目录

1. 实验目的

1） 理解优先级抢占调度对于操作系统的重要性。
2） 实现一个抢占式优先级调度算法。
3） 理解 pintos 中 lock 和信号量。

2. 实验过程

(一)Test 分析

1) Priority-change

i. 测试内容

降低测试线程的优先级，使其不再是系统中最高优先级的线程，检测其是否能被立即调度。

ii. 测试要求

按照优先确保优先级高的线程才是当前线程，当前线程优先级不是最高的时候，停下当前线程去执行优先级高的线程。我们看看其输出要求。

```
(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
```

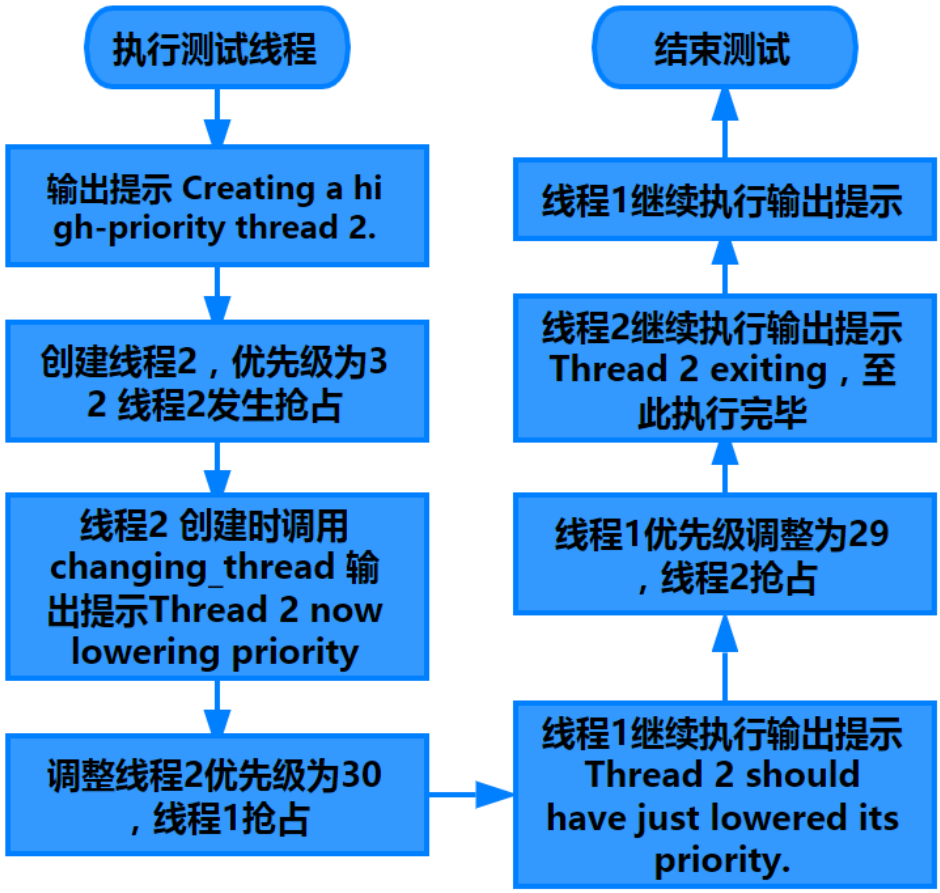
iii. 测试分析

我们先看一下测试主线程做了什么：

```
  msg ("Creating a high-priority thread 2.");//1
  thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL);
  msg ("Thread 2 should have just lowered its priority.");//2
  thread_set_priority (PRI_DEFAULT - 2);
  msg ("Thread 2 should have just exited.");//3
}
static void changing_thread (void *aux UNUSED)
{
  msg ("Thread 2 now lowering priority.");//4
  thread_set_priority (PRI_DEFAULT - 1);
  msg ("Thread 2 exiting.");//5
}
```

测试主线程开始优先级为（未定义值 31），先输出 1 号信息，之后创建线程 2 优先级为 32，所以此时执行线程 2 创建时传入的 changing_thread 函数，查看其要求输出我们发现，现在测试主线程应该是被中断回到 ready 队列，然后输出 4 号信息，并降低 2 号线程优先级为 30，之后我们看其要求输出为 2 号信息，意味着现在线程 2 被调度回 ready 队列，之后线程 1 降低优先级为 29，此时应输出信息 5，所以此时线程 1 调度回 ready 队列，之后线程 2 执行完毕，线程 1 继续执行输出信息 3。

整个测试执行顺序如下流程：



iv. 测试结论

如果想要通过这个测试那么就需要我们在线程创建时和线程优先级改变时实现优先级抢占调度。我们看到之前测试时留下的 result，可以发现我们之前是没有发生抢占的导致的主线程直接运行至 end，check 时没有读入子线程 2 输出的任何信息，虽然他在 end 之后确实输出了，但是测试已经结束。

```
(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 should have just exited.
(priority-change) end
```

2) Priority-preempt

i. 测试内容

确定高优先级线程时真抢占。

ii. 测试要求

在测试主线程中创建一个优先级更高的子线程，子线程创建时调用，函数输出，此时主线程应该是在 ready 队列中等待，直至子线程执行完毕才继续执行，要求输出如下：

```
(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
```

iii. 测试分析

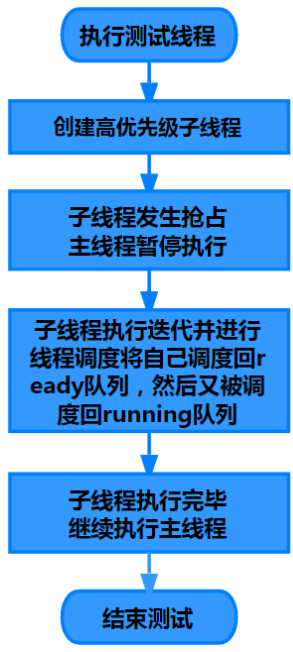
```
/* Make sure our priority is the default. */
ASSERT (thread_get_priority () == PRI_DEFAULT);

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

测试主线程开始优先级为（未定义值 31），此处为了断言主线程优先级一定为 31，之后创建测试用子线程。因为此处子线程优先级为 32，于是主线程被抢占，我们看看子线程干了什么。

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

在其创建调用的函数，迭代输出信息同时还是使用了 thread_yield，作用是调度测试，看看这个线程会不会接着回到 running 队列。以此确认其优先级最高的是真的抢占。之后等待子线程执行完毕之后再执行主线程，然后结束测试，才能得到要求输出。

整个测试执行顺序如右图：



iv. 测试结论

如果想要通过这个测试那么就需要我们在线程创建时实现优先级抢占调度。于此同时在线程调度的时候要确保优先级最高的线程位于 ready 队列的头部，此处上次实验已经实现。如果没有实现任意之一那么会发生的是测试主线程直接输出并提前结束 check，导致 fail。虽然说我们可以在后面看到线程迭代输出，但是出现在 end 之后，此时 check 已经完成。

3) Priority-fifo

i. 测试内容

创建多个相同优先级的子线程并确保他们按照创建顺序迭代运行。

ii. 测试要求

在测试主线程中创建一定数量的优先级相同的线程，要求执行他们的时候是 fifo（先进先出）的。输出结果如下（重复 16 次）：

```
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

iii. 测试分析

我们直接看到主线程创建子线程的部分：

```
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);
  }
```

测试主线程在创建测试用的子线程的时候首先提高了及自己优先级确保创建子线程时子线程不会因为优先级被子线程抢占。创建完毕后降低自己的优先级使子线程运行。之后我们看看子线程调用的函数内部的东西：

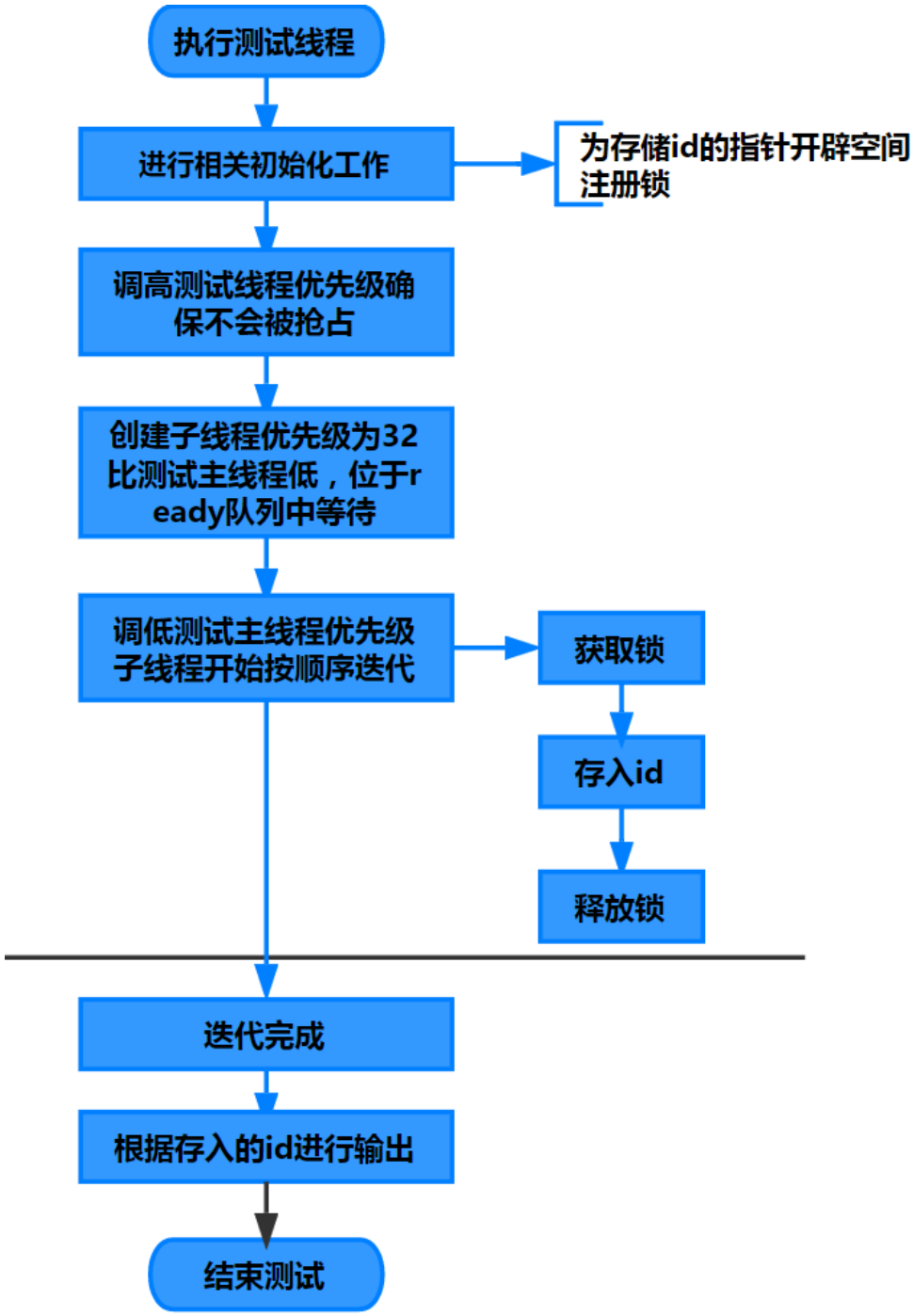
```
for (i = 0; i < ITER_CNT; i++)
  {
    lock_acquire (data->lock);
    *(*data->op)++ = data->id;
    lock_release (data->lock);
    thread_yield ();
  }
```

这其实只是用一个迭代循环迭代次数固定已给出，使用了锁的机制来确保每个输出储存位置上同时只有一个 id 记录，之后调度下一条线程，因为优先级相同，所以在调度的时候会被放在主线程之前的最末的位置。

之后我们在回到主线程继续执行。

```
ASSERT (lock.holder == NULL);

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++;
  }
```

通过之前储存的线程 id 输出每个线程的迭代顺序。测试过程流程如下图：



我们可以看到此处储存 id 使用了二维指针**op，此处和我们之前所分析过的样例，我们能发现这次的储存 id 的方式比较特别，之前都是使用一维指针储存 id。我们看看他的存入 `*(*data->op)++ = data->id;` 这里存入只用了一维，另一维没有使用，再看看读出数据。 `d = data + *output;` 我们不难看出其存入的时候存入的是 op[i]，但是读出的时候很特别是读出 op[op[i]]中的 id。这样的做我们只能得出一个结论，只要你调度实现没问题，即使你的线程创建时 id 不是 0-15 有序，那么输出的时候会变成强行有序，也就是说这样是为了不让创建时的时间差导致的线程在 ready 队列中的顺序出现问题，强行以这种方式确保输出正确。

此外我们可以发现在没有进行优先级抢占调度的修改时其输出的是

```
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) enUnexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
```

出现这个结果的主要原因是这个时候其实主线程已经结束，内存被释放，但是子线程还在执行，子线程访问已经不存在的内存造成的内存错误。

iv. 测试结论

如果我们要通过这是测试就需要确保优先级抢占正常，高优先级的不会被低优先的的抢占，同时相同优先级的进程不会被抢占，相同优先级的按照 fifo 的顺序执行即可。

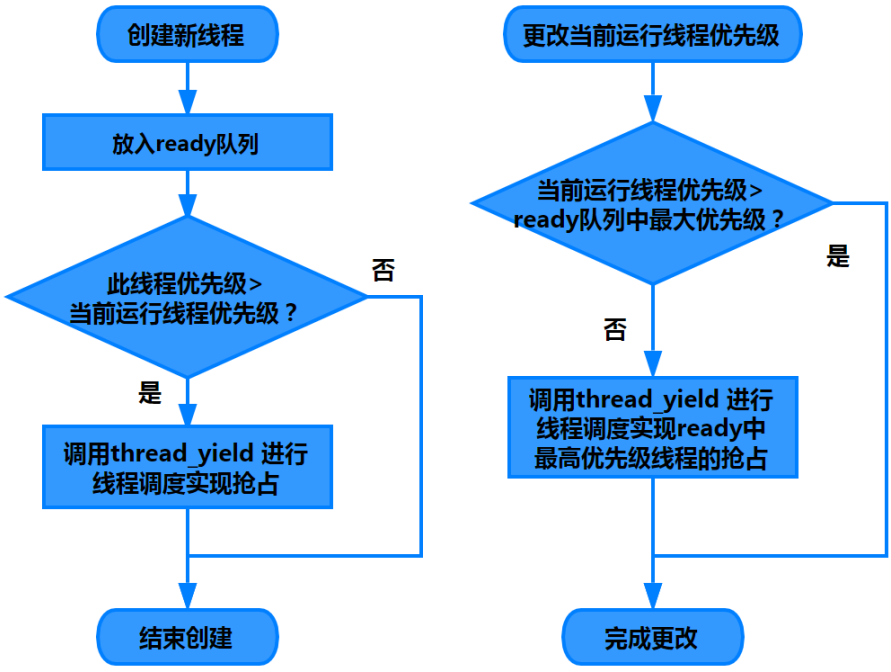
(二)分析及实现

通过对以上测试的分析，我们只要我们需要做两件事情，同时还有一个地方要主要。

第一件事情是创建新线程时要根据优先级进行抢占，那么我们需要判断创建的线程其优先级是不是大于当前线程，大于的话就抢占，否则扔到 ready 里待着，让其自动根据优先级在 ready 中排队即可。

第二件事情是我们需要在当前线程的优先级更改之后检测其是不是还是最高优先级的线程，如果不是的话他就应该被调度出 running 队列回到 ready 队里中。

要注意的一点是优先级抢占同优先级应该是 FIFO 的，所以所有的判断条件都是只能大于。具体要实现的两个抢占流程如下流程图：



要实现第一个目标，创建线程时实现优先级抢占很简单，我们只要在在线程穿件的最后一步把线程放入 ready 队列中在加一步判断其是否大于当前运行线程的优先级的条件判断即可。我们可以使用 thread_current 获取当前进程，使用 thread_yield 实现调度抢占具体实现如下：

```
/* Add to run queue. */
thread_unblock (t);
if (thread_current ()->priority < priority)
  thread_yield ();
```

实现第二个目标，重设当前运行线程优先级时判断其是否应该被抢占，只需要在其修改完优先级之后加一句条件判断即可，但是我们需要获得 ready 队列中优先级最大的线程，这里我们不难想到由于之前的实验我们已经确保了 ready 队列中的线程是从大到小有序的，因为我们已经修改了将线程放入 ready 的函数，确保其有序放入了，那我们就可以很方便之间从 ready 队列中取出第一个线程，他即是 ready 队列中优先级最大的线程。现在我们就需要去找一个取出 ready 队列中第一个线程的函数，我们在 list.h 中很容易发现 list_begin：

```
/* List traversal. */
struct list_elem *list_begin (struct list *);
```

他能返回一个 list_elem，这样我们就能获得 ready 队列的第一个元素，这时候我们使用 list_entry 这个宏定义就可以获得这个线程之后我们只需要判断需不需要调度即可。实现如下：

```
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
  if(list_entry(list_begin(&ready_list),struct thread,elem)
                                    ->priority>new_priority)
    thread_yield();
```

其实还有另外一种方法，如果我们的 ready 中不是以优先级有序排序的那么我就需要`struct list_elem *list_max (struct list *, list_less_func *, void *aux);`这个函数其遍历整个 ready 返回我们需要的线程，但是我们需要传入比较优先级的函数因为我们之前已经实现所以现在直接调用即可。实现如下：

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

  if(list_entry(list_max (&ready_list,
          (list_less_func *) &thread_cmp_priority, NULL),
                  struct thread,elem)->priority>new_priority)
    thread_yield();
}
```

做完这两步我们的实验就算完成了。

3. 实验结果

1） 完成后 make check 结果

```
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
root@15352008蔡荣裕:~/pintos/src/threads/build#
```

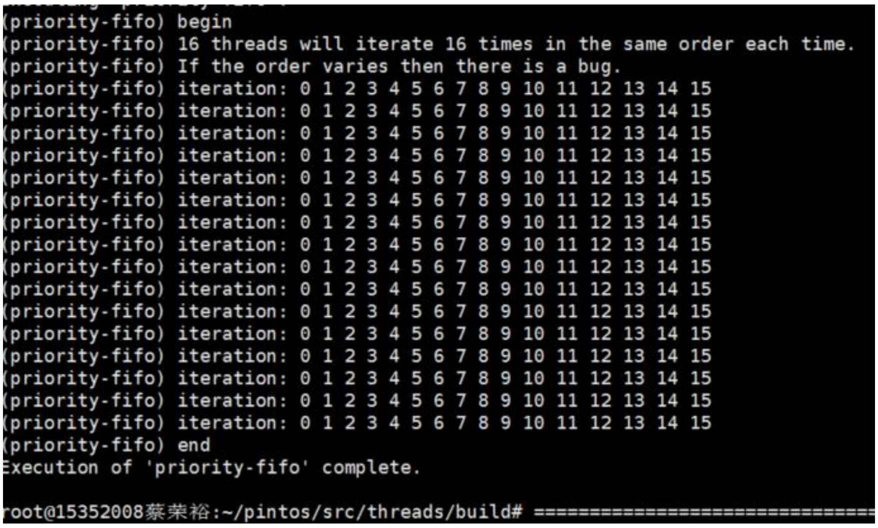
2） Priority-change 结果

```
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.

root@15352008蔡荣裕:~/pintos/src/threads/build# ==================
```

3） Priority-preempt 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.

root@15352008蔡荣裕:~/pintos/src/threads/build# ========================
```

4） Priority-fifo 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.

root@15352008蔡荣裕:~/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？**

没有考虑修改 thread_create 函数，次次需要通过的 3 个 test 中只有 fifo 这个测试能通过，其他的两个不能通过，因为此时 fifo 在创建时并不需要优先级抢占，只需要在 set 优先级的时候能进行抢占即可，而其余两个测试都需要创建时就是有优先级抢占。

此时的情况为，只有当前线程优先级改变才会发生抢占，创建时不会发生抢占。那么我们之前分析过了 priority-change 和 priority-preemet 这两个测试如果要通过那么就需要有创建时抢占的功能才能透过测试否则无法通过测试，而 priority-fifo,只要能在调整优先级之后进行抢占即可创建时并不需要抢占。所以情况应该是 fail 18/27。Make check 结果如下

```
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
root@15352008蔡荣裕:~/pintos/src/threads/build#
```

我们但看这两个不能通过的测试样例首先是 priority-change

```
Acceptable output:
  (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
Differences in `diff -u' format:
  (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 now lowering priority.
  (priority-change) Thread 2 exiting.
  (priority-change) Thread 2 should have just exited.
  (priority-change) end
```

其出现了什么情况呢？就是 thread 2 now lowering priority 这句话输出位置错了，现在输出这句是在测试主线程调整优先级之后，也就是说实在主线程调整优先级之后线程 2 才进行了抢占，之后虽然线程 2 调了一次优先级，但调整后的优先级仍比测试主线程高 1，所以线程 2 继续输出直至运行完毕，之后回到测试主线程执行继续输出。但是要求的输出在其创建时就应该输出了。所以其创建时没有发生抢占这就是没有修改 thread create 的结果。

再来看看 priority-preempt 的输出，我们发现其测试子线程的迭代输出根本没有被 check 检测到，为什么根据其测试流程如果子线程创建时没有抢占那么主线程将直接运行到结束其主线程结束了，代表测试完成了。那么子线程即使跑了，然后输出信息，但是因为其输出在 end 之后，所以不会被 check 到，然后也会显示，缺少这子线程的这几句输出。

```
Acceptable output:
  (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
Differences in `diff -u' format:
  (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
```

测试中我们还能发现一个很有趣的现象有的同学只修改 set 部分结果是 17/27，通过了理论上不能通过 priority-change，而且这种现象只存在于 14.04 以上（不包括）的系统中，所以为了探究一下原因配置了一个 16.04 的系统，做了和当前 14.04 系统相同的修改真的得到了 17/27 的诡异结果。

```
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
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
17 of 27 tests failed.
```

我们单跑一下 change 测试看看为什么会这样：

```
(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.

root@15352008:~/pintos/src/threads/build# ========================
```

此处的确输出了.ck 文件中需要的正确的结果，我们研究一下为什么，研究方法在每局 msg 之前加一句 printf 输出当前时间，修改方式如下图，这是我们再来看看结果

```
printf("%d",timer_ticks());msg ("Creating a high-priority thread 2.");//1
```

结果如下：

```
(priority-change) begin
25(priority-change) Creating a high-priority thread 2.
33(priority-change) Thread 2 should have just lowered its priority.
37(priority-change) Thread 2 now lowering priority.
45(priority-change) Thread 2 exiting.
50(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.

root@15352008:~/pintos/src/threads/build# ========================
```

这个时候我们可以看到加了时间之后结果突然正常，是我们预计的不会通过的结果。那我们只能说明一点这个测试运行是正常的。那为什么呢？我猜想一个可能原因 msg，因为除了输出异常之外完全没事，所以我们把所有的句子改成用 printf 输出我们可以看到 `18 of 27 tests failed.` 又回来了，结果也恢复正常，所以我感觉主要就是不同版本 msg 的优化问题。至于为什么加了时间输出之后为什么会正常，我认为应该就是获取系统时间时候的关中断保证了输出的时候不会因为轮转法的时间片被另一个线程抢占。

```
Acceptable output:
  (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
Differences in `diff -u' format:
  (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 now lowering priority.
  (priority-change) Thread 2 exiting.
  (priority-change) Thread 2 should have just exited.
  (priority-change) end
```

在看了一下 msg 的代码之后我觉得 msg 的输出效率应该背锅，首先我们测试一下输出时间使用 printf 输出最长的那句 Thread 2 should have just lowered its priority.时间是 39-32=7。我们再看一下出现错误输出的时候 Thread 2 should have just lowered its priority. 的输出时间也是就是使用 msg 的输出时间是 44-32=12 慢了将近 1 倍。这个时候我们得到了 Thread 2 should have just lowered its priority.这句话是从 32 开始进入 msg，44 时才完成输出工作。

```
32(priority-change) Thread 2 should have just lowered its priority.
44(priority-change) Thread 2 exiting.
```

现在我们测试一下之前排在他前面的那句 Thread 2 now lowering priority.输出问题，我们可以看到这句话的输出开始时间 33 符合要求因为其实在主线程调整优先级之后才开始执行的输出程序，但是我们可以看到他在 38 的时候就已经完成了输出工作，那么他应该是提前 msg 输出于屏幕上。

```
33(priority-change) Thread 2 now lowering priority.
38(priority-change) Thread 2 should have just lowered its priority.
```

所以我觉得应该就是不同系统对 msg 的优化问题。因为在 14.04 的系统中如果使用这种检测输出的方法，我们能看到如下情况：

```
3032(priority-change) Thread 2 should have just lowered its priority.
```

其对 msg 的处理明显不太一样，其到结束输出时间那句的输出在 msg 输出之前，也就是说 msg 输出过程不会影响到线程继续执行，不需要等待 msg 完成。

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

首先我们看一下 semaphore 和 lock 的定义：

```
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
  };
```

对比这 semaphore 和 lock，我们可以发现其实 lock 是使用 semaphore 作为基础的，lock 相对于 semaphore 锁了一个线程结构体指针 holder 用来记录当前使用 lock 的线程，我们再继续看看这两者先相关的函数，我们能看到 lock 相关函数都是基于 semaphore 的

相关函数实现的，同时我们找到 lock 的初始化函数，如下：

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

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

从这个函数我们可以发现 lock 被初始化的时候，实际上是初始化成了二元信号量。同时我们还能发现在 lock 相关的函数中都有使用断言调用如下函数：

```
bool
lock_held_by_current_thread (const struct lock *lock)
{
  ASSERT (lock != NULL);

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

他是为了确保在获得锁的时候，已获得锁的进程不会再次获得锁，从而造成死锁，而在释放时确保释放获得锁的线程，也是防止死锁，同时还确保在锁空的时候不能进行释放锁的操作。

至此，我们不难得出结论 lock，就是一个二元信号量，lock 是基于 semaphore 实现的，相比于信号量 lock 只能允许 1 个线程在临界区，而 semaphore 可以根据初始化时的 value 初始值决定有多少线程同时在临界区，但是 lock 有一点可以直接返回当前在使用 lock 的线程，而 semaphore 做不到这一点。同时 semaphore 的 PV 操作的顺序可以是无序的，换句话说就是可以先进行 V 操作在进行 P 操作，而 lock 必须严格按照'获得锁->释放锁'的的操作顺序。

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

首先我们看回到 alarm-priority 测试，之前分析过了此测试，现在主要分析修改前后区别。

其创建子线程效果如下，测试主线程的优先级默认是 31，所以修改过后主线程在创建子线程的时候不会被抢占因为主线程优先级更高。

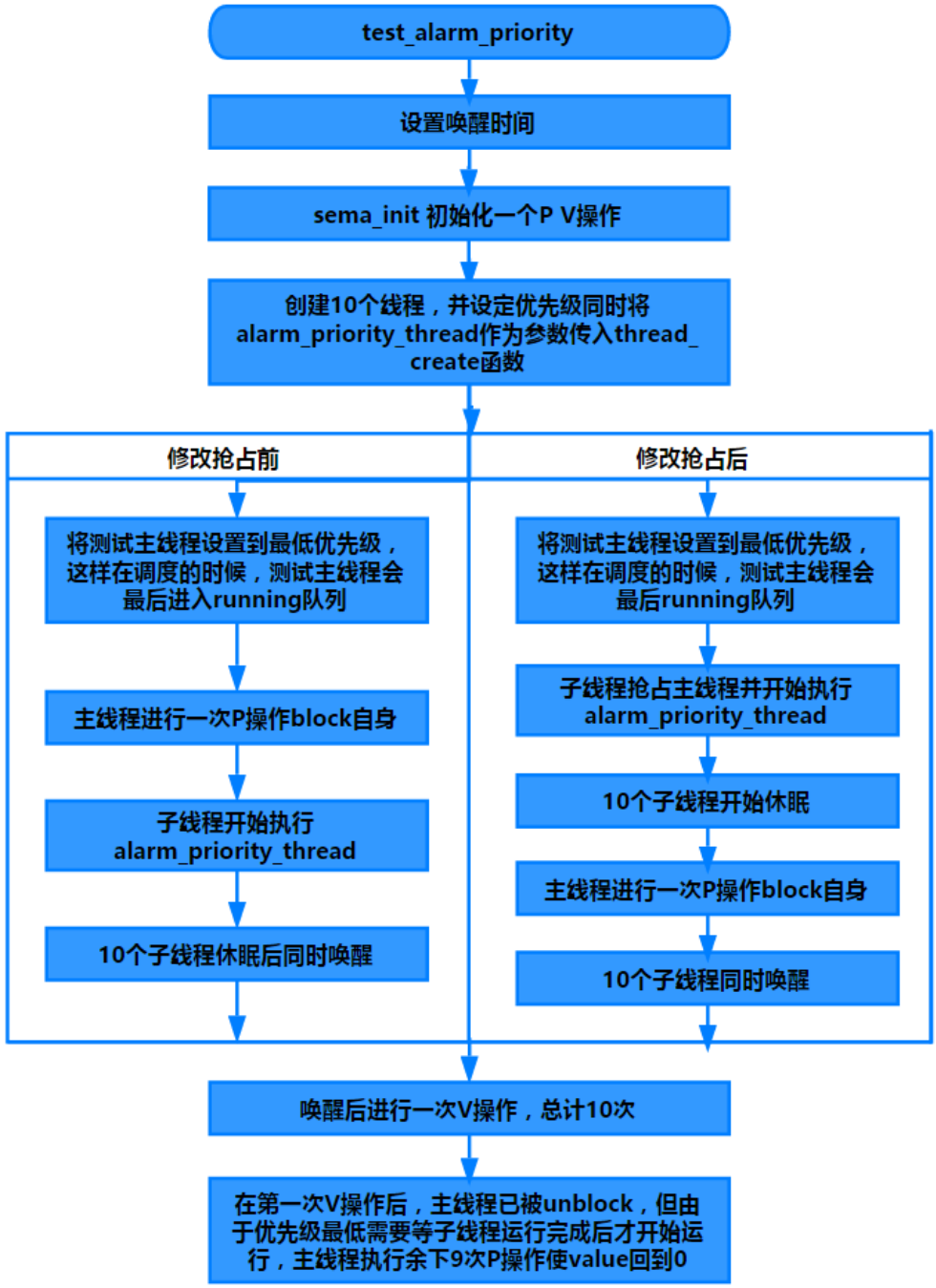
```
for (i = 0; i < 10; i++)
  {
    int priority = PRI_DEFAULT - (i + 5) % 10 - 1;
    char name[16];
    snprintf (name, sizeof name, "priority %d", priority);
    thread_create (name, priority, alarm_priority_thread, NULL);
  }
```

之后我们看看创建完成之后主线程干了什么：

```
thread_set_priority (PRI_MIN);

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

主线程调低了自己优先级，因为现在调整优先级之后，如果优先级低于 ready 队列

中的最大优先级线程那么就会被抢占，实际上此时 set 优先级之后的确被抢占了所以现在子线程都开始按照优先级开始进入休眠。当所有子线程进入休眠之后现在 ready 队列中只剩下这个最低优先级的测试主线程还没有执行完毕，于是主线程出来执行一次 P 操作将然后自己就被 P 操作 block 掉了。之后当子线程休眠结束时连续进行了 10 次 V 操作，虽然第一次就把测试主线程放回 ready 队列，但是其优先级最低所以最后执行。最后测试主线程执行余下 9 次 P 操作以平衡之前多做的 9 次 V 操作，到此线程执行完毕。



相比于没有优先级抢占的时候，也就是 set 优先级不会进行抢占的时候不同的一点在于没有优先级抢占的时候测试线程会通过降低优先级然后在利用 P 操作 block 掉自己，之后才是执行子线程的休眠。如果有了优先级抢占那么现在就是在 set 优先级之后测试主线程直接被子线程抢占，直至所有子线程开始休眠之后主线程才执行 P 操作 block 掉自己。想要证明这个过程很简单，我们在 P 操作前加一句时间输出，在线程休眠开始

前加一句输出时间对比即可。结果如下图，首先没有优先级抢占：

```
(alarm-priority) begin
 P 45  Sleep begin 47  Sleep begin 49  Sleep begin 51  Sleep begin
 53  Sleep begin 55  Sleep begin 58  Sleep begin 60  Sleep begin 6
2  Sleep begin 64  Sleep begin 66 (alarm-priority) Thread priority
 30 woke up.
 V 530 (alarm-priority) Thread priority 29 woke up.
 V 536 (alarm-priority) Thread priority 28 woke up.
 V 543 (alarm-priority) Thread priority 27 woke up.
 V 549 (alarm-priority) Thread priority 26 woke up.
 V 556 (alarm-priority) Thread priority 25 woke up.
 V 562 (alarm-priority) Thread priority 24 woke up.
 V 568 (alarm-priority) Thread priority 23 woke up.
 V 574 (alarm-priority) Thread priority 22 woke up.
 V 581 (alarm-priority) Thread priority 21 woke up.
 V 588  P 590  P 591  P 592  P 592  P 593  P 594  P 594  P 595  P
596 (alarm-priority) end
```

之后一图优先级抢占：

```
(alarm-priority) begin
 Sleep begin 46  Sleep begin 48  Sleep begin 50  Sleep begin 52  S
leep begin 54  Sleep begin 57  Sleep begin 59  Sleep begin 61  Sle
ep begin 63  Sleep begin 65  P 67 (alarm-priority) Thread priority
 30 woke up.
 V 530 (alarm-priority) Thread priority 29 woke up.
 V 536 (alarm-priority) Thread priority 28 woke up.
 V 543 (alarm-priority) Thread priority 27 woke up.
 V 549 (alarm-priority) Thread priority 26 woke up.
 V 556 (alarm-priority) Thread priority 25 woke up.
 V 562 (alarm-priority) Thread priority 24 woke up.
 V 568 (alarm-priority) Thread priority 23 woke up.
 V 574 (alarm-priority) Thread priority 22 woke up.
 V 581 (alarm-priority) Thread priority 21 woke up.
 V 588  P 590  P 591  P 592  P 592  P 593  P 594  P 594  P 595  P
596 (alarm-priority) end
Execution of 'alarm-priority' complete.
```

对比我们可以发现其执行 P 操作的时间不同一个是在所有线程开始休眠前，一个是在所有线程开始休眠之后，很好的印证了之前的结论。