

# 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. 25	完成日期	2016. 5. 4

## 1. 实验目的

通过修改 pintos, 实现条件变量的优先级调度以及实现多级反馈队列调度算法。通过 mlfqs 系列测试和 priority-condvar 测试。

## 2. 实验过程

(1) test 解释:

priority-condvar:

首先我们来看源代码:

```
16 void
17 test_priority_condvar (void)
18 {
19     int i;
20
21     /* This test does not work with the MLFQS. */
22     ASSERT (!thread_mlfqs);
23
24     lock_init (&lock);
25     cond_init (&condition);
26
27     thread_set_priority (PRI_MIN);
28     for (i = 0; i < 10; i++)
29     {
30         int priority = PRI_DEFAULT - (i + 7) % 10 - 1;
31         char name[16];
32         snprintf (name, sizeof name, "priority %d", priority);
33         thread_create (name, priority, priority_condvar_thread, NULL);
34     }
35
36     for (i = 0; i < 10; i++)
37     {
38         lock_acquire (&lock);
39         msg ("Signaling...");
40         cond_signal (&condition, &lock);
41         lock_release (&lock);
42     }
43 }
```

以上是测试的主体部分, 我们首先将主线程的优先级设置为 PRI\_MIN, 然后在 for 循环内我们一共创建了 10 个子线程, 每个子线程的优先级由  $PRI\_DEFAULT - (i+7) \% 10 - 1$  公式给出, 优先级的范

围是在 21-30 以内，均高于主线程的优先级，因此每创建一个子线程之后，子线程都会抢占 cpu 并执行以下函数：

```
45 static void
46 priority_condvar_thread (void *aux UNUSED)
47 {
48     msg ("Thread %s starting.", thread_name ());
49     lock_acquire (&lock);
50     cond_wait (&condition, &lock);
51     msg ("Thread %s woke up.", thread_name ());
52     lock_release (&lock);
53 }
```

该函数中，每个线程会先申请锁 lock，并得到 lock。之后子线程执行 cond\_wait 函数，在 cond\_wait 函数中，锁 lock 会先被释放掉，然后子线程被挂起到 condition 的等待队列中。因此，10 个子线程均在 cond\_wait 函数中被先后挂起。此后主线程重新获得了 cpu。

接下来，我们进入测试的第二部分，主线程进入第二个 for 循环，在循环内主线程首先是获取锁 lock，接着执行 cond\_signal 使用条件量唤醒 condition 等待队列中的优先级最高的一个子线程。因此子线程抢占到 cpu，并开始执行 cond\_wait 中的剩余内容。然而，由于 lock 目前被主线程占有了，因此子线程在尝试获取锁 lock 的时候，会再次被 block 掉，放入到 lock 的等待队列中。接下来主线程接着获取 cpu 开始执行，并释放了锁 lock，这时 lock 等待队列中的那个子线程得到锁 lock 被唤醒，输出被唤醒的信息后，释放了 lock，并执行完毕。在 10 次循环中，每次循环均执行一次这个操作。由于我们设计的 condition 的等待队列是一个优先队列，因此每次被唤醒的线程优先级应该是呈递减顺序的。

因此我们预期的输出结果应该为：

```
6 (priority-condvar) begin
7 (priority-condvar) Thread priority 23 starting
8 (priority-condvar) Thread priority 22 starting
9 (priority-condvar) Thread priority 21 starting
10 (priority-condvar) Thread priority 30 starting
11 (priority-condvar) Thread priority 29 starting
12 (priority-condvar) Thread priority 28 starting
13 (priority-condvar) Thread priority 27 starting
14 (priority-condvar) Thread priority 26 starting
15 (priority-condvar) Thread priority 25 starting
16 (priority-condvar) Thread priority 24 starting
17 (priority-condvar) Signaling...
18 (priority-condvar) Thread priority 30 woke up.
19 (priority-condvar) Signaling...
20 (priority-condvar) Thread priority 29 woke up.
21 (priority-condvar) Signaling...
22 (priority-condvar) Thread priority 28 woke up.
23 (priority-condvar) Signaling...
24 (priority-condvar) Thread priority 27 woke up.
25 (priority-condvar) Signaling...
26 (priority-condvar) Thread priority 26 woke up.
27 (priority-condvar) Signaling...
28 (priority-condvar) Thread priority 25 woke up.
29 (priority-condvar) Signaling...
30 (priority-condvar) Thread priority 24 woke up.
31 (priority-condvar) Signaling...
32 (priority-condvar) Thread priority 23 woke up.
33 (priority-condvar) Signaling...
34 (priority-condvar) Thread priority 22 woke up.
35 (priority-condvar) Signaling...
36 (priority-condvar) Thread priority 21 woke up.
37 (priority-condvar) end
```

mlfqs-load-1:

首先我们来看源代码:

```
17 void
18 test_mlfqs_load_1 (void)
19 {
20     int64_t start_time;
21     int elapsed;
22     int load_avg;
23     //thread_mlfqs = 1 ;
24     ASSERT (thread_mlfqs);
25
26     msg ("spinning for up to 45 seconds, please wait...");
27
28     start_time = timer_ticks ();
29     for (;;)
30     {
31         load_avg = thread_get_load_avg ();
32         ASSERT (load_avg >= 0);
33         elapsed = timer_elapsed (start_time) / TIMER_FREQ;
34         if (load_avg > 100)
35             fail ("load average is %d.%02d "
36                 "but should be between 0 and 1 (after %d seconds)",
37                 load_avg / 100, load_avg % 100, elapsed);
38         else if (load_avg > 50)
39             break;
40         else if (elapsed > 45)
41             fail ("load average stayed below 0.5 for more than 45 seconds");
42     }
43
44     if (elapsed < 38)
45         fail ("load average took only %d seconds to rise above 0.5", elapsed);
46     msg ("load average rose to 0.5 after %d seconds", elapsed);
47
48     msg ("sleeping for another 10 seconds, please wait...");
49     timer_sleep (TIMER_FREQ * 10);
50
51     load_avg = thread_get_load_avg ();
52     if (load_avg < 0)
53         fail ("load average fell below 0");
54     if (load_avg > 50)
55         fail ("load average stayed above 0.5 for more than 10 seconds");
56     msg ("load average fell back below 0.5 (to %d.%02d)",
57         load_avg / 100, load_avg % 100);
58
59     pass ();
60 }
61
```

以上是测试的主体部分，主线程首先记录下了线程的开始时间 ticks，然后进入 for 循环中，在每次 for 循环内，我们都及时地获取操作系统的当前的 load\_avg 值，并时刻监控 load\_avg 的值，只有在 load\_avg 值的范围在 50-100 之间时才进行循环。当 load\_avg 的值小于 0 时，会触发断言，终止程序的执行；当 load\_avg 的值大于 100 时，说明 load\_avg 的数值有误超出了设定的

0%-100%的范围，同样会终止程序。当 load\_avg 的值大于 50 时，则会跳出循环，接着执行下面的语句。但是需要注意的是，系统并不会无限限制运行线程在该循环中反复执行，一旦时间超过了 45s，但是 load\_avg 还没有提升到要求的 50%，则同样说明 load\_avg 的计算有误，也会终止程序继续执行。

总结一下，该测试的第一部分其实就是要求我们在 38s-45s 之内将 load\_avg 的值由 0 提升到 0.5。而又因为本测试只存在一个线程，因此由 load\_avg 的计算公式可知， $\text{load\_avg}(n) = (59/60) * \text{load\_avg}(n-1) + 1/60$ ，利用一点简单的数学知识，我们可以求出通项公式为  $\text{load\_avg}(n) = 1 - (59/60)^n$ 。经过计算，当  $\text{load\_avg}(n)=0.5$  时，n 的值约为 41，恰好落在 38-45 的范围之内。

接下来，我们进入测试的第二部分，主线程被 timer\_sleep 函数睡眠了 10s，在这期间内，由于整个 cpu 中没有线程在执行，因此 load\_avg 的值会逐渐回落，其回落的公式为： $\text{load\_avg}(n) = 59/60 * \text{load\_avg}(n-1)$ ，因此通项公式为  $\text{load\_avg}(n) = 0.5 * (59/60)^n$ 。带入  $n=10$ ，我们可以得到  $\text{load\_avg}(10)=0.42$ ，所以在主线程重新被唤醒后，load\_avg 的值应该已经回落到了 0.5 以下。

因此我们预期的输出结果应该为：

```
15 Executing 'mlfqs-load-1':
16 (mlfqs-load-1) begin
17 (mlfqs-load-1) spinning for up to 45 seconds, please wait...
18 (mlfqs-load-1) load average rose to 0.5 after 41 seconds
19 (mlfqs-load-1) sleeping for another 10 seconds, please wait...
20 (mlfqs-load-1) load average fell back below 0.5 (to 0.43)
21 (mlfqs-load-1) PASS
22 (mlfqs-load-1) end
23 Execution of 'mlfqs-load-1' complete.
```

mlfqs-load-60:

首先我们来看源代码：

```
114 void
115 test_mlfqs_load_60 (void)
116 {
117     int i;
118
119     ASSERT (thread_mlfqs);
120
121     start_time = timer_ticks ();
122     msg ("Starting %d niced load threads...", THREAD_CNT);
123     for (i = 0; i < THREAD_CNT; i++)
124     {
125         char name[16];
126         snprintf(name, sizeof name, "load %d", i);
127         thread_create (name, PRI_DEFAULT, load_thread, NULL);
128     }
129     msg ("Starting threads took %d seconds.",
130         timer_elapsed (start_time) / TIMER_FREQ);
131
132     for (i = 0; i < 90; i++)
133     {
134         int64_t sleep_until = start_time + TIMER_FREQ * (2 * i + 10);
135         int load_avg;
136         timer_sleep (sleep_until - timer_ticks ());
137         load_avg = thread_get_load_avg ();
138         msg ("After %d seconds, load average=%d.%02d.",
139             i * 2, load_avg / 100, load_avg % 100);
140     }
141 }
```

以上是测试的主体部分，主线程首先记录下测试开始的时间，并随后创建了 60 个子线程，每个子线程的优先级是相同的，均为 PRI\_DEFAULT。由于主线程在执行的过程中优先级已经被降低了，因此每创建一个新线程都会发生优先级的抢占，每个子线程执行以下函数：

```

143 static void
144 load_thread (void *aux UNUSED)
145 {
146     int64_t sleep_time = 10 * TIMER_FREQ;
147     int64_t spin_time = sleep_time + 60 * TIMER_FREQ;
148     int64_t exit_time = spin_time + 60 * TIMER_FREQ;
149
150     thread_set_nice (20);
151     timer_sleep (sleep_time - timer_elapsed (start_time));
152     while (timer_elapsed (start_time) < spin_time)
153         continue;
154     timer_sleep (exit_time - timer_elapsed (start_time));
155 }

```

可以看到，每个线程都是先休眠到 10s，再在 while 循环中运行 60s，最后再休眠 60s。因此当我们创建完线程后，每个子线程都会先进入休眠，使得主线程重新获得 cpu。我们可以保证的是，当创建完 60 个子线程后，所花的时间还远不足 10s，因此当前 60 个子线程均被 block 掉了，就绪队列中只有主线程一个线程。接下来主线程进入了第二个循环中，在循环中主线程首先进入休眠，等到 start\_time+10s，也即子线程被唤醒的时刻。需要注意的是这里的 60 个子线程几乎是同时被唤醒的。

因此，在接下来的 60s 内，由于 60 个线程被唤醒，因此我们从输出结果中应该看到 load\_avg 的数值较为快速的逐步升高。并恰好在 60s 的适合达到了峰值。在接下来的 60-120s 时间里，由于子线程执行完了 while 循环，因此又进入了休眠状态，因此就绪队列中的线程个数又开始下降，因此 load\_avg 的值又开始下降。120s 后子线程执行完毕，只剩下主线程一个线程，因此 load\_avg 会继续下降。我们预期的输出结果应该是：

8	After 0 seconds, load average=1.00.	38	After 60 seconds, load average=37.48.
9	After 2 seconds, load average=2.95.	39	After 62 seconds, load average=36.24.
10	After 4 seconds, load average=4.84.	40	After 64 seconds, load average=35.04.
11	After 6 seconds, load average=6.66.	41	After 66 seconds, load average=33.88.
12	After 8 seconds, load average=8.42.	42	After 68 seconds, load average=32.76.
13	After 10 seconds, load average=10.13.	43	After 70 seconds, load average=31.68.
14	After 12 seconds, load average=11.78.	44	After 72 seconds, load average=30.63.
15	After 14 seconds, load average=13.37.	45	After 74 seconds, load average=29.62.
16	After 16 seconds, load average=14.91.	46	After 76 seconds, load average=28.64.
17	After 18 seconds, load average=16.40.	47	After 78 seconds, load average=27.69.
18	After 20 seconds, load average=17.84.	48	After 80 seconds, load average=26.78.
19	After 22 seconds, load average=19.24.	49	After 82 seconds, load average=25.89.
20	After 24 seconds, load average=20.58.	50	After 84 seconds, load average=25.04.
21	After 26 seconds, load average=21.89.	51	After 86 seconds, load average=24.21.
22	After 28 seconds, load average=23.15.	52	After 88 seconds, load average=23.41.
23	After 30 seconds, load average=24.37.	53	After 90 seconds, load average=22.64.
24	After 32 seconds, load average=25.54.	54	After 92 seconds, load average=21.89.
25	After 34 seconds, load average=26.68.	55	After 94 seconds, load average=21.16.
26	After 36 seconds, load average=27.78.	56	After 96 seconds, load average=20.46.
27	After 38 seconds, load average=28.85.	57	After 98 seconds, load average=19.79.
28	After 40 seconds, load average=29.88.	58	After 100 seconds, load average=19.13.
29	After 42 seconds, load average=30.87.	59	After 102 seconds, load average=18.50.
30	After 44 seconds, load average=31.84.	60	After 104 seconds, load average=17.89.
31	After 46 seconds, load average=32.77.	61	After 106 seconds, load average=17.30.
32	After 48 seconds, load average=33.67.	62	After 108 seconds, load average=16.73.
33	After 50 seconds, load average=34.54.	63	After 110 seconds, load average=16.17.
34	After 52 seconds, load average=35.38.	64	After 112 seconds, load average=15.64.
35	After 54 seconds, load average=36.19.	65	After 114 seconds, load average=15.12.
36	After 56 seconds, load average=36.98.	66	After 116 seconds, load average=14.62.
37	After 58 seconds, load average=37.74.	67	After 118 seconds, load average=14.14.
38	After 60 seconds, load average=37.48.	68	After 120 seconds, load average=13.67.



mlfqs-load-avg:

首先我们来看源代码:

```
125 void
126 test_mlfqs_load_avg (void)
127 {
128     int i;
129
130     ASSERT (thread_mlfqs);
131
132     start_time = timer_ticks ();
133     msg ("Starting %d load threads...", THREAD_CNT);
134     for (i = 0; i < THREAD_CNT; i++)
135     {
136         char name[16];
137         snprintf(name, sizeof name, "load %d", i);
138         thread_create (name, PRI_DEFAULT, load_thread, (void *) i);
139     }
140     msg ("Starting threads took %d seconds.",
141         timer_elapsed (start_time) / TIMER_FREQ);
142     thread_set_nice (-20);
143
144     for (i = 0; i < 90; i++)
145     {
146         int64_t sleep_until = start_time + TIMER_FREQ * (2 * i + 10);
147         int load_avg;
148         timer_sleep (sleep_until - timer_ticks ());
149         load_avg = thread_get_load_avg ();
150         msg ("After %d seconds, load average=%d.%02d.",
151             i * 2, load_avg / 100, load_avg % 100);
152     }
153 }
```

以上是测试的主体部分, 主线程首先记录下测试开始的时间, 并随后创建了 60 个子线程, 每个子线程的优先级是相同的, 均为 PRI\_DEFAULT。由于主线程在执行的过程中优先级已经被降低了, 因此每创建一个新线程都会发生优先级的抢占, 每个子线程执行以下函数:

```
155 static void
156 load_thread (void *seq_no_)
157 {
158     int seq_no = (int) seq_no_;
159     int sleep_time = TIMER_FREQ * (10 + seq_no);
160     int spin_time = sleep_time + TIMER_FREQ * THREAD_CNT;
161     int exit_time = TIMER_FREQ * (THREAD_CNT * 2);
162
163     timer_sleep (sleep_time - timer_elapsed (start_time));
164     while (timer_elapsed (start_time) < spin_time)
165         continue;
166     timer_sleep (exit_time - timer_elapsed (start_time));
167 }
```

可以看到, 每个线程都是先休眠到 $(10+i)s$ , 再在 while 循环中运行 60s, 最后再休眠到 120s。因此当我们创建完线程后, 每个子线程都会先进入休眠, 使得主线程重新获得 cpu。我们可以保证的是, 当创建完 60 个子线程后, 所花的时间还远不足 10s, 因此当前 60 个子线程均被 block 掉了, 就绪队列中只有主线程一个线程。接下来主线程进入了第二个循环中, 在循环中主线程首先进入睡眠, 等到  $start\_time+10s$ , 也即第一个子线程被唤醒的时刻。

因此, 在接下来的 60s 内, 60 个线程先后被唤醒 (基本上是每一秒唤醒一个线程), 因此我们从输出结果中应该看到 load\_avg 的数值逐步升高。这里需要与上一个 load-60 的测试相区分, 由于上一个测试是 60 个线程同时被唤醒, 因此其 load\_avg 的增长速度是明显更快的, 而本测试中

由于线程是先后唤醒的，尤其是在最初的一段时间，由于 alllist 中的线程数量比较少，因此增长并不会很明显。60s 之后，所有线程都被唤醒了，但是不幸的是第一个线程恰好也执行完了 while 循环，而进入了第二次休眠，直到第 120s 的时刻才会被唤醒。因此在 60s-120s 内，处于 ready 队列中的线程数量又开始下降。但是需要注意的是，load\_avg 的值并不会随着线程数量的减少而马上减少，load\_avg 开始减少的临界点应该是  $1/60 * \text{load\_avg} = \text{thread\_num}/60$ ，计算之后我们发现应该是在 90s 左右 load\_avg 才开始减少的。120s 之后，子线程全部执行完毕，只有主线程继续执行，因此 load\_avg 会继续较为快速地减少。预期的输出结果应该是：

```
16 (mlfqs-load-avg) begin
17 (mlfqs-load-avg) Starting 60 load threads...
18 (mlfqs-load-avg) Starting threads took 1 seconds.
19 (mlfqs-load-avg) After 0 seconds, load average=0.06.
20 (mlfqs-load-avg) After 2 seconds, load average=0.11.
21 (mlfqs-load-avg) After 4 seconds, load average=0.22.
22 (mlfqs-load-avg) After 6 seconds, load average=0.39.
23 (mlfqs-load-avg) After 8 seconds, load average=0.63.
24 (mlfqs-load-avg) After 10 seconds, load average=0.92.
25 (mlfqs-load-avg) After 12 seconds, load average=1.27.
26 (mlfqs-load-avg) After 14 seconds, load average=1.67.
27 (mlfqs-load-avg) After 16 seconds, load average=2.13.
28 (mlfqs-load-avg) After 18 seconds, load average=2.64.
29 (mlfqs-load-avg) After 20 seconds, load average=3.20.
30 (mlfqs-load-avg) After 22 seconds, load average=3.80.
31 (mlfqs-load-avg) After 24 seconds, load average=4.45.
32 (mlfqs-load-avg) After 26 seconds, load average=5.15.
33 (mlfqs-load-avg) After 28 seconds, load average=5.89.
34 (mlfqs-load-avg) After 30 seconds, load average=6.67.
35 (mlfqs-load-avg) After 32 seconds, load average=7.49.
36 (mlfqs-load-avg) After 34 seconds, load average=8.35.
37 (mlfqs-load-avg) After 36 seconds, load average=9.25.
38 (mlfqs-load-avg) After 38 seconds, load average=10.18.
39 (mlfqs-load-avg) After 40 seconds, load average=11.15.
40 (mlfqs-load-avg) After 42 seconds, load average=12.15.
41 (mlfqs-load-avg) After 44 seconds, load average=13.19.
42 (mlfqs-load-avg) After 46 seconds, load average=14.26.
43 (mlfqs-load-avg) After 48 seconds, load average=15.36.
44 (mlfqs-load-avg) After 50 seconds, load average=16.49.
45 (mlfqs-load-avg) After 52 seconds, load average=17.64.
46 (mlfqs-load-avg) After 54 seconds, load average=18.83.
47 (mlfqs-load-avg) After 56 seconds, load average=20.04.
48 (mlfqs-load-avg) After 58 seconds, load average=21.28.
49 (mlfqs-load-avg) After 60 seconds, load average=22.54.
50 (mlfqs-load-avg) After 62 seconds, load average=23.78.
51 (mlfqs-load-avg) After 64 seconds, load average=24.90.
52 (mlfqs-load-avg) After 66 seconds, load average=25.92.
53 (mlfqs-load-avg) After 68 seconds, load average=26.85.
54 (mlfqs-load-avg) After 70 seconds, load average=27.65.
55 (mlfqs-load-avg) After 72 seconds, load average=28.36.
56 (mlfqs-load-avg) After 74 seconds, load average=28.99.
57 (mlfqs-load-avg) After 76 seconds, load average=29.53.
58 (mlfqs-load-avg) After 78 seconds, load average=29.98.
59 (mlfqs-load-avg) After 80 seconds, load average=30.33.
60 (mlfqs-load-avg) After 82 seconds, load average=30.60.
61 (mlfqs-load-avg) After 84 seconds, load average=30.81.
62 (mlfqs-load-avg) After 86 seconds, load average=30.95.
63 (mlfqs-load-avg) After 88 seconds, load average=31.05.
64 (mlfqs-load-avg) After 90 seconds, load average=31.05.
65 (mlfqs-load-avg) After 92 seconds, load average=30.96.
66 (mlfqs-load-avg) After 94 seconds, load average=30.85.
67 (mlfqs-load-avg) After 96 seconds, load average=30.64.
68 (mlfqs-load-avg) After 98 seconds, load average=30.37.
69 (mlfqs-load-avg) After 100 seconds, load average=30.04.
70 (mlfqs-load-avg) After 102 seconds, load average=29.66.
71 (mlfqs-load-avg) After 104 seconds, load average=29.26.
72 (mlfqs-load-avg) After 106 seconds, load average=28.77.
73 (mlfqs-load-avg) After 108 seconds, load average=28.23.
74 (mlfqs-load-avg) After 110 seconds, load average=27.64.
75 (mlfqs-load-avg) After 112 seconds, load average=27.03.
76 (mlfqs-load-avg) After 114 seconds, load average=26.35.
77 (mlfqs-load-avg) After 116 seconds, load average=25.63.
78 (mlfqs-load-avg) After 118 seconds, load average=24.86.
79 (mlfqs-load-avg) After 120 seconds, load average=24.06.
80 (mlfqs-load-avg) After 122 seconds, load average=23.26.
81 (mlfqs-load-avg) After 124 seconds, load average=22.49.
82 (mlfqs-load-avg) After 126 seconds, load average=21.75.
83 (mlfqs-load-avg) After 128 seconds, load average=21.03.
84 (mlfqs-load-avg) After 130 seconds, load average=20.34.
85 (mlfqs-load-avg) After 132 seconds, load average=19.66.
86 (mlfqs-load-avg) After 134 seconds, load average=19.01.
87 (mlfqs-load-avg) After 136 seconds, load average=18.38.
88 (mlfqs-load-avg) After 138 seconds, load average=17.78.
89 (mlfqs-load-avg) After 140 seconds, load average=17.19.
90 (mlfqs-load-avg) After 142 seconds, load average=16.62.
91 (mlfqs-load-avg) After 144 seconds, load average=16.07.
92 (mlfqs-load-avg) After 146 seconds, load average=15.54.
93 (mlfqs-load-avg) After 148 seconds, load average=15.03.
94 (mlfqs-load-avg) After 150 seconds, load average=14.53.
95 (mlfqs-load-avg) After 152 seconds, load average=14.05.
96 (mlfqs-load-avg) After 154 seconds, load average=13.59.
97 (mlfqs-load-avg) After 156 seconds, load average=13.14.
98 (mlfqs-load-avg) After 158 seconds, load average=12.70.
99 (mlfqs-load-avg) After 160 seconds, load average=12.28.
00 (mlfqs-load-avg) After 162 seconds, load average=11.88.
01 (mlfqs-load-avg) After 164 seconds, load average=11.48.
02 (mlfqs-load-avg) After 166 seconds, load average=11.10.
03 (mlfqs-load-avg) After 168 seconds, load average=10.74.
04 (mlfqs-load-avg) After 170 seconds, load average=10.38.
05 (mlfqs-load-avg) After 172 seconds, load average=10.04.
06 (mlfqs-load-avg) After 174 seconds, load average=9.71.
07 (mlfqs-load-avg) After 176 seconds, load average=9.39.
08 (mlfqs-load-avg) After 178 seconds, load average=9.08.
09 (mlfqs-load-avg) end
```

mlfqs-block:

首先我们来看源代码：

```

22 void
23 test_mlfqs_block (void)
24 {
25     int64_t start_time;
26     struct lock lock;
27
28     ASSERT (thread_mlfqs);
29
30     msg ("Main thread acquiring lock.");
31     lock_init (&lock);
32     lock_acquire (&lock);
33
34     msg ("Main thread creating block thread, sleeping 25 seconds...");
35     thread_create ("block", PRI_DEFAULT, block_thread, &lock);
36     timer_sleep (25 * TIMER_FREQ);
37
38     msg ("Main thread spinning for 5 seconds...");
39     start_time = timer_ticks ();
40     while (timer_elapsed (start_time) < 5 * TIMER_FREQ)
41         continue;
42
43     msg ("Main thread releasing lock.");
44     lock_release (&lock);
45
46     msg ("Block thread should have already acquired lock.");
47 }

```

以上是测试的主体部分，主线程首先初始化一个锁 lock，然后占有该锁，接着主线程创建了一个优先级为 PRI\_DEFAULT 的线程 block，由于此时主线程优先级更高，因此会继续执行并进入 25s 的休眠中，接下来由子线程占有 cpu 并开始执行如下函数：

```

49 static void
50 block_thread (void *lock_)
51 {
52     struct lock *lock = lock_;
53     int64_t start_time;
54
55     msg ("Block thread spinning for 20 seconds...");
56     start_time = timer_ticks ();
57     while (timer_elapsed (start_time) < 20 * TIMER_FREQ)
58         continue;
59
60     msg ("Block thread acquiring lock...");
61     lock_acquire (lock);
62
63     msg ("...got it.");
64 }

```

可以看到子线程在函数中会先循环运行 20s，之后再申请锁 lock，但由于该锁已经被主线程所占有，因此子线程会在 P 操作时被 block 掉。之后主线程重新获取 cpu 并开始在一个 while 循环中执行 5s，在这 5s 内，主线程的优先级因为长时间占用 cpu 而会被调低，而子线程虽然被 block 掉，但是其优先级依然会被更新，并且由于长时间的等待而使得优先级升高。因此，加入我们对 block 的优先级变化计算是正确的话，当主线程执行完 5s 之后，子线程的优先级已经高于主线程。



程，因此在线程释放锁 lock 的时候，会发生优先级抢占。由于子线程得到锁 lock，并开始执行。因此预期的输出结果应该为：

```
16 (mlfqs-block) begin
17 (mlfqs-block) Main thread acquiring lock.
18 (mlfqs-block) Main thread creating block thread, sleeping 25 seconds...
19 (mlfqs-block) Block thread spinning for 20 seconds...
20 (mlfqs-block) Block thread acquiring lock...
21 (mlfqs-block) Main thread spinning for 5 seconds...
22 (mlfqs-block) Main thread releasing lock.
23 (mlfqs-block) ...got it.
24 (mlfqs-block) Block thread should have already acquired lock.
25 (mlfqs-block) end
26 Execution of 'mlfqs-block' complete
```

mlfqs-recent-1:

首先我们来看源代码：

```
109 void
110 test_mlfqs_recent_1 (void)
111 {
112     int64_t start_time;
113     int last_elapsed = 0;
114
115     ASSERT (thread_mlfqs);
116
117     do
118     {
119         msg ("Sleeping 10 seconds to allow recent_cpu to decay, please wait...");
120         start_time = timer_ticks ();
121         timer_sleep (DIV_ROUND_UP (start_time, TIMER_FREQ) - start_time
122                     + 10 * TIMER_FREQ);
123     }
124     while (thread_get_recent_cpu () > 700);
125
126     start_time = timer_ticks ();
127     for (;;)
128     {
129         int elapsed = timer_elapsed (start_time);
130         if (elapsed % (TIMER_FREQ * 2) == 0 && elapsed > last_elapsed)
131         {
132             int recent_cpu = thread_get_recent_cpu ();
133             int load_avg = thread_get_load_avg ();
134             int elapsed_seconds = elapsed / TIMER_FREQ;
135             msg ("After %d seconds, recent_cpu is %d.%02d, load_avg is %d.%02d.",
136                 elapsed_seconds,
137                 recent_cpu / 100, recent_cpu % 100,
138                 load_avg / 100, load_avg % 100);
139             if (elapsed_seconds >= 180)
140                 break;
141         }
142         last_elapsed = elapsed;
143     }
144 }
```

以上是测试的主体部分，主线程首先初始化休眠 10s，使得 recent\_cpu 的值回落。接下来，主线程记录下程序执行的开始时间，并进入 for 循环中。在 for 循环中，主线程一直处于运行的状态，且每过 2s 输出一次当前的 load\_avg 与 recent\_cpu。由于当前只有主线程一个线程一直在占用 cpu 执行，所以 recent\_cpu 应该是呈现出较快的增长趋势，而 load\_avg 也会增长，不过增长速度是符合我们之前推导出的公式  $load\_avg(n)=1-(59/60)^n$ 。因此预期的部分输出结果应该为：

```
16 (mlfqs-recent-1) begin
17 (mlfqs-recent-1) Sleeping 10 seconds to allow recent_cpu to decay, please wait...
18 (mlfqs-recent-1) After 2 seconds, recent_cpu is 7.39, load_avg is 0.03.
19 (mlfqs-recent-1) After 4 seconds, recent_cpu is 13.59, load_avg is 0.06.
20 (mlfqs-recent-1) After 6 seconds, recent_cpu is 19.60, load_avg is 0.10.
21 (mlfqs-recent-1) After 8 seconds, recent_cpu is 25.42, load_avg is 0.13.
22 (mlfqs-recent-1) After 10 seconds, recent_cpu is 31.06, load_avg is 0.15.
23 (mlfqs-recent-1) After 12 seconds, recent_cpu is 36.52, load_avg is 0.18.
24 (mlfqs-recent-1) After 14 seconds, recent_cpu is 41.80, load_avg is 0.21.
25 (mlfqs-recent-1) After 16 seconds, recent_cpu is 46.93, load_avg is 0.24.
26 (mlfqs-recent-1) After 18 seconds, recent_cpu is 51.89, load_avg is 0.26.
27 (mlfqs-recent-1) After 20 seconds, recent_cpu is 56.70, load_avg is 0.29.
28 (mlfqs-recent-1) After 22 seconds, recent_cpu is 61.35, load_avg is 0.31.
29 (mlfqs-recent-1) After 24 seconds, recent_cpu is 65.86, load_avg is 0.33.
30 (mlfqs-recent-1) After 26 seconds, recent_cpu is 70.22, load_avg is 0.35.
31 (mlfqs-recent-1) After 28 seconds, recent_cpu is 74.45, load_avg is 0.38.
32 (mlfqs-recent-1) After 30 seconds, recent_cpu is 78.54, load_avg is 0.40.
33 (mlfqs-recent-1) After 32 seconds, recent_cpu is 82.50, load_avg is 0.42.
34 (mlfqs-recent-1) After 34 seconds, recent_cpu is 86.33, load_avg is 0.43.
35 (mlfqs-recent-1) After 36 seconds, recent_cpu is 90.05, load_avg is 0.45.
36 (mlfqs-recent-1) After 38 seconds, recent_cpu is 93.64, load_avg is 0.47.
37 (mlfqs-recent-1) After 40 seconds, recent_cpu is 97.13, load_avg is 0.49.
38 (mlfqs-recent-1) After 42 seconds, recent_cpu is 100.49, load_avg is 0.51.
39 (mlfqs-recent-1) After 44 seconds, recent_cpu is 103.76, load_avg is 0.52.
40 (mlfqs-recent-1) After 46 seconds, recent_cpu is 106.92, load_avg is 0.54.
41 (mlfqs-recent-1) After 48 seconds, recent_cpu is 109.97, load_avg is 0.55.
42 (mlfqs-recent-1) After 50 seconds, recent_cpu is 112.93, load_avg is 0.57.
43 (mlfqs-recent-1) After 52 seconds, recent_cpu is 115.79, load_avg is 0.58.
44 (mlfqs-recent-1) After 54 seconds, recent_cpu is 118.57, load_avg is 0.60.
45 (mlfqs-recent-1) After 56 seconds, recent_cpu is 121.25, load_avg is 0.61.
46 (mlfqs-recent-1) After 58 seconds, recent_cpu is 123.85, load_avg is 0.62.
47 (mlfqs-recent-1) After 60 seconds, recent_cpu is 126.36, load_avg is 0.63.
```

mlfqs-fair-2 & mlfqs-fair-20:

由于这两个测试的内容与结构均十分相似，唯一的不同是在于创建的子线程数量不同。因此我们不妨把这两个测试放到一起来进行分析。

首先我们来看源代码：

```

65 static void
66 test_mlfqs_fair (int thread_cnt, int nice_min, int nice_step)
67 {
68     struct thread_info info[MAX_THREAD_CNT];
69     int64_t start_time;
70     int nice;
71     int i;
72     //thread_mlfqs = 1 ;
73     ASSERT (thread_mlfqs);
74     ASSERT (thread_cnt <= MAX_THREAD_CNT);
75     ASSERT (nice_min >= -10);
76     ASSERT (nice_step >= 0);
77     ASSERT (nice_min + nice_step * (thread_cnt - 1) <= 20);
78
79     thread_set_nice (-20);
80
81     start_time = timer_ticks ();
82     msg ("Starting %d threads...", thread_cnt);
83     nice = nice_min;
84     for (i = 0; i < thread_cnt; i++)
85     {
86         struct thread_info *ti = &info[i];
87         char name[16];
88
89         ti->start_time = start_time;
90         ti->tick_count = 0;
91         ti->nice = nice;
92
93         snprintf(name, sizeof name, "load %d", i);
94         thread_create (name, PRI_DEFAULT, load_thread, ti);
95
96         nice += nice_step;
97     }
98     msg ("Starting threads took %PRIu64 ticks.", timer_elapsed (start_time));
99
100    msg ("Sleeping 40 seconds to let threads run, please wait...");
101    timer_sleep (40 * TIMER_FREQ);
102
103    for (i = 0; i < thread_cnt; i++)
104        msg ("Thread %d received %d ticks.", i, info[i].tick_count);
105 }

```

以上是测试的主体部分，而两个测试的区别在于调用方式的不同。fair-2 的调用为：

```

30 void
31 test_mlfqs_fair_2 (void)
32 {
33     test_mlfqs_fair (2, 0, 0);
34 }

```

而 fair-20 的调用为：

```

36 void
37 test_mlfqs_fair_20 (void)
38 {
39     test_mlfqs_fair (20, 0, 0);
40 }

```

所以现在我们回到测试的主体部分中，我们发现测试首先是创建了若干个线程，由于 fair 测试把 nice\_step 均设置为了 0，因此所创建线程的 nice 值均不会发生变化，接下来，主线程进入

40s 的休眠状态，让子线程开始执行。子线程执行的函数为：

```
107 static void
108 load_thread (void *ti_)
109 {
110     struct thread_info *ti = ti_;
111     int64_t sleep_time = 5 * TIMER_FREQ;
112     int64_t spin_time = sleep_time + 30 * TIMER_FREQ;
113     int64_t last_time = 0;
114
115     thread_set_nice (ti->nice);
116     timer_sleep (sleep_time - timer_elapsed (ti->start_time));
117     while (timer_elapsed (ti->start_time) < spin_time)
118     {
119         int64_t cur_time = timer_ticks ();
120         if (cur_time != last_time)
121             ti->tick_count++;
122         last_time = cur_time;
123     }
124 }
```

可以看到子线程首先是睡眠了 5s，接着就进入 while 循环中开始不断地运行。在 while 循环中同时会记录下当前线程一共执行了多少个 ticks，以此反映其占用 cpu 的时间长短。这里由于 spin\_time 已经是一个确定的时间点，而 cpu 的时间是由子线程们共享的，因此在 nice 值相同的情况下，各子线程之间会频繁地发生优先级抢占，而每个子线程分到的 ticks 个数应该基本相同。由于 30s 一共是 3000 个 ticks，所以 fair-2 测试中 3000 个 ticks 由 2 个子线程平分，每个线程得到的 ticks 应该是 1500 左右，fair-20 测试中 3000 个 ticks 由 20 个子线程平分，每个线程得到的 ticks 应该是 150 左右。预期的输出结果为：

```
16 (mlfqs-fair-2) begin
17 (mlfqs-fair-2) Starting 2 threads...
18 (mlfqs-fair-2) Starting threads took 8 ticks.
19 (mlfqs-fair-2) Sleeping 40 seconds to let threads run, please wait...
20 (mlfqs-fair-2) Thread 0 received 1496 ticks.
21 (mlfqs-fair-2) Thread 1 received 1504 ticks.
22 (mlfqs-fair-2) end
16 (mlfqs-fair-20) begin
17 (mlfqs-fair-20) Starting 20 threads...
18 (mlfqs-fair-20) Starting threads took 49 ticks.
19 (mlfqs-fair-20) Sleeping 40 seconds to let threads run, please wait...
20 (mlfqs-fair-20) Thread 0 received 148 ticks.
21 (mlfqs-fair-20) Thread 1 received 148 ticks.
22 (mlfqs-fair-20) Thread 2 received 152 ticks.
23 (mlfqs-fair-20) Thread 3 received 148 ticks.
24 (mlfqs-fair-20) Thread 4 received 152 ticks.
25 (mlfqs-fair-20) Thread 5 received 152 ticks.
26 (mlfqs-fair-20) Thread 6 received 153 ticks.
27 (mlfqs-fair-20) Thread 7 received 151 ticks.
28 (mlfqs-fair-20) Thread 8 received 148 ticks.
29 (mlfqs-fair-20) Thread 9 received 148 ticks.
30 (mlfqs-fair-20) Thread 10 received 149 ticks.
31 (mlfqs-fair-20) Thread 11 received 148 ticks.
32 (mlfqs-fair-20) Thread 12 received 149 ticks.
33 (mlfqs-fair-20) Thread 13 received 149 ticks.
34 (mlfqs-fair-20) Thread 14 received 156 ticks.
35 (mlfqs-fair-20) Thread 15 received 153 ticks.
36 (mlfqs-fair-20) Thread 16 received 153 ticks.
37 (mlfqs-fair-20) Thread 17 received 152 ticks.
38 (mlfqs-fair-20) Thread 18 received 149 ticks.
39 (mlfqs-fair-20) Thread 19 received 149 ticks.
40 (mlfqs-fair-20) end
```



mlfqs-nice:

由于这两个测试的内容与结构均十分相似，唯一的不同是在于创建的子线程数量不同。因此我们不妨把这两个测试放到一起来进行分析。

首先我们来看源代码，该测试的主体函数部分与上一个 mlfqs-fair 测试完全相同：

```
65 static void
66 test_mlfqs_fair (int thread_cnt, int nice_min, int nice_step)
67 {
68     struct thread_info info[MAX_THREAD_CNT];
69     int64_t start_time;
70     int nice;
71     int i;
72     //thread_mlfqs = 1 ;
73     ASSERT (thread_mlfqs);
74     ASSERT (thread_cnt <= MAX_THREAD_CNT);
75     ASSERT (nice_min >= -10);
76     ASSERT (nice_step >= 0);
77     ASSERT (nice_min + nice_step * (thread_cnt - 1) <= 20);
78
79     thread_set_nice (-20);
80
81     start_time = timer_ticks ();
82     msg ("Starting %d threads...", thread_cnt);
83     nice = nice_min;
84     for (i = 0; i < thread_cnt; i++)
85     {
86         struct thread_info *ti = &info[i];
87         char name[16];
88
89         ti->start_time = start_time;
90         ti->tick_count = 0;
91         ti->nice = nice;
92
93         snprintf(name, sizeof name, "load %d", i);
94         thread_create (name, PRI_DEFAULT, load_thread, ti);
95
96         nice += nice_step;
97     }
98     msg ("Starting threads took %PRIId64 ticks.", timer_elapsed (start_time));
99
100    msg ("Sleeping 40 seconds to let threads run, please wait...");
101    timer_sleep (40 * TIMER_FREQ);
102
103    for (i = 0; i < thread_cnt; i++)
104        msg ("Thread %d received %d ticks.", i, info[i].tick_count);
105 }
```

以上是测试的主体部分，而两个测试的区别在于调用方式的不同。nice-2 的调用为：

```
42 void
43 test_mlfqs_nice_2 (void)
44 {
45     test_mlfqs_fair (2, 0, 5);
46 }
47
```

而 nice-10 的调用为：

```
48 void
49 test_mlfqs_nice_10 (void)
50 {
51     test_mlfqs_fair (10, 0, 1);
52 }
53
```

所以现在我们回到测试的主体部分中，我们发现测试首先是创建了若干个线程，由于 nice-2 测试把 nice\_step 设置为了 5，因此所创建的两个线程的 nice 值分别为 0 和 5。而 nice-10 测试吧 nice\_step 设置为了 1，因此所创建的十个线程的 nice 值为 0-9。接下来，主线程进入 40s 的休眠状态，让子线程开始执行。子线程执行的函数为：

```
107 static void
108 load_thread (void *ti_)
109 {
110     struct thread_info *ti = ti_;
111     int64_t sleep_time = 5 * TIMER_FREQ;
112     int64_t spin_time = sleep_time + 30 * TIMER_FREQ;
113     int64_t last_time = 0;
114
115     thread_set_nice (ti->nice);
116     timer_sleep (sleep_time - timer_elapsed (ti->start_time));
117     while (timer_elapsed (ti->start_time) < spin_time)
118     {
119         int64_t cur_time = timer_ticks ();
120         if (cur_time != last_time)
121             ti->tick_count++;
122         last_time = cur_time;
123     }
124 }
```

可以看到子线程首先是睡眠了 5s，接着就进入 while 循环中开始不断地运行。在 while 循环中同时会记录下当前线程一共执行了多少个 ticks，以此反映其占用 cpu 的时间长短。这里由于 spin\_time 已经是一个确定的时间点，而 cpu 的时间是由子线程们共享的，因此在 nice 值不同的情况下，各子线程之间会频繁地发生优先级抢占，而每个子线程分到的 ticks 个数应该是与其 nice 值大小成反比的，即 nice 值越大，得到的 ticks 数应该越小。由于 30s 一共是 3000 个 ticks，所以 nice-2 测试中 3000 个 ticks 由 2 个子线程来分，应该是第一个线程得到的 ticks 数目更多。而 nice-10 测试中 3000 个 ticks 由 10 个子线程来分，应该是第一个子线程得到的 ticks 最多，第十个 ticks 得到的 ticks 最少。预期的输出结果为：

```
16 (mlfqs-nice-2) begin
17 (mlfqs-nice-2) Starting 2 threads...
18 (mlfqs-nice-2) Starting threads took 8 ticks.
19 (mlfqs-nice-2) Sleeping 40 seconds to let threads run, please wait...
20 (mlfqs-nice-2) Thread 0 received 1925 ticks.
21 (mlfqs-nice-2) Thread 1 received 1077 ticks.
22 (mlfqs-nice-2) end
23
24 (mlfqs-nice-10) begin
25 (mlfqs-nice-10) Starting 10 threads...
26 (mlfqs-nice-10) Starting threads took 25 ticks.
27 (mlfqs-nice-10) Sleeping 40 seconds to let threads run, please wait...
28 (mlfqs-nice-10) Thread 0 received 684 ticks.
29 (mlfqs-nice-10) Thread 1 received 588 ticks.
30 (mlfqs-nice-10) Thread 2 received 492 ticks.
31 (mlfqs-nice-10) Thread 3 received 413 ticks.
32 (mlfqs-nice-10) Thread 4 received 320 ticks.
33 (mlfqs-nice-10) Thread 5 received 224 ticks.
34 (mlfqs-nice-10) Thread 6 received 145 ticks.
35 (mlfqs-nice-10) Thread 7 received 84 ticks.
36 (mlfqs-nice-10) Thread 8 received 40 ticks.
37 (mlfqs-nice-10) Thread 9 received 12 ticks.
38 (mlfqs-nice-10) end
```

## (2) 代码分析:

本次实验需要用到浮点数的计算, 由于 pintos 本身并没有支持浮点数计算, 因此我们需要自己实现一个浮点基本运算的体系。我的实现如下图所示:

```
4  typedef int fixed_t;
5  #define FP_SHIFT_AMOUNT 16
6
7  #define FP_CONST(A)      ((fixed_t)(A << FP_SHIFT_AMOUNT))
8  #define FP_ADD(A,B)      (A+B)
9  #define FP_ADD_MIX(A,B)  (A+(B<<FP_SHIFT_AMOUNT))
10 #define FP_SUB(A,B)      (A-B)
11 #define FP_SUB_MIX(A,B)  (A-(B<<FP_SHIFT_AMOUNT))
12 #define FP_MULT(A,B)     ((fixed_t)((((int64_t)A)*B >> FP_SHIFT_AMOUNT))
13 #define FP_MULT_MIX(A,B) (A*B)
14 #define FP_DIV(A,B)      ((fixed_t)((((int64_t)A) << FP_SHIFT_AMOUNT) / B))
15 #define FP_DIV_MIX(A,B)  (A / B)
16 #define FP_INT(A)        (A >> FP_SHIFT_AMOUNT)
17 #define FP_ROUND(A)      (A >= 0 ? ((A + (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT) :
18                          ((A - (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT))
19
```

我们首先定义了一个新的数据类型 `fixed_t`, 其本质仍然与 `int` 相同, 是一个 32 位的数。不过我们使用其低 16 位表示小数部分, 其 17-31 位表示整数部分, 第 32 位为符号位, 表示数的正负。因此我们对浮点数的四则运算要严格区分数据类型, 对于加减操作而言, `fixed_t` 与 `fixed_t` 可以直接相加减, `fixed_t` 与整数相加减时, 需要先把整数进行移位操作转换为 `fixed_t` 类型的数字才可以进行加减。而对于乘除操作而言, 恰恰相反, `fixed_t` 与 `fixed_t` 的相乘除需要先把 `fixed_t` 进行移位操作转换为整数类型的数字才可以进行加减, `fixed_t` 与整数相乘除时可以直接相乘除 (具体原因可以理解为 `fixed_t` 类型本来也是整数类型经过放大倍数后得到的, 而两个经过放大后的数字相乘会把倍数再次放大, 而导致错误)。在取整操作时, 主要是要理解到正数与负数四舍五入的不同, 因此在 `ROUND` 函数里需要先判断数字是正还是负。

首先我们需要对 `thread` 结构体做一些改动, 增加一些成员变量:

```
110  /*added for mlfqs */
111  int nice ;
112  fixed_t recent_cpu ;
113  /*added for mlfqs */
```

```
62  fixed_t load_avg;
```

并在 `thread.c` 文件中设置一个全局变量: 以上三个变量都是要用来计算多级反馈队列调度优先级的。接下来我们需要知道多级反馈队列调度的调节方式。每 1s (100 个 ticks) 时间更新一次系统 `load_avg` 和所有线程的 `recent_cpu`; 每 4 个 `timer_ticks` 更新一次线程优先级; 每个 `timer_tick` running 线程的 `recent_cpu` 加 1。因此我们首先需要实现更新这些变量的函数:

```
666 void
667 mlfqs_recent_cpu_add_one()
668 {
669     ASSERT (thread_mlfqs);
670     ASSERT (intr_context ());
671
672     struct thread *current_thread = thread_current ();
673     if (current_thread == idle_thread)
674         return;
675     current_thread->recent_cpu = FP_ADD_MIX (current_thread->recent_cpu, 1);
676 }
677
```

这是对当前正在 running 线程更新 `recent_cpu` 的函数。其原理比较简单, 就是先获取当前正在 running 线程的指针, 然后判断该线程的状态是否为 `idle`, 不是的话就可以确定该线程是在正常运行的, 因此我们对其 `recent_cpu` 进行加 1 操作。

```

677
678 void
679 mlfqs_load_avg_recent_cpu_update()
680 {
681     ASSERT (thread_mlfqs);
682     ASSERT (intr_context ());
683
684     struct thread *curr = thread_current() ;
685     size_t tot = list_size(&ready_list) ;
686     if(curr != idle_thread) tot++ ;
687
688     load_avg = FP_ADD(FP_DIV_MIX (FP_MULT_MIX (load_avg, 59), 60),
689 |FP_DIV_MIX (FP_CONST(tot), 60));
690
691     struct thread *t ;
692     struct list_elem * elem ;
693     for(elem = list_begin (&all_list) ; elem != list_end(&all_list) ; elem = list_next(elem))
694     {
695         t = list_entry(elem , struct thread , allelem) ;
696         if(t != idle_thread)
697         {
698             t->recent_cpu = FP_ADD_MIX (FP_MULT (FP_DIV (FP_MULT_MIX (load_avg, 2),
699 |FP_ADD_MIX (FP_MULT_MIX (load_avg, 2), 1)), t->recent_cpu), t->nice);
700         }
701     }
702 }

```

这里是更改 load\_avg 和 recent\_cpu 的函数，因为 load\_avg 是一个全局变量，因此对于 load\_avg 只需要根据给定的公式修改一次即可。而对于 recent\_cpu 而言，我们需要更新的是每一个线程的 recent\_cpu，这包括了正在 running 的线程、在就绪队列中线程、以及被 block 掉的线程。因此我们需要遍历 all\_list 中的每一个线程，对每一个线程都根据给定的公式做一次更新。

```

703
704 void
705 mlfqs_priority_update(struct thread* thrd)
706 {
707     if (thrd == idle_thread)
708         return;
709
710     ASSERT (thread_mlfqs);
711     ASSERT (thrd != idle_thread);
712
713     if(thrd != idle_thread)
714     {
715         thrd->priority = FP_INT (FP_SUB_MIX (FP_SUB (FP_CONST (PRI_MAX),
716 |FP_DIV_MIX (thrd->recent_cpu, 4)), 2 * thrd->nice));
717         if(thrd->priority > PRI_MAX)
718             thrd->priority = PRI_MAX ;
719         else if(thrd->priority < PRI_MIN)
720             thrd->priority = PRI_MIN ;
721     }
722 }

```

这里是更改单个线程优先级的函数，对于一个线程而言，我们需要利用当前系统的 load\_avg 值、线程自身的 recent\_cpu 和 nice 值来计算线程的优先级。之后我们需要进行一个判断，如果计算出来的优先级大于了 PRI\_MAX 的限制的话，则其优先级赋值为 PRI\_MAX，若计算出来的优先级小于 PRI\_MIN 的限制的话，则其优先级赋值为 PRI\_MIN。



```

724 void
725 mlfqs_priority_update_all()
726 {
727     struct thread *t ;
728     struct list_elem * elem ;
729     for(elem = list_begin (&all_list) ; elem != list_end(&all_list) ; elem = list_next(elem))
730     {
731         t = list_entry(elem , struct thread , allelem) ;
732         if(t != idle_thread)
733         {
734             mlfqs_priority_update (t);
735         }
736     }
737 }

```

这里是更改所有线程优先级的函数，与之前更新 recent\_cpu 相类似，我们同样是需要对于所有线程都更新优先级的。因此我们需要遍历 all\_list 中的每一个线程，对每一个线程均调用 mlfqs\_priority\_update 来更新其优先级。

在完成了上述几个关键函数的编写后，我们可以结合调节规则来合理地调用这些函数。由于这些调节都是以 ticks 为单位的，因此我们不难想到 timer.c 中的 timer\_interrupt 中断函数，我们可以在中断函数中来进行这些函数的调用。

```

173 static void
174 timer_interrupt (struct intr_frame *args UNUSED)
175 {
176     ticks++;
177     enum intr_level old = intr_disable() ;
178     thread_foreach(blocked_thread_check,NULL) ;
179     intr_set_level(old) ;
180     thread_tick () ;
181     if (thread_mlfqs)
182     {
183         mlfqs_recent_cpu_add_one() ;
184         if (ticks % TIMER_FREQ == 0)
185             mlfqs_load_avg_recent_cpu_update() ;
186         else if (ticks % 4 == 0)
187             mlfqs_priority_update_all() ;
188     }
189 }
190

```

以上是修改后的 timer\_interrupt() 函数，其主要的改动是在于在每一次中断中先对当前正在运行的线程的 recent\_cpu 执行加 1 操作，并进行判断，每满 100 个 ticks 就对所有的线程更新一次 load\_avg 与 recent\_cpu 数值，每过 4 个 ticks 则更新一次所有线程的优先级。在完成上述内容之后，我们本次的任务已经基本完成，接下来只需要填一下原来系统没有实现的几个函数：

```

375 void
376 thread_set_nice (int nice)
377 {
378     thread_current() -> nice = nice ;
379     mlfqs_priority_update(thread_current()) ;
380     thread_yield () ;
381 }
382
383 /* Returns the current thread's nice value. */
384 int
385 thread_get_nice (void)
386 {
387     /* Not yet implemented. */
388     return thread_current ()->nice;
389 }
390

```

```

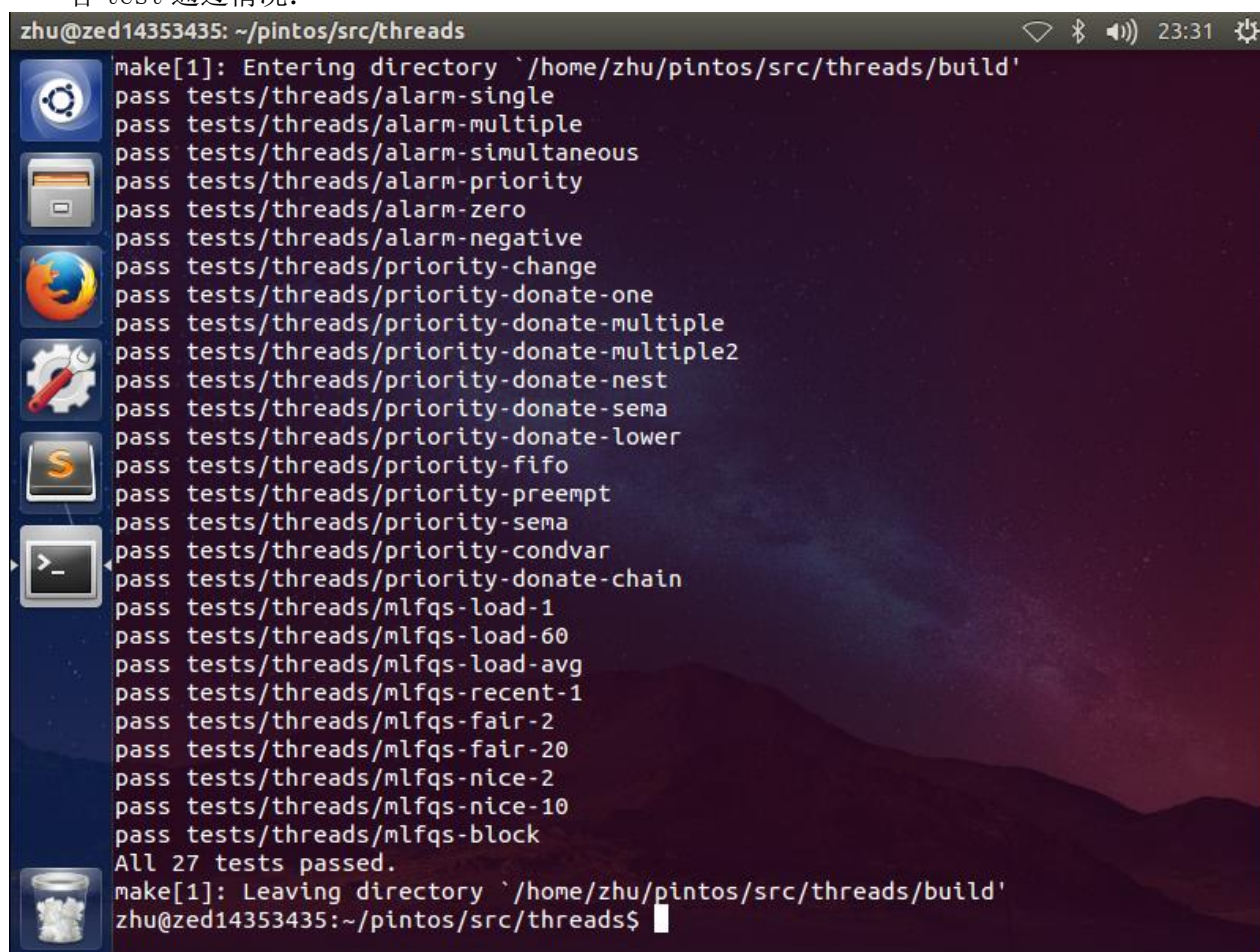
391  /* Returns 100 times the system load average. */
392  int
393  thread_get_load_avg (void)
394  {
395      /* Not yet implemented. */
396      return FP_ROUND (FP_MULT_MIX (load_avg, 100));
397  }
398
399  /* Returns 100 times the current thread's recent_cpu value. */
400  int
401  thread_get_recent_cpu (void)
402  {
403      /* Not yet implemented. */
404      return FP_ROUND (FP_MULT_MIX (thread_current ()->recent_cpu, 100));
405  }

```

这里需要注意的是，nice 值可以直接返回，但是 load\_avg 与 recent\_cpu 的值在返回时，为了与之后的测试相匹配，应该返回原始数值乘以 100 后的百分数结果。

### 3. 实验结果

各 test 通过情况：



```

zhu@zed14353435: ~/pintos/src/threads
make[1]: Entering directory `/home/zhu/pintos/src/threads/build'
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
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
make[1]: Leaving directory `/home/zhu/pintos/src/threads/build'
zhu@zed14353435:~/pintos/src/threads$

```

mlfqs-load-1 运行情况：

```
zhu@zed14353435: ~/pintos/src/threads
=====
000000000000i[      ] reading configuration from bochsrc.txt
000000000000e[      ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keybo
ard' option.
000000000000i[      ] installing nogui module as the Bochs GUI
000000000000i[      ] using log file bochsout.txt
Pilo hda1
Loading.....
Kernel command line: -mlfqs run mlfqs-load-1
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 'mlfqs-load-1':
(mlfqs-load-1) begin
(mlfqs-load-1) spinning for up to 45 seconds, please wait...
(mlfqs-load-1) load average rose to 0.5 after 41 seconds
(mlfqs-load-1) sleeping for another 10 seconds, please wait...
(mlfqs-load-1) load average fell back below 0.5 (to 0.43)
(mlfqs-load-1) PASS
(mlfqs-load-1) end
Execution of 'mlfqs-load-1' complete.

zhu@zed14353435:~/pintos/src/threads$ =====
=====
Bochs is exiting with the following message:
[      ] SIGNAL 2 caught
=====

zhu@zed14353435:~/pintos/src/threads$
```

mlfqs-load-60 运行情况:



```
(mlfqs-load-60) begin
(mlfqs-load-60) Starting 60 niced load threads...
(mlfqs-load-60) Starting threads took 1 seconds.
(mlfqs-load-60) After 0 seconds, load average=1.32.
(mlfqs-load-60) After 2 seconds, load average=3.26.
(mlfqs-load-60) After 4 seconds, load average=5.13.
(mlfqs-load-60) After 6 seconds, load average=6.95.
(mlfqs-load-60) After 8 seconds, load average=8.70.
(mlfqs-load-60) After 10 seconds, load average=10.40.
(mlfqs-load-60) After 12 seconds, load average=12.04.
(mlfqs-load-60) After 14 seconds, load average=13.62.
(mlfqs-load-60) After 16 seconds, load average=15.15.
(mlfqs-load-60) After 18 seconds, load average=16.64.
(mlfqs-load-60) After 20 seconds, load average=18.07.
(mlfqs-load-60) After 22 seconds, load average=19.46.
(mlfqs-load-60) After 24 seconds, load average=20.80.
(mlfqs-load-60) After 26 seconds, load average=22.09.
(mlfqs-load-60) After 28 seconds, load average=23.35.
(mlfqs-load-60) After 30 seconds, load average=24.56.
(mlfqs-load-60) After 32 seconds, load average=25.73.
(mlfqs-load-60) After 34 seconds, load average=26.86.
(mlfqs-load-60) After 36 seconds, load average=27.96.
(mlfqs-load-60) After 38 seconds, load average=29.02.
(mlfqs-load-60) After 40 seconds, load average=30.04.
(mlfqs-load-60) After 42 seconds, load average=31.03.
(mlfqs-load-60) After 44 seconds, load average=31.99.
(mlfqs-load-60) After 46 seconds, load average=32.91.
(mlfqs-load-60) After 48 seconds, load average=33.81.
(mlfqs-load-60) After 50 seconds, load average=34.68.
(mlfqs-load-60) After 52 seconds, load average=35.51.
(mlfqs-load-60) After 54 seconds, load average=36.32.
```

```
(mlfqs-load-60) After 58 seconds, load average=37.86.
(mlfqs-load-60) After 60 seconds, load average=38.59.
(mlfqs-load-60) After 62 seconds, load average=37.32.
(mlfqs-load-60) After 64 seconds, load average=36.08.
(mlfqs-load-60) After 66 seconds, load average=34.89.
(mlfqs-load-60) After 68 seconds, load average=33.74.
(mlfqs-load-60) After 70 seconds, load average=32.62.
(mlfqs-load-60) After 72 seconds, load average=31.54.
(mlfqs-load-60) After 74 seconds, load average=30.50.
(mlfqs-load-60) After 76 seconds, load average=29.49.
(mlfqs-load-60) After 78 seconds, load average=28.52.
(mlfqs-load-60) After 80 seconds, load average=27.58.
(mlfqs-load-60) After 82 seconds, load average=26.66.
(mlfqs-load-60) After 84 seconds, load average=25.78.
(mlfqs-load-60) After 86 seconds, load average=24.93.
(mlfqs-load-60) After 88 seconds, load average=24.11.
(mlfqs-load-60) After 90 seconds, load average=23.31.
(mlfqs-load-60) After 92 seconds, load average=22.54.
(mlfqs-load-60) After 94 seconds, load average=21.79.
(mlfqs-load-60) After 96 seconds, load average=21.07.
(mlfqs-load-60) After 98 seconds, load average=20.38.
(mlfqs-load-60) After 100 seconds, load average=19.70.
(mlfqs-load-60) After 102 seconds, load average=19.05.
(mlfqs-load-60) After 104 seconds, load average=18.42.
(mlfqs-load-60) After 106 seconds, load average=17.81.
(mlfqs-load-60) After 108 seconds, load average=17.22.
(mlfqs-load-60) After 110 seconds, load average=16.65.
(mlfqs-load-60) After 112 seconds, load average=16.10.
(mlfqs-load-60) After 114 seconds, load average=15.57.
(mlfqs-load-60) After 116 seconds, load average=15.06.
(mlfqs-load-60) After 118 seconds, load average=14.56.
```



```
zhu@zed14353435: ~/pintos/src/threads
(mlfqs-load-60) After 130 seconds, load average=12.01.
(mlfqs-load-60) After 132 seconds, load average=11.62.
(mlfqs-load-60) After 134 seconds, load average=11.23.
(mlfqs-load-60) After 136 seconds, load average=10.86.
(mlfqs-load-60) After 138 seconds, load average=10.50.
(mlfqs-load-60) After 140 seconds, load average=10.16.
(mlfqs-load-60) After 142 seconds, load average=9.82.
(mlfqs-load-60) After 144 seconds, load average=9.50.
(mlfqs-load-60) After 146 seconds, load average=9.18.
(mlfqs-load-60) After 148 seconds, load average=8.88.
(mlfqs-load-60) After 150 seconds, load average=8.58.
(mlfqs-load-60) After 152 seconds, load average=8.30.
(mlfqs-load-60) After 154 seconds, load average=8.03.
(mlfqs-load-60) After 156 seconds, load average=7.76.
(mlfqs-load-60) After 158 seconds, load average=7.50.
(mlfqs-load-60) After 160 seconds, load average=7.26.
(mlfqs-load-60) After 162 seconds, load average=7.02.
(mlfqs-load-60) After 164 seconds, load average=6.78.
(mlfqs-load-60) After 166 seconds, load average=6.56.
(mlfqs-load-60) After 168 seconds, load average=6.34.
(mlfqs-load-60) After 170 seconds, load average=6.13.
(mlfqs-load-60) After 172 seconds, load average=5.93.
(mlfqs-load-60) After 174 seconds, load average=5.73.
(mlfqs-load-60) After 176 seconds, load average=5.55.
(mlfqs-load-60) After 178 seconds, load average=5.36.
(mlfqs-load-60) end
Execution of 'mlfqs-load-60' complete.

zhu@zed14353435:~/pintos/src/threads$ =====
=====
```

mlfqs-load-avg 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'mlfqs-load-avg':
(mlfqs-load-avg) begin
(mlfqs-load-avg) Starting 60 load threads...
(mlfqs-load-avg) Starting threads took 1 seconds.
(mlfqs-load-avg) After 0 seconds, load average=0.06.
(mlfqs-load-avg) After 2 seconds, load average=0.11.
(mlfqs-load-avg) After 4 seconds, load average=0.22.
(mlfqs-load-avg) After 6 seconds, load average=0.39.
(mlfqs-load-avg) After 8 seconds, load average=0.63.
(mlfqs-load-avg) After 10 seconds, load average=0.92.
(mlfqs-load-avg) After 12 seconds, load average=1.27.
(mlfqs-load-avg) After 14 seconds, load average=1.67.
(mlfqs-load-avg) After 16 seconds, load average=2.13.
(mlfqs-load-avg) After 18 seconds, load average=2.64.
(mlfqs-load-avg) After 20 seconds, load average=3.20.
(mlfqs-load-avg) After 22 seconds, load average=3.80.
(mlfqs-load-avg) After 24 seconds, load average=4.45.
(mlfqs-load-avg) After 26 seconds, load average=5.15.
(mlfqs-load-avg) After 28 seconds, load average=5.89.
(mlfqs-load-avg) After 30 seconds, load average=6.67.
(mlfqs-load-avg) After 32 seconds, load average=7.49.
(mlfqs-load-avg) After 34 seconds, load average=8.35.
(mlfqs-load-avg) After 36 seconds, load average=9.25.
(mlfqs-load-avg) After 38 seconds, load average=10.18.
(mlfqs-load-avg) After 40 seconds, load average=11.15.
(mlfqs-load-avg) After 42 seconds, load average=12.15.
(mlfqs-load-avg) After 44 seconds, load average=13.19.
(mlfqs-load-avg) After 46 seconds, load average=14.26.
```



```
zhu@zed14353435: ~/pintos/src/threads 23:37
(mlfqs-load-avg) After 50 seconds, load average=16.49.
(mlfqs-load-avg) After 52 seconds, load average=17.64.
(mlfqs-load-avg) After 54 seconds, load average=18.83.
(mlfqs-load-avg) After 56 seconds, load average=20.04.
(mlfqs-load-avg) After 58 seconds, load average=21.28.
(mlfqs-load-avg) After 60 seconds, load average=22.54.
(mlfqs-load-avg) After 62 seconds, load average=23.78.
(mlfqs-load-avg) After 64 seconds, load average=24.90.
(mlfqs-load-avg) After 66 seconds, load average=25.92.
(mlfqs-load-avg) After 68 seconds, load average=26.85.
(mlfqs-load-avg) After 70 seconds, load average=27.65.
(mlfqs-load-avg) After 72 seconds, load average=28.36.
(mlfqs-load-avg) After 74 seconds, load average=28.99.
(mlfqs-load-avg) After 76 seconds, load average=29.53.
(mlfqs-load-avg) After 78 seconds, load average=29.98.
(mlfqs-load-avg) After 80 seconds, load average=30.33.
(mlfqs-load-avg) After 82 seconds, load average=30.60.
(mlfqs-load-avg) After 84 seconds, load average=30.81.
(mlfqs-load-avg) After 86 seconds, load average=30.95.
(mlfqs-load-avg) After 88 seconds, load average=31.05.
(mlfqs-load-avg) After 90 seconds, load average=31.05.
(mlfqs-load-avg) After 92 seconds, load average=30.96.
(mlfqs-load-avg) After 94 seconds, load average=30.85.
(mlfqs-load-avg) After 96 seconds, load average=30.64.
(mlfqs-load-avg) After 98 seconds, load average=30.37.
(mlfqs-load-avg) After 100 seconds, load average=30.04.
(mlfqs-load-avg) After 102 seconds, load average=29.66.
(mlfqs-load-avg) After 104 seconds, load average=29.26.
(mlfqs-load-avg) After 106 seconds, load average=28.77.
(mlfqs-load-avg) After 108 seconds, load average=28.23.
(mlfqs-load-avg) After 110 seconds, load average=27.64.

zhu@zed14353435: ~/pintos/src/threads 23:38
(mlfqs-load-avg) After 122 seconds, load average=23.26.
(mlfqs-load-avg) After 124 seconds, load average=22.49.
(mlfqs-load-avg) After 126 seconds, load average=21.75.
(mlfqs-load-avg) After 128 seconds, load average=21.03.
(mlfqs-load-avg) After 130 seconds, load average=20.34.
(mlfqs-load-avg) After 132 seconds, load average=19.66.
(mlfqs-load-avg) After 134 seconds, load average=19.01.
(mlfqs-load-avg) After 136 seconds, load average=18.38.
(mlfqs-load-avg) After 138 seconds, load average=17.78.
(mlfqs-load-avg) After 140 seconds, load average=17.19.
(mlfqs-load-avg) After 142 seconds, load average=16.62.
(mlfqs-load-avg) After 144 seconds, load average=16.07.
(mlfqs-load-avg) After 146 seconds, load average=15.54.
(mlfqs-load-avg) After 148 seconds, load average=15.03.
(mlfqs-load-avg) After 150 seconds, load average=14.53.
(mlfqs-load-avg) After 152 seconds, load average=14.05.
(mlfqs-load-avg) After 154 seconds, load average=13.59.
(mlfqs-load-avg) After 156 seconds, load average=13.14.
(mlfqs-load-avg) After 158 seconds, load average=12.70.
(mlfqs-load-avg) After 160 seconds, load average=12.28.
(mlfqs-load-avg) After 162 seconds, load average=11.88.
(mlfqs-load-avg) After 164 seconds, load average=11.48.
(mlfqs-load-avg) After 166 seconds, load average=11.10.
(mlfqs-load-avg) After 168 seconds, load average=10.74.
(mlfqs-load-avg) After 170 seconds, load average=10.38.
(mlfqs-load-avg) After 172 seconds, load average=10.04.
(mlfqs-load-avg) After 174 seconds, load average=9.71.
(mlfqs-load-avg) After 176 seconds, load average=9.39.
(mlfqs-load-avg) After 178 seconds, load average=9.08.
(mlfqs-load-avg) end
Execution of 'mlfqs-load-avg' complete.
```



mlfqs-recent-1 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
Executing 'mlfqs-recent-1':
(mlfqs-recent-1) begin
(mlfqs-recent-1) Sleeping 10 seconds to allow recent_cpu to decay, please wait...
(mlfqs-recent-1) After 2 seconds, recent_cpu is 7.39, load_avg is 0.03.
(mlfqs-recent-1) After 4 seconds, recent_cpu is 13.59, load_avg is 0.06.
(mlfqs-recent-1) After 6 seconds, recent_cpu is 19.60, load_avg is 0.10.
(mlfqs-recent-1) After 8 seconds, recent_cpu is 25.42, load_avg is 0.13.
(mlfqs-recent-1) After 10 seconds, recent_cpu is 31.06, load_avg is 0.15.
(mlfqs-recent-1) After 12 seconds, recent_cpu is 36.52, load_avg is 0.18.
(mlfqs-recent-1) After 14 seconds, recent_cpu is 41.80, load_avg is 0.21.
(mlfqs-recent-1) After 16 seconds, recent_cpu is 46.93, load_avg is 0.24.
(mlfqs-recent-1) After 18 seconds, recent_cpu is 51.89, load_avg is 0.26.
(mlfqs-recent-1) After 20 seconds, recent_cpu is 56.70, load_avg is 0.29.
(mlfqs-recent-1) After 22 seconds, recent_cpu is 61.35, load_avg is 0.31.
(mlfqs-recent-1) After 24 seconds, recent_cpu is 65.86, load_avg is 0.33.
(mlfqs-recent-1) After 26 seconds, recent_cpu is 70.22, load_avg is 0.35.
(mlfqs-recent-1) After 28 seconds, recent_cpu is 74.45, load_avg is 0.38.
(mlfqs-recent-1) After 30 seconds, recent_cpu is 78.54, load_avg is 0.40.
(mlfqs-recent-1) After 32 seconds, recent_cpu is 82.50, load_avg is 0.42.
(mlfqs-recent-1) After 34 seconds, recent_cpu is 86.33, load_avg is 0.43.
(mlfqs-recent-1) After 36 seconds, recent_cpu is 90.05, load_avg is 0.45.
(mlfqs-recent-1) After 38 seconds, recent_cpu is 93.64, load_avg is 0.47.
(mlfqs-recent-1) After 40 seconds, recent_cpu is 97.13, load_avg is 0.49.
(mlfqs-recent-1) After 42 seconds, recent_cpu is 100.49, load_avg is 0.51.
(mlfqs-recent-1) After 44 seconds, recent_cpu is 103.76, load_avg is 0.52.
(mlfqs-recent-1) After 46 seconds, recent_cpu is 106.92, load_avg is 0.54.
(mlfqs-recent-1) After 48 seconds, recent_cpu is 109.97, load_avg is 0.55.
(mlfqs-recent-1) After 50 seconds, recent_cpu is 112.93, load_avg is 0.57.
(mlfqs-recent-1) After 52 seconds, recent_cpu is 115.79, load_avg is 0.58.
(mlfqs-recent-1) After 54 seconds, recent_cpu is 118.57, load_avg is 0.60.
(mlfqs-recent-1) After 56 seconds, recent_cpu is 121.25, load_avg is 0.61.
```



```
zhu@zed14353435: ~/pintos/src/threads 23:39
(mlfqs-recent-1) After 60 seconds, recent_cpu is 126.36, load_avg is 0.63.
(mlfqs-recent-1) After 62 seconds, recent_cpu is 128.79, load_avg is 0.65.
(mlfqs-recent-1) After 64 seconds, recent_cpu is 131.15, load_avg is 0.66.
(mlfqs-recent-1) After 66 seconds, recent_cpu is 133.42, load_avg is 0.67.
(mlfqs-recent-1) After 68 seconds, recent_cpu is 135.63, load_avg is 0.68.
(mlfqs-recent-1) After 70 seconds, recent_cpu is 137.76, load_avg is 0.69.
(mlfqs-recent-1) After 72 seconds, recent_cpu is 139.83, load_avg is 0.70.
(mlfqs-recent-1) After 74 seconds, recent_cpu is 141.82, load_avg is 0.71.
(mlfqs-recent-1) After 76 seconds, recent_cpu is 143.76, load_avg is 0.72.
(mlfqs-recent-1) After 78 seconds, recent_cpu is 145.62, load_avg is 0.73.
(mlfqs-recent-1) After 80 seconds, recent_cpu is 147.43, load_avg is 0.74.
(mlfqs-recent-1) After 82 seconds, recent_cpu is 149.18, load_avg is 0.75.
(mlfqs-recent-1) After 84 seconds, recent_cpu is 150.87, load_avg is 0.76.
(mlfqs-recent-1) After 86 seconds, recent_cpu is 152.51, load_avg is 0.76.
(mlfqs-recent-1) After 88 seconds, recent_cpu is 154.10, load_avg is 0.77.
(mlfqs-recent-1) After 90 seconds, recent_cpu is 155.63, load_avg is 0.78.
(mlfqs-recent-1) After 92 seconds, recent_cpu is 157.12, load_avg is 0.79.
(mlfqs-recent-1) After 94 seconds, recent_cpu is 158.56, load_avg is 0.79.
(mlfqs-recent-1) After 96 seconds, recent_cpu is 159.94, load_avg is 0.80.
(mlfqs-recent-1) After 98 seconds, recent_cpu is 161.29, load_avg is 0.81.
(mlfqs-recent-1) After 100 seconds, recent_cpu is 162.58, load_avg is 0.81.
(mlfqs-recent-1) After 102 seconds, recent_cpu is 163.84, load_avg is 0.82.
(mlfqs-recent-1) After 104 seconds, recent_cpu is 165.06, load_avg is 0.83.
(mlfqs-recent-1) After 106 seconds, recent_cpu is 166.23, load_avg is 0.83.
(mlfqs-recent-1) After 108 seconds, recent_cpu is 167.37, load_avg is 0.84.
(mlfqs-recent-1) After 110 seconds, recent_cpu is 168.47, load_avg is 0.84.
(mlfqs-recent-1) After 112 seconds, recent_cpu is 169.54, load_avg is 0.85.
(mlfqs-recent-1) After 114 seconds, recent_cpu is 170.57, load_avg is 0.85.
(mlfqs-recent-1) After 116 seconds, recent_cpu is 171.57, load_avg is 0.86.
(mlfqs-recent-1) After 118 seconds, recent_cpu is 172.53, load_avg is 0.86.
(mlfqs-recent-1) After 120 seconds, recent_cpu is 173.46, load_avg is 0.87.

zhu@zed14353435: ~/pintos/src/threads 23:39
(mlfqs-recent-1) After 124 seconds, recent_cpu is 175.23, load_avg is 0.87.
(mlfqs-recent-1) After 126 seconds, recent_cpu is 176.08, load_avg is 0.88.
(mlfqs-recent-1) After 128 seconds, recent_cpu is 176.89, load_avg is 0.88.
(mlfqs-recent-1) After 130 seconds, recent_cpu is 177.68, load_avg is 0.89.
(mlfqs-recent-1) After 132 seconds, recent_cpu is 178.45, load_avg is 0.89.
(mlfqs-recent-1) After 134 seconds, recent_cpu is 179.19, load_avg is 0.89.
(mlfqs-recent-1) After 136 seconds, recent_cpu is 179.90, load_avg is 0.90.
(mlfqs-recent-1) After 138 seconds, recent_cpu is 180.59, load_avg is 0.90.
(mlfqs-recent-1) After 140 seconds, recent_cpu is 181.26, load_avg is 0.90.
(mlfqs-recent-1) After 142 seconds, recent_cpu is 181.91, load_avg is 0.91.
(mlfqs-recent-1) After 144 seconds, recent_cpu is 182.53, load_avg is 0.91.
(mlfqs-recent-1) After 146 seconds, recent_cpu is 183.13, load_avg is 0.91.
(mlfqs-recent-1) After 148 seconds, recent_cpu is 183.72, load_avg is 0.92.
(mlfqs-recent-1) After 150 seconds, recent_cpu is 184.28, load_avg is 0.92.
(mlfqs-recent-1) After 152 seconds, recent_cpu is 184.82, load_avg is 0.92.
(mlfqs-recent-1) After 154 seconds, recent_cpu is 185.35, load_avg is 0.92.
(mlfqs-recent-1) After 156 seconds, recent_cpu is 185.87, load_avg is 0.93.
(mlfqs-recent-1) After 158 seconds, recent_cpu is 186.36, load_avg is 0.93.
(mlfqs-recent-1) After 160 seconds, recent_cpu is 186.84, load_avg is 0.93.
(mlfqs-recent-1) After 162 seconds, recent_cpu is 187.30, load_avg is 0.93.
(mlfqs-recent-1) After 164 seconds, recent_cpu is 187.75, load_avg is 0.94.
(mlfqs-recent-1) After 166 seconds, recent_cpu is 188.18, load_avg is 0.94.
(mlfqs-recent-1) After 168 seconds, recent_cpu is 188.59, load_avg is 0.94.
(mlfqs-recent-1) After 170 seconds, recent_cpu is 189.00, load_avg is 0.94.
(mlfqs-recent-1) After 172 seconds, recent_cpu is 189.39, load_avg is 0.94.
(mlfqs-recent-1) After 174 seconds, recent_cpu is 189.77, load_avg is 0.95.
(mlfqs-recent-1) After 176 seconds, recent_cpu is 190.14, load_avg is 0.95.
(mlfqs-recent-1) After 178 seconds, recent_cpu is 190.49, load_avg is 0.95.
(mlfqs-recent-1) After 180 seconds, recent_cpu is 190.84, load_avg is 0.95.
(mlfqs-recent-1) end
Execution of 'mlfqs-recent-1' complete.
```

mlfqs-nice-1 运行情况:



```
zhu@zed14353435: ~/pintos/src/threads
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: -mlfqs run mlfqs-nice-2
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 'mlfqs-nice-2':
(mlfqs-nice-2) begin
(mlfqs-nice-2) Starting 2 threads...
(mlfqs-nice-2) Starting threads took 8 ticks.
(mlfqs-nice-2) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-nice-2) Thread 0 received 1925 ticks.
(mlfqs-nice-2) Thread 1 received 1077 ticks.
(mlfqs-nice-2) end
Execution of 'mlfqs-nice-2' complete.

zhu@zed14353435:~/pintos/src/threads$ =====
Bochs is exiting with the following message:
[      ] SIGNAL 2 caught
=====
```

mlfqs-nice-10 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads 23:41
ard' option.
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: -mlfqs run mlfqs-nice-10
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 'mlfqs-nice-10':
(mlfqs-nice-10) begin
(mlfqs-nice-10) Starting 10 threads...
(mlfqs-nice-10) Starting threads took 25 ticks.
(mlfqs-nice-10) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-nice-10) Thread 0 received 685 ticks.
(mlfqs-nice-10) Thread 1 received 589 ticks.
(mlfqs-nice-10) Thread 2 received 493 ticks.
(mlfqs-nice-10) Thread 3 received 412 ticks.
(mlfqs-nice-10) Thread 4 received 321 ticks.
(mlfqs-nice-10) Thread 5 received 225 ticks.
(mlfqs-nice-10) Thread 6 received 144 ticks.
(mlfqs-nice-10) Thread 7 received 85 ticks.
(mlfqs-nice-10) Thread 8 received 40 ticks.
(mlfqs-nice-10) Thread 9 received 12 ticks.
(mlfqs-nice-10) end
Execution of 'mlfqs-nice-10' complete.

zhu@zed14353435:~/pintos/src/threads$ =====
=====
```

mlfqs-fair-2 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads 23:42
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: -mlfqs run mlfqs-fair-2
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 'mlfqs-fair-2':
(mlfqs-fair-2) begin
(mlfqs-fair-2) Starting 2 threads...
(mlfqs-fair-2) Starting threads took 8 ticks.
(mlfqs-fair-2) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-fair-2) Thread 0 received 1496 ticks.
(mlfqs-fair-2) Thread 1 received 1504 ticks.
(mlfqs-fair-2) end
Execution of 'mlfqs-fair-2' complete.

zhu@zed14353435:~/pintos/src/threads$ =====
=====
Bochs is exiting with the following message:
[      ] SIGNAL 2 caught
=====
```



mlfqs-fair-20 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
383 pages available in user pool.
Calibrating timer... 204,600 loops/s.
Boot complete.
Executing 'mlfqs-fair-20':
(mlfqs-fair-20) begin
(mlfqs-fair-20) Starting 20 threads...
(mlfqs-fair-20) Starting threads took 49 ticks.
(mlfqs-fair-20) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-fair-20) Thread 0 received 148 ticks.
(mlfqs-fair-20) Thread 1 received 149 ticks.
(mlfqs-fair-20) Thread 2 received 152 ticks.
(mlfqs-fair-20) Thread 3 received 148 ticks.
(mlfqs-fair-20) Thread 4 received 152 ticks.
(mlfqs-fair-20) Thread 5 received 153 ticks.
(mlfqs-fair-20) Thread 6 received 153 ticks.
(mlfqs-fair-20) Thread 7 received 151 ticks.
(mlfqs-fair-20) Thread 8 received 149 ticks.
(mlfqs-fair-20) Thread 9 received 148 ticks.
(mlfqs-fair-20) Thread 10 received 149 ticks.
(mlfqs-fair-20) Thread 11 received 149 ticks.
(mlfqs-fair-20) Thread 12 received 148 ticks.
(mlfqs-fair-20) Thread 13 received 149 ticks.
(mlfqs-fair-20) Thread 14 received 157 ticks.
(mlfqs-fair-20) Thread 15 received 153 ticks.
(mlfqs-fair-20) Thread 16 received 152 ticks.
(mlfqs-fair-20) Thread 17 received 153 ticks.
(mlfqs-fair-20) Thread 18 received 149 ticks.
(mlfqs-fair-20) Thread 19 received 148 ticks.
(mlfqs-fair-20) end
Execution of 'mlfqs-fair-20' complete.
```

mlfqs-block 运行情况:

```
zhu@zed14353435: ~/pintos/src/threads
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: -mlfqs run mlfqs-block
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 'mlfqs-block':
(mlfqs-block) begin
(mlfqs-block) Main thread acquiring lock.
(mlfqs-block) Main thread creating block thread, sleeping 25 seconds...
(mlfqs-block) Block thread spinning for 20 seconds...
(mlfqs-block) Block thread acquiring lock...
(mlfqs-block) Main thread spinning for 5 seconds...
(mlfqs-block) Main thread releasing lock.
(mlfqs-block) ...got it.
(mlfqs-block) Block thread should have already acquired lock.
(mlfqs-block) end
Execution of 'mlfqs-block' complete.

zhu@zed14353435:~/pintos/src/threads$ =====
=====
```

## 4. 回答问题

### (1) 信号量，锁，条件变量的异同点：

首先信号量、锁以及条件变量都是我们为了解决同步问题而引入的概念。我们要清楚，锁其实是一种特殊的信号量，锁是一种二值信号量，只存在被占有与可占有这两种状态。而条件变量更像是一个通知者的角色，它的组成只有一个等待队列，因此它的操作也只有挂起和唤醒这两种状态。但是我们可以通过条件变量来解决仅仅依靠互斥锁难以解决的一些问题，比如说现在有两个线程 a 和 b，a 线程是对一个全局变量 sum 做加 1 操作，b 线程是对全局变量 sum 做减 1 操作，这时我们使用互斥锁就足够解决这两者的同步问题了。但是，倘若我们再增加一个线程 c，它在全局变量 sum=100 的时候输出一个提示信息，这时我们仅仅依靠一个互斥锁就无法完成了，因为线程 c 涉及到了对 sum 状态的判断。这时加入我们引入一个条件变量，在 a 线程和 b 线程每次改变 sum 的值之后进行判断是否达到 100，若达到 100 就通过条件变量来通知线程 c 唤醒，开始执行。这样可以大大降低我们编程的复杂度。

### (2) 多级反馈调度为什么不需要优先级捐赠：

因为优先级捐赠是发生在优先级调度中的一个操作，而多级反馈队列调度与优先级调度本身是属于两种完全不同的调度方式。在优先级调度中，每一线程的优先级是静态的，也即不会随着线程的执行而发生改变。因此，假如说有一个优先级较低的线程在获取锁之后，又处于等待队列中，就很容易发生优先级较高的线程被优先级较低的线程 block 的情况。为了解决这一问题，我们才引入了优先级捐赠的机制，以来保证获取锁的线程可以优先得到执行。然而，在多级反馈调度中，线程的优先级是出于动态调节中的。一般而言，线程的优先级是会随着线程的执行而降低的，而出于等待中的线程的优先级则会随着等待而升高。因此即使一个优先级较低的线程在获取锁之后，也会在等待队列中不断提升他的优先级。所以多级反馈调度不需要优先级捐赠，假如有了优先级捐赠，反而会破坏我们调节的机制。

### (3) 多级反馈调度为什么能避免饥饿现象：

之前仅仅依靠优先级调度的方式容易造成饥饿现象，即若有一个优先级很高且需要运行很久的线程 A 被创建之后，A 可能一直占着 CPU，而其他较低优先级的线程可能运行时间较短，但是需要等待 A 的执行完毕。而多级反馈队列调度是会综合考虑 CPU 的吞吐量(load\_avg)，根据线程的属性(nice 值)，线程占用 CPU 的时间(recent\_cpu)等线程运转情况来动态调整线程优先级的。一般来说，一个线程占有 cpu 越久，它的优先级就会不断降低；一个线程等待时间越久，它的优先级就会不断升高。因此，整个系统的响应速率是提高了的，也就不会再出现一个线程长时间占用 cpu 而导致其他线程饿死的状态出现了。

## 5. 实验感想

本次实验是关于条件变量的操作与多级反馈队列调度机制的实现。本次实验在原理上稍显复杂，其基本思想已经在之前讨论的比较详细了。所以在这部分我主要说一下在本次实验中遇到的一些细节问题。

首先是本次实验中一开始在对于浮点数的运算上理解出了一点偏差，尤其是在对于浮点数的乘除运算上。浮点数的乘除运算与加减运算稍有不同，对于加减操作而言，fixed\_t 与 fixed\_t 类型的数字可以直接相加减，fixed\_t 类型与整数相加减时，需要先把整数进行移位操作转换为 fixed\_t 类型的数字才可以进行加减。而对于乘除操作而言，恰恰相反，fixed\_t 与 fixed\_t 类型的相乘除

需要先把 `fixed_t` 进行移位操作转换为整数类型的数字才可以进行加减，`fixed_t` 与整数相乘除时可以直接相乘除。这是因为我们 `fixed_t` 的本质仍然是一个 32 位的 `int` 型，我们将其左移 16 位，其实相当于是把原本的 `int` 型数据放大了  $2^{16}$  倍。所以当两个 `fixed_t` 类型的数据相乘时，这个放大倍数会变成  $(2^{16})^2=2^{32}$ ，因此我们需要先用一个 `int_64` 类型的整数确保可以保存下计算的结果。然后，我们再将计算结果左移 16 位，相当于把放大倍数还原，这样才可以得到正确的计算结果。除法的原理完全相同，只不过两个 `fixed_t` 相除会把放大倍数消除掉，因此我们需要将结果右移 16 位。

同时，在本次实验中我学会使用了 `pintos` 中的 `debug` 工具。由于本次实验刚写完的时候，运行总是出现断言错误，但是又不知道是在什么地方调用了断言所处的函数，因此很难去发现问题出现的原因。后来我在网上阅读 `pintos` 的说明文档时发现 `pintos` 是有内置的 `debug` 函数的，其实定义和实现分别在 `debug.h` 和 `debug.c` 中。比如我这次 `debug_backtrace_all()` 和 `debug_backtrace()` 这两个函数，分别是用来追踪函数调用栈的，前者是输出当前线程和就绪队列中的所有线程的调用栈，后者是输出当前运行线程的调用栈。这两个函数可以放在代码中的任何地方，一旦执行到这两个函数，就输出之前所有的调用栈。比如我这次本来总是在 `list_insert` 中的一个断言出错，所以我在这个断言之前插入一个 `debug_backtrace()` 函数，输出之前的调用栈。但是输出的调用栈是 16 进制的机器码，很难读懂，因此可以使用 `pintos` 指令中的 `backtrace xxx` 指令，将机器码转换成对应的代码行。这样就可逐层地查看每一级函数调用，从而发现自己在哪个函数出错。比如本次我就是通过 `debug_backtrace` 发现了在链表初始化时的一个错误。

本次实验还有一点是关于 `mlfqs` 模式与优先级捐赠调度的冲突的问题。由于之前我们已经说过，多级反馈队列调节是不需要优先级捐赠的，如果有了优先级捐赠，反而会得到错误的结果。因此我们需要将上一次实现的优先级捐赠部分进行一点改动，加上 `(!mlfqs)` 的判断，确保优先级捐赠的内容仅仅在非 `mlfqs` 调度的模式下才会运行。