 函数中。若创建的线程的优先级高于正在运行的线程的优先级，则调用 thread_yield 函数将其切出 cpu，并重新调度。代码如下：

```
if (priority > thread_current ()->priority)
    thread_yield();
```

3. 实验结果

```
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
16 of 27 tests failed.
make: *** [check] 错误 1
shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$
```

如上图所示，实现优先级抢占后多通过了 3 个测试，截图如下：

```
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
```

```
pass tests/threads/priority-change
```

4. 回答问题

- 1) 如果没有修改 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。

```

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

```

```

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

```

如上图所示：只通过一个测试：priority-fifo;priority-change 和 priority-preempt 均没有通过失败。

- priority-change:

若没有修改 thread_create 的代码，该测试的实际输出如下：

```

Executing 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2. 1
(priority-change) Thread 2 should have just lowered its priority. 2
(priority-change) Thread 2 now lowering priority. 3
(priority-change) Thread 2 exiting. 4
(priority-change) Thread 2 should have just exited. 5
(priority-change) end
Execution of 'priority-change' complete.

shuzhan@16339038shuzhan:~/PINTOS/pintos/src/threads/build$ =====

```

期望的输出顺序为 1、3、2、4、5，而实际的输出顺序如图为 1、2、3、4、5。

产生这种现象的原因：

- 因为在创建函数时没有考虑优先级抢占，所以即使主线程创建了一个比自己优先级高的子线程，也不会被抢占，于是按顺序输出 1、2 语句。
- 当主线程调用 thread_set_priority 调低优先级，由于 thread_set_priority 考虑了优先级抢占，于是会被子线程抢占，按顺序输出 3、4。
- 最后回到主线程，输出 5。

- priority-preempt:

若没有修改 `thread_create` 代码，该测试的实际输出如下：

```
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!

shuzhan@16339038shuzhan:~/Pintos/pintos/src/threads/build$ =====
```

期望的输出顺序是 2、1，而实际的输出顺序为 1、2。

产生这种现象的原因：

- 当主线程创建一个优先级更高的子线程时，由于 `thread_create` 函数并没有考虑优先级抢占问题，所以会接下来输出 1 语句；然后子线程被分配到 cpu 后，才会输出 2。

• priority-fifo:

能通过测试的原因：

因为在调用 `thread_create` 时原本就不存在优先级的抢占，创建的子线程优先级为 `PRI_DEFAULT + 1`，比此时主线程的主线程的优先级 `PRI_DEFAULT + 2` 低。

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

区别：

- 信号量：关于信号量，我在知乎上搜到了一段非常有意思的解释，如下：

信号量就是在一个叫做互斥区的门口放一个盒子，盒子里面装着固定数量的小球，每个线程过来的时候，都从盒子里面摸走一个小球，然后去互斥区里面浪（？） ，浪开心了出来的时候，再把小球放回盒子里。如果一个线程走过来一摸盒子，得，一个球都没了，不拿球不让进啊，那就只能站在门口等一个线程出来放回来一个球，再进去。这样由于小球的数量是固定的，那么互斥区里面的最大线程数量就是固定的，不会出现一下进去太多线程把互斥区给挤爆了的情况。这是用信号量做并发量限制。

由此，我的理解是：**信号量是一种对线程获取资源的限制**。用上面的这段话举个例子：若信号量初始值为 n （盒子中有 n 个球），那么只能同时允许 n 个线程访问资源（进入互斥区）；若信号量的值变为了 0，则说明此时有 n 个线程在访问资源，新来的线程只能排队等待。

- 锁：

锁是比信号量更加**专一**的存在，即锁规定了一次只能有一个线程访问资源。如果说信号量是限制访问，那么锁就可以说是锁定访问。锁与信号量相比，更加强调了拥有者的概念，即在 `lock` 结构体中的 `holder` 变量，这就说明：哪个线程上的锁，那么必须还是这个线程解锁。

如果以上面截图中的话为例子，那么一开始盒子中只有一个球，永远不可能同时有两个线程访问资源。

信号量的值可以为非负数，锁的值只能是 0 和 1。

联系：

我的理解是锁是**专一**的信号量，即当信号量的值只能是 0 和 1 时，那么它起到的作用就是锁。

以 priority-fifo 测试中的锁为例：信号量的初始值为 1，表示此时允许有一个线程访问 output 缓冲区，当它变为 0 时，后面想要访问的线程只能进入 waiters 队列中等待，只有当锁的拥有者解锁，值变回 1 时，才能允许 waiters 中的一个线程访问资源。

3) 考虑优先级抢占调度后，重新分析 alarm-priority 测试

alarm-priority 重分析：

当实现优先级抢占后，主线程与子线程的调度顺序会变得跟以前不一样。调度过程如下：

```
sema_init (&wait_sema, 0);
```

在主线程 test_alarm_priority（此时优先级为 PRI_DEFAULT31）中，先初始化信号量，信号量的初始值为 0。

```
thread_create (name, priority, alarm_priority_thread, NULL);
```

在接下来的 for 循环中，调用 thread_create 方法来创建 10 个子线程。这 10 个子线程的优先级为 21-30，均比此时主线程的优先级低，所以此时不会发生子线程抢占主线程 cpu 资源的情况。

```
thread_set_priority (PRI_MIN);
```

然后主线程调用此方法，将自己的优先级调至最低（0）。

这时，这 10 个子线程便会抢占主线程的 cpu 资源。当 10 个子线程都执行完后，主线程才有机会得到 cpu 资源，此时信号量的值为 10。

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

主线程最后执行这个 for 循环，每次循环信号量的值减 1。（因为信号量的值不为 0，所以每次调用 sema_down 函数，主线程都不会被阻塞，而是直接减 1，直到为 0）

5. 实验感想

这次的实验还是蛮简单的（应该是最简单的一次了，要好好珍惜），两三行代码就可以多通过 3 个 test。Test 分析起来也相对轻松，没有像第一次时遇到大量的问题无法解决。最大的收获还是对 priority-fifo 文件的分析，通过厘清 lock_acquire()、lock_release()、sema_down()、sema_up 这些函数之间的调度

关系，以及主线程与子线程之间的调度关系、整个测试的流程，总算是对锁和信号量的概念有了一个初步的认识。接下来的实验就与锁和信号量息息相关了，希望在接下来的学习中，通过敲代码、做实验，能够对这些知识点能够有一个更加全面且深入的理解。