# 中山大学移动信息工程学院本科生实验报告

（2017 学年春季学期）

课程名称：Operating System　　　　任课教师：**饶洋辉**　　　　**批改人(此处为 TA 填写)：**

| | | | |
|---|---|---|---|
| 年级+班级 | 1501 | 专业（方向） | **移动信息工程** |
| 学号 | 15352006 | 姓名 | **蔡丽芝** |
| 电话 | 13538489980 | Email | 314749816@qq.com |
| 开始日期 | 2017/5/22 | 完成日期 | 2017/5/25 |

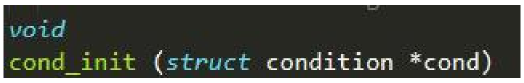
## 1. 实验目的

1） A 理解条件变量的含义并通过 priority-cond 测试
2） B 实现多级反馈队列的调度并通过所有的 mlfqs-*测试

## 2. 实验过程

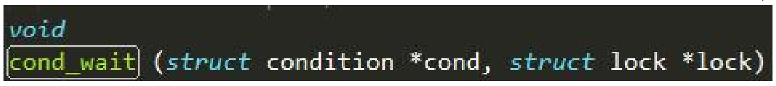
(一)Test 分析（30 分）

实验 test 涉及到的相关函数的作用：



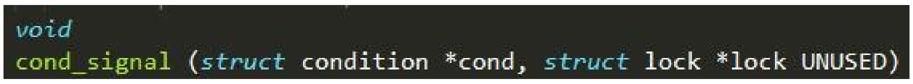
```
void
cond_init (struct condition *cond)
```

初始化状态变量 cond，将 cond 的 waiters 链表初始化为空，表明没有线程被 cond 阻塞.



```
void
cond_wait (struct condition *cond, struct lock *lock)
```

当满足等待为真的时候，线程调用 cond_wait 函数，会先释放占有的锁，并且将线程对应的信号量插入到状态变量 cond 的 waiters 链表中，接着调用 sema_down 函数阻塞掉自己。直到能被唤醒后，才重新申请获得锁。

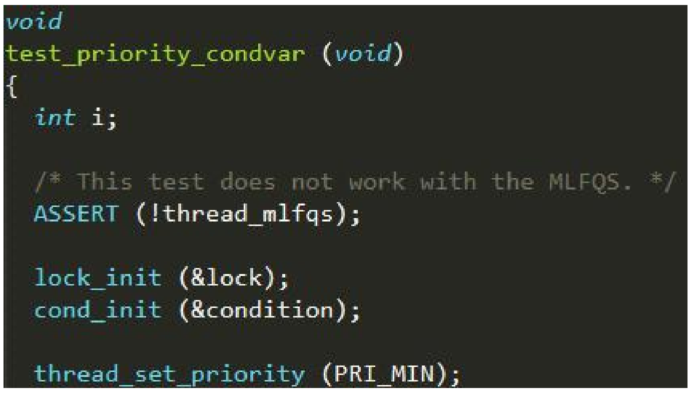


```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
```

当另外的占有锁的线程满足使得条件为真时，线程会调用 cond_signal 函数，唤醒其中一个被 cond 所阻塞得线程(实现后，应该为优先级最高的线程)

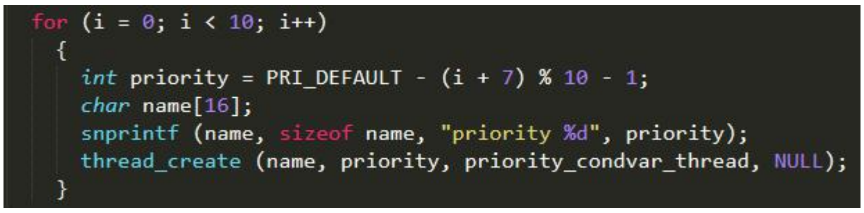
**Test 1**: priority-condvar

测试目的：测试被 condition 阻塞线程唤醒时的顺序是否是按优先级大小有序进行的。



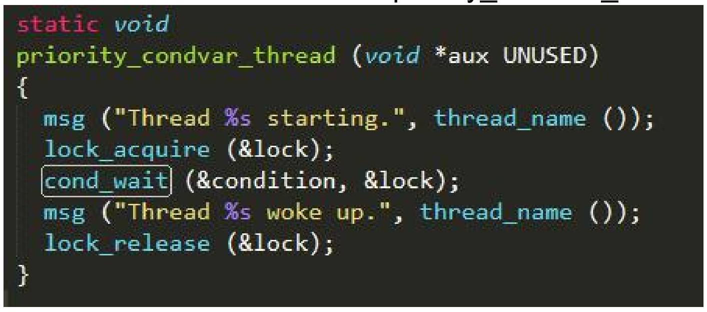
```
void
test_priority_condvar (void)
{
  int i;

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

  lock_init (&lock);
  cond_init (&condition);

  thread_set_priority (PRI_MIN);
```

分别调用 lock_init 函数，cond_init 函数初始化锁 lock 和状态量 condition，调用 thread_set_priority 函数将测试线程的优先级设为 0。测试线程的优先级最低。



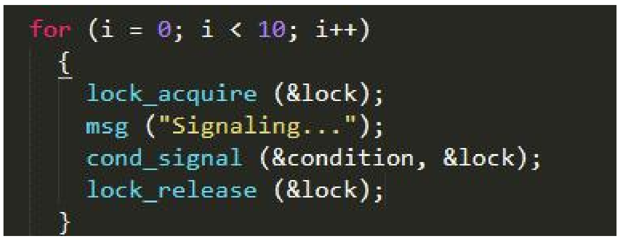
```c
  for (i = 0; i < 10; i++)
    {
      int priority = PRI_DEFAULT - (i + 7) % 10 - 1;
      char name[16];
      snprintf (name, sizeof name, "priority %d", priority);
      thread_create (name, priority, priority_condvar_thread, NULL);
    }
```

进入循环，第一次循环，调用 thread_create 函数创建优先级为 23 的线程 priority 23，优先级高于测试线程，抢占调度 priority_condvar_thread 函数



```c
static void
priority_condvar_thread (void *aux UNUSED)
{
  msg ("Thread %s starting.", thread_name ());
  lock_acquire (&lock);
  cond_wait (&condition, &lock);
  msg ("Thread %s woke up.", thread_name ());
  lock_release (&lock);
}
```

输出信息 msg（Thread priority 23 starting)，调用 lock_acquire 函数获得锁 lock，调用 cond_wait 函数，当前线程 priority 23 释放掉当前的锁，并且被 condition 阻塞。测试线程获得 cpu，在之前中断的地方开始执行，进入第二次循环，创建线程 priority 22,同理进行优先级抢占调用 priority_condvar_thread 函数，依旧被 condition 阻塞。测试线程再次获得 cpu，进行与上述同样的操作过程，直到创建完第 10 个线程，此时在 condition 的 waiters 链表中是由 10 个之前被创建的线程所组成。

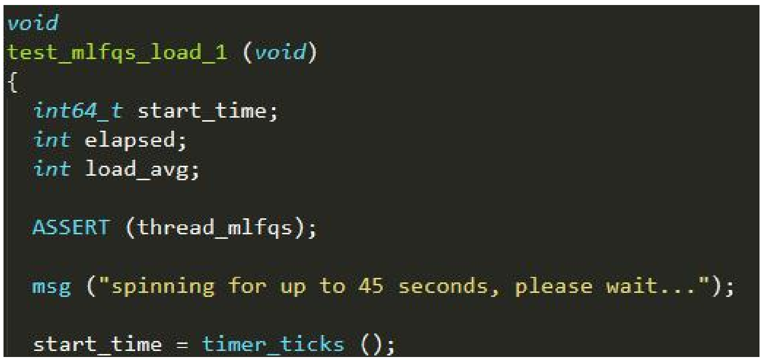


```c
  for (i = 0; i < 10; i++)
    {
      lock_acquire (&lock);
      msg ("Signaling...");
      cond_signal (&condition, &lock);
      lock_release (&lock);
    }
```

测试线程获得 cpu，再次进入循环，第一次循环，调用 lock_acquire 函数获得锁(当前线程能调用 cond_signal 函数唤醒线程的前提条件)，输出 msg 信息。调用 cond_signal 函数唤醒被 condition 所阻塞的线程，又因为 condition 的 waiters 链表是按照优先级大小排序，因此唤醒的是该链表中优先级最高的线程，即 priority 30，调用 lock_release 函数释放锁 lock，priority 30 获得 cpu，输出 msg(“Thread priority 30 woke up”)，释放锁，priority 30 线程执行完成。测试线程获得 cpu，重复上述操作，直到 10 个线程被阻塞在 condition 的线程唤醒并执行完毕。(这里 10 个线程唤醒执行的顺序是按照优先级大小从大到小执行)。

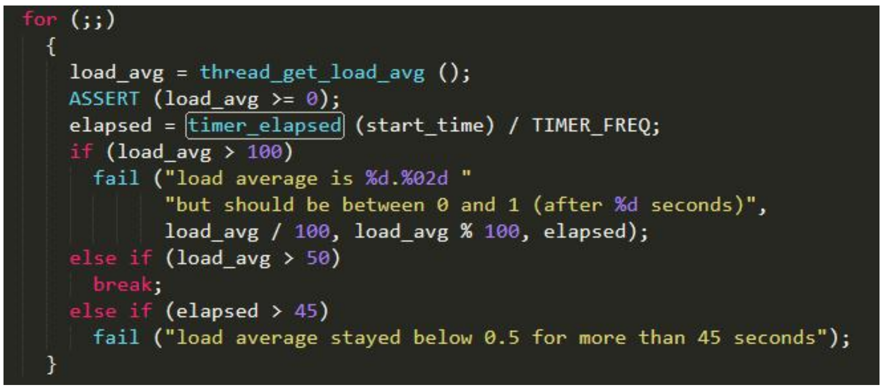
## Test 2：mlfqs-load-1

测试目的： 测试是否是在 38 至 45s 内，一个繁忙的线程将 cpu 平均负载升到 0.5 以上的，但不超过 1.



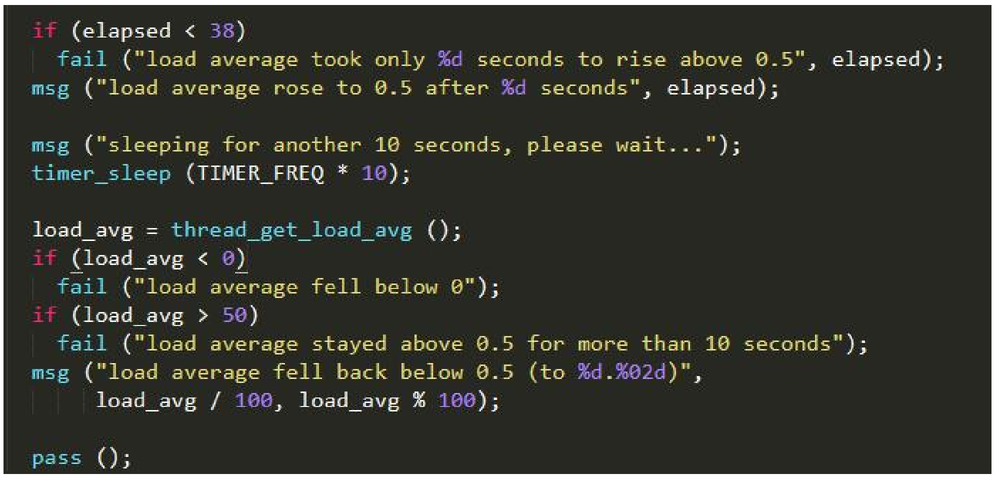
```c
void
test_mlfqs_load_1 (void)
{
  int64_t start_time;
  int elapsed;
  int load_avg;

  ASSERT (thread_mlfqs);

  msg ("spinning for up to 45 seconds, please wait...");

  start_time = timer_ticks ();
```

声明定义 test 后面所需变量，start_time 记录线程开始时间。



```c
for (;;)
  {
    load_avg = thread_get_load_avg ();
    ASSERT (load_avg >= 0);
    elapsed = timer_elapsed (start_time) / TIMER_FREQ;
    if (load_avg > 100)
      fail ("load average is %d.%02d "
            "but should be between 0 and 1 (after %d seconds)",
            load_avg / 100, load_avg % 100, elapsed);
    else if (load_avg > 50)
      break;
    else if (elapsed > 45)
      fail ("load average stayed below 0.5 for more than 45 seconds");
  }
```

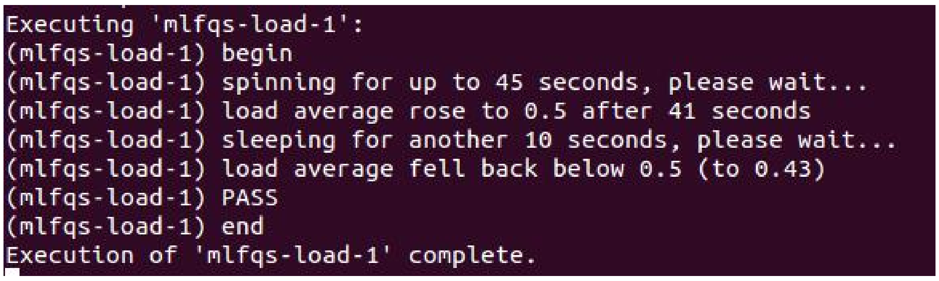
Load_avg 记录 cpu 的平均负载时间，elapsed 记录流逝的时间，除以 TIMER_FREQ，使 elapsed 的单位变成秒。当 load_avg>100 时，执行 fail 语句，终止该线程并且输出相应的信息。当 50<load_avg<=100 时，跳出 for 循环。当 elapsed 流逝的时间大于 45s 时，执行 fail 语句终止该线程，表明该线程需要超过 45s 的时间才能将 cpu 平均负载升到所应达到的值。
因此：能跳出该 for 循环，表明线程在 45s 内使得 cpu 平均负载上升到 0.5 以上，但不超过 1.



```c
  if (elapsed < 38)
    fail ("load average took only %d seconds to rise above 0.5", elapsed);
  msg ("load average rose to 0.5 after %d seconds", elapsed);

  msg ("sleeping for another 10 seconds, please wait...");
  timer_sleep (TIMER_FREQ * 10);

  load_avg = thread_get_load_avg ();
  if (load_avg < 0)
    fail ("load average fell below 0");
  if (load_avg > 50)
    fail ("load average stayed above 0.5 for more than 10 seconds");
  msg ("load average fell back below 0.5 (to %d.%02d)",
       load_avg / 100, load_avg % 100);

  pass ();
```

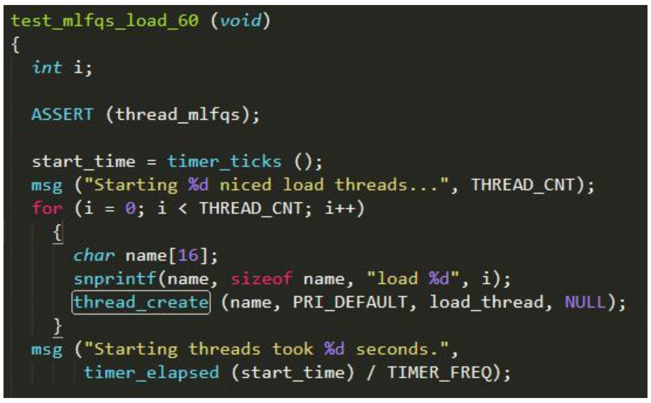
当 elapsed 流逝时间小于 38，执行 fail 语句，输出失败信息，不符要求。否则，不满足上面的情况，表明流逝时间是在 38s 和 45s 之间。就可以输出 msg 里的信息。
调用 timer_sleep 函数休眠 10s，10s 后线程休眠结束，当 load_avg 小于 0 时，执行 fail 语句并输出信息终止线程。当 load_avg 大于 50，同样会执行 fail 语句结束线程。在线程休眠的 10s 内，由于 load_avg 是一个全局变量，因此在线程休眠的这段时间内，线程进入等待队列，load_avg 的值会减少但是由于就绪队列中仍然有线程，因此不会减少到 0。所以上述的两个判断条件都不符和，线程不会执行上述两个条件里的内容，最后执行 pass 函数。

因此运行的正确结果为：



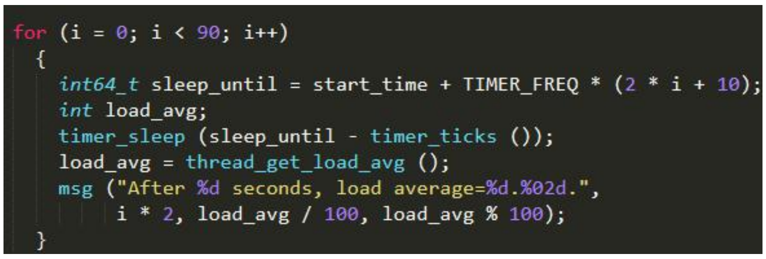
```
Executing 'mlfqs-load-1':
(mlfqs-load-1) begin
(mlfqs-load-1) spinning for up to 45 seconds, please wait...
(mlfqs-load-1) load average rose to 0.5 after 41 seconds
(mlfqs-load-1) sleeping for another 10 seconds, please wait...
(mlfqs-load-1) load average fell back below 0.5 (to 0.43)
(mlfqs-load-1)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

**Test 3：**mlfqs_load_60
测试目的：测试当多个线程运行时，load_avg 的变化



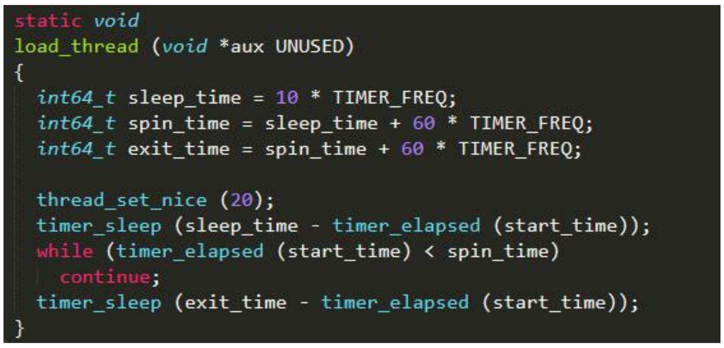
```
test_mlfqs_load_60 (void)
{
  int i;

  ASSERT (thread_mlfqs);

  start_time = timer_ticks ();
  msg ("Starting %d niced load threads...", THREAD_CNT);
  for (i = 0; i < THREAD_CNT; i++)
    {
      char name[16];
      snprintf(name, sizeof name, "load %d", i);
      thread_create (name, PRI_DEFAULT, load_thread, NULL);
    }
  msg ("Starting threads took %d seconds.",
       timer_elapsed (start_time) / TIMER_FREQ);
```

主线程进入一个 for 循环，依次创建 60 个优先级均为 31 的线程。



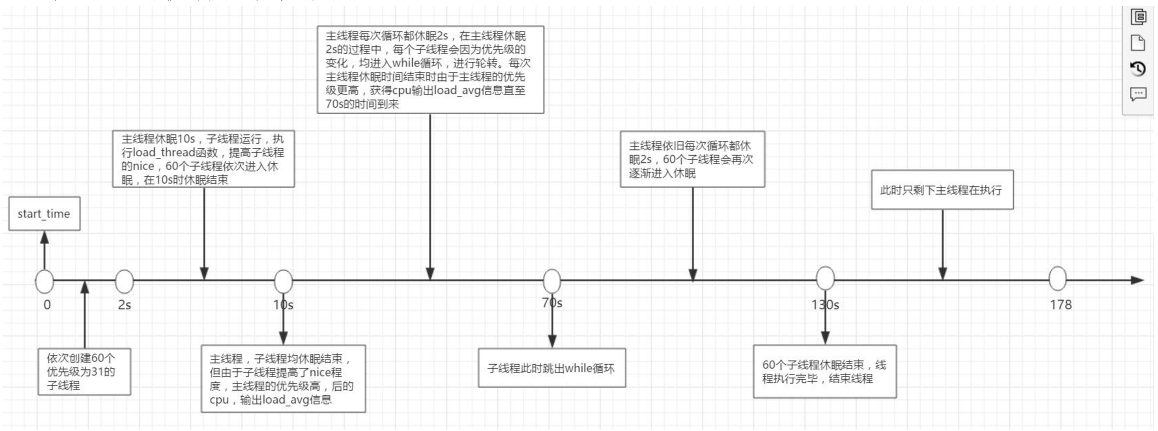
```
for (i = 0; i < 90; i++)
  {
    int64_t sleep_until = start_time + TIMER_FREQ * (2 * i + 10);
    int load_avg;
    timer_sleep (sleep_until - timer_ticks ());
    load_avg = thread_get_load_avg ();
    msg ("After %d seconds, load average=%d.%02d.",
         i * 2, load_avg / 100, load_avg % 100);
  }
```

主线程又进入一个 for 循环，第一次进入循环时，调用 timer_sleep 函数休眠 10s，此时前面创建的线程会获得 cpu 执行 load_thread 函数，10s 后主线程休眠结束，记录 cpu 的平均负载至 load_avg，调用 msg 输出(After 0 seconds，load_avg 值)。之后的每一次进入循环时，主线程休眠的时间均为 2s，同样当休眠结束后，输出 load_avg



```
static void
load_thread (void *aux UNUSED)
{
  int64_t sleep_time = 10 * TIMER_FREQ;
  int64_t spin_time = sleep_time + 60 * TIMER_FREQ;
  int64_t exit_time = spin_time + 60 * TIMER_FREQ;

  thread_set_nice (20);
  timer_sleep (sleep_time - timer_elapsed (start_time));
  while (timer_elapsed (start_time) < spin_time)
    continue;
  timer_sleep (exit_time - timer_elapsed (start_time));
}
```

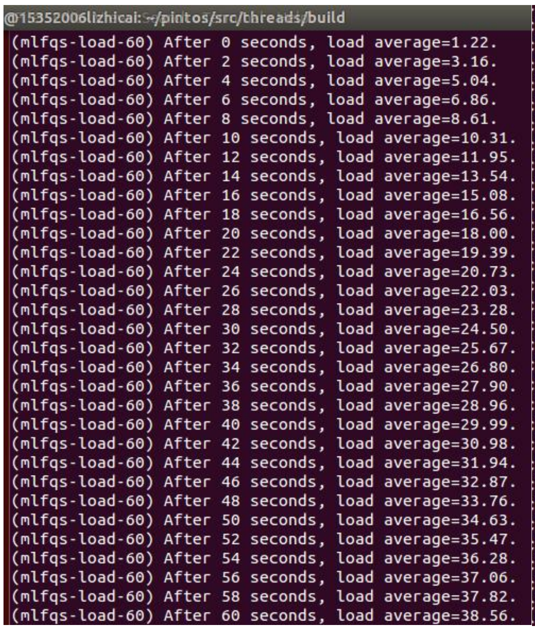
整个 test 的执行流程如图：



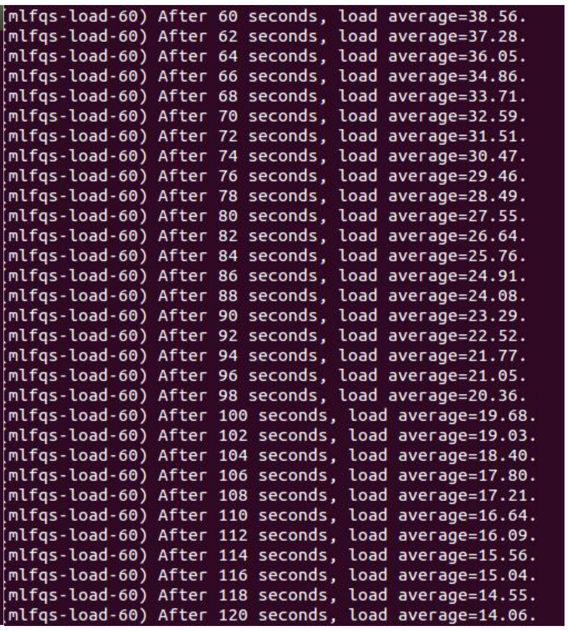
主线程每次循环都休眠2s，在主线程休眠2s的过程中，每个子线程会因为优先级的变化，均进入while循环，进行轮转。每次主线程休眠时间结束时由于主线程的优先级更高，获得cpu输出load_avg信息直至70s的时间到来
主线程休眠10s，子线程运行，执行load_thread函数，提高子线程的nice，60个子线程依次进入休眠，在10s时休眠结束
start_time
主线程依旧每次循环都休眠2s，60个子线程会再次逐渐进入休眠
此时只剩下主线程在执行
0
2s
10s
70s
130s
178
依次创建60个优先级为31的子线程
主线程，子线程均休眠结束，但由于子线程提高了nice程度，主线程的优先级高，后的cpu，输出load_avg信息
子线程此时跳出while循环
60个子线程休眠结束，线程执行完毕，结束线程

分析：在主线程输出信息时，对应的时间段也就是时间轴上的 10s~178s,在 10s~70s(对应输出结果的 after 0 至 afrer 60s)，此阶段，子线程不断地进入运行队列执行 while 循环，根据 Load_avg 地计算公式可知，load_avg=59/60*load_avg+1/60*ready_threads, ready_threads = 60，因此 1/60*ready_threads > 1，所以 load_avg 会不断增大。当在 70s~130s(60s~120s)，60 个线程会很快地进入休眠，此时 read_threads = 1,只剩下主线程根据公式，明显 load_avg 会不断减小。当 130s~178s(120s~)，此时只剩下主线程在就绪队列和等待队列中来回切换，因此依旧会减小，但是因为 load_avg 反映的是在过去 1

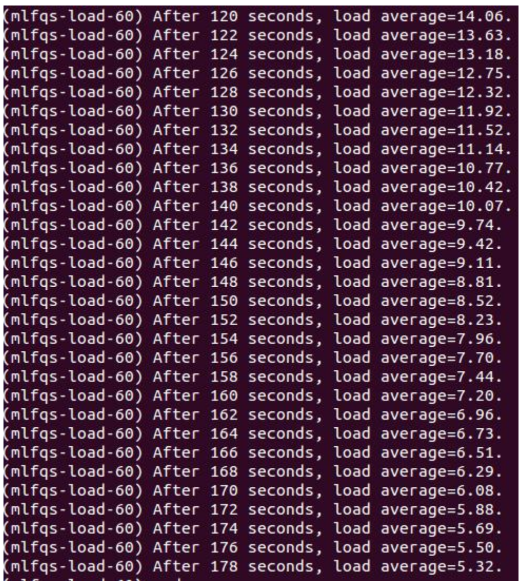
分钟 cpu 的平均负载，所以 load_avg 不会急速下降至 0。
所以运行结果：



```
@15352006lizhicai: ~/pintos/src/threads/build
(mlfqs-load-60) After 0 seconds, load average=1.22.
(mlfqs-load-60) After 2 seconds, load average=3.16.
(mlfqs-load-60) After 4 seconds, load average=5.04.
(mlfqs-load-60) After 6 seconds, load average=6.86.
(mlfqs-load-60) After 8 seconds, load average=8.61.
(mlfqs-load-60) After 10 seconds, load average=10.31.
(mlfqs-load-60) After 12 seconds, load average=11.95.
(mlfqs-load-60) After 14 seconds, load average=13.54.
(mlfqs-load-60) After 16 seconds, load average=15.08.
(mlfqs-load-60) After 18 seconds, load average=16.56.
(mlfqs-load-60) After 20 seconds, load average=18.00.
(mlfqs-load-60) After 22 seconds, load average=19.39.
(mlfqs-load-60) After 24 seconds, load average=20.73.
(mlfqs-load-60) After 26 seconds, load average=22.03.
(mlfqs-load-60) After 28 seconds, load average=23.28.
(mlfqs-load-60) After 30 seconds, load average=24.50.
(mlfqs-load-60) After 32 seconds, load average=25.67.
(mlfqs-load-60) After 34 seconds, load average=26.80.
(mlfqs-load-60) After 36 seconds, load average=27.90.
(mlfqs-load-60) After 38 seconds, load average=28.96.
(mlfqs-load-60) After 40 seconds, load average=29.99.
(mlfqs-load-60) After 42 seconds, load average=30.98.
(mlfqs-load-60) After 44 seconds, load average=31.94.
(mlfqs-load-60) After 46 seconds, load average=32.87.
(mlfqs-load-60) After 48 seconds, load average=33.76.
(mlfqs-load-60) After 50 seconds, load average=34.63.
(mlfqs-load-60) After 52 seconds, load average=35.47.
(mlfqs-load-60) After 54 seconds, load average=36.28.
(mlfqs-load-60) After 56 seconds, load average=37.06.
(mlfqs-load-60) After 58 seconds, load average=37.82.
(mlfqs-load-60) After 60 seconds, load average=38.56.
```

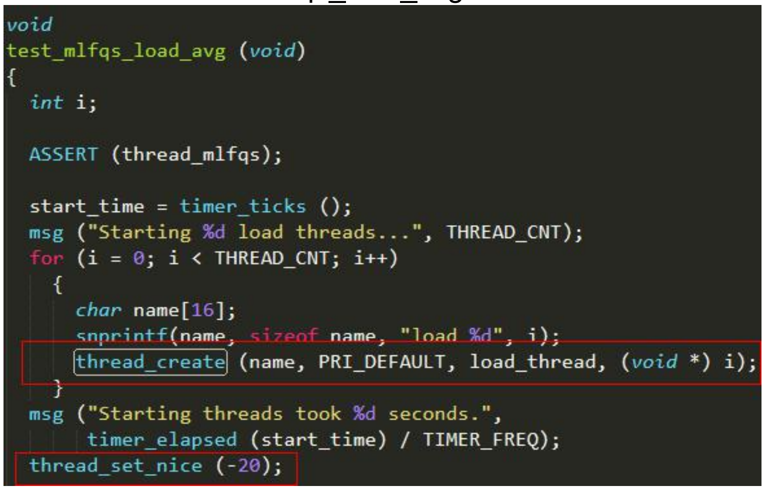


```
(mlfqs-load-60) After 60 seconds, load average=38.56.
(mlfqs-load-60) After 62 seconds, load average=37.28.
(mlfqs-load-60) After 64 seconds, load average=36.05.
(mlfqs-load-60) After 66 seconds, load average=34.86.
(mlfqs-load-60) After 68 seconds, load average=33.71.
(mlfqs-load-60) After 70 seconds, load average=32.59.
(mlfqs-load-60) After 72 seconds, load average=31.51.
(mlfqs-load-60) After 74 seconds, load average=30.47.
(mlfqs-load-60) After 76 seconds, load average=29.46.
(mlfqs-load-60) After 78 seconds, load average=28.49.
(mlfqs-load-60) After 80 seconds, load average=27.55.
(mlfqs-load-60) After 82 seconds, load average=26.64.
(mlfqs-load-60) After 84 seconds, load average=25.76.
(mlfqs-load-60) After 86 seconds, load average=24.91.
(mlfqs-load-60) After 88 seconds, load average=24.08.
(mlfqs-load-60) After 90 seconds, load average=23.29.
(mlfqs-load-60) After 92 seconds, load average=22.52.
(mlfqs-load-60) After 94 seconds, load average=21.77.
(mlfqs-load-60) After 96 seconds, load average=21.05.
(mlfqs-load-60) After 98 seconds, load average=20.36.
(mlfqs-load-60) After 100 seconds, load average=19.68.
(mlfqs-load-60) After 102 seconds, load average=19.03.
(mlfqs-load-60) After 104 seconds, load average=18.40.
(mlfqs-load-60) After 106 seconds, load average=17.80.
(mlfqs-load-60) After 108 seconds, load average=17.21.
(mlfqs-load-60) After 110 seconds, load average=16.64.
(mlfqs-load-60) After 112 seconds, load average=16.09.
(mlfqs-load-60) After 114 seconds, load average=15.56.
(mlfqs-load-60) After 116 seconds, load average=15.04.
(mlfqs-load-60) After 118 seconds, load average=14.55.
(mlfqs-load-60) After 120 seconds, load average=14.06.
```



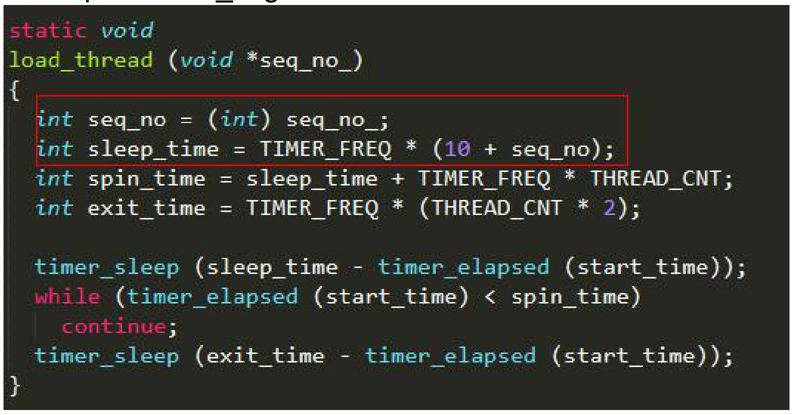
```
(mlfqs-load-60) After 120 seconds, load average=14.06.
(mlfqs-load-60) After 122 seconds, load average=13.63.
(mlfqs-load-60) After 124 seconds, load average=13.18.
(mlfqs-load-60) After 126 seconds, load average=12.75.
(mlfqs-load-60) After 128 seconds, load average=12.32.
(mlfqs-load-60) After 130 seconds, load average=11.92.
(mlfqs-load-60) After 132 seconds, load average=11.52.
(mlfqs-load-60) After 134 seconds, load average=11.14.
(mlfqs-load-60) After 136 seconds, load average=10.77.
(mlfqs-load-60) After 138 seconds, load average=10.42.
(mlfqs-load-60) After 140 seconds, load average=10.07.
(mlfqs-load-60) After 142 seconds, load average=9.74.
(mlfqs-load-60) After 144 seconds, load average=9.42.
(mlfqs-load-60) After 146 seconds, load average=9.11.
(mlfqs-load-60) After 148 seconds, load average=8.81.
(mlfqs-load-60) After 150 seconds, load average=8.52.
(mlfqs-load-60) After 152 seconds, load average=8.23.
(mlfqs-load-60) After 154 seconds, load average=7.96.
(mlfqs-load-60) After 156 seconds, load average=7.70.
(mlfqs-load-60) After 158 seconds, load average=7.44.
(mlfqs-load-60) After 160 seconds, load average=7.20.
(mlfqs-load-60) After 162 seconds, load average=6.96.
(mlfqs-load-60) After 164 seconds, load average=6.73.
(mlfqs-load-60) After 166 seconds, load average=6.51.
(mlfqs-load-60) After 168 seconds, load average=6.29.
(mlfqs-load-60) After 170 seconds, load average=6.08.
(mlfqs-load-60) After 172 seconds, load average=5.88.
(mlfqs-load-60) After 174 seconds, load average=5.69.
(mlfqs-load-60) After 176 seconds, load average=5.50.
(mlfqs-load-60) After 178 seconds, load average=5.32.
```

**Test 4：** mlfqs_load_avg
代码的思路和 test2 mlfqs_load_avg 基本一致，这里主要讲不同点：



```c
void
test_mlfqs_load_avg (void)
{
  int i;

  ASSERT (thread_mlfqs);

  start_time = timer_ticks ();
  msg ("Starting %d load threads...", THREAD_CNT);
  for (i = 0; i < THREAD_CNT; i++)
    {
      char name[16];
      snprintf(name, sizeof name, "load %d", i);
      thread_create (name, PRI_DEFAULT, load_thread, (void *) i);
    }
  msg ("Starting threads took %d seconds.",
       timer_elapsed (start_time) / TIMER_FREQ);
  thread_set_nice (-20);
```

在进入循环，创建 60 个子线程的时候，传入了一个参数 i，用于后面子线程的休眠时间调用 thread_set_nice 函数将线程的 nice 设为-20，目的是为了降低主线程的友好度。增大自身的优先级获得更多的时间片，从而使得主线程能在休眠结束后获得 cpu，执行 msg 输出和 cpu 的 load_avg。



```
static void
load_thread (void *seq_no_)
{
  int seq_no = (int) seq_no_;
  int sleep_time = TIMER_FREQ * (10 + seq_no);
  int spin_time = sleep_time + TIMER_FREQ * THREAD_CNT;
  int exit_time = TIMER_FREQ * (THREAD_CNT * 2);

  timer_sleep (sleep_time - timer_elapsed (start_time));
  while (timer_elapsed (start_time) < spin_time)
    continue;
  timer_sleep (exit_time - timer_elapsed (start_time));
}
```

第一次子线程进入休眠，此时每个子线程的休眠时间是不一样的，并且每个线程休眠结束的时间间隔是 1s。每个子线程 while 循环结束的时间不一样，第二次进入休眠，休眠花费的时间也不一样，load 51 到 load 59 是不会进入第二次休眠。因为线程 load 50 第一次休眠结束的时间是在 60s，进入 while 循环，会在第 120s 时刻结束 while 循环，此时 exit_time-timer_elapsed(start_time) = 0，因此不会进行休眠，直接结束线程。所以 load 50 是一个临界值，在 load_50 以后创建的线程都不会进行第二次休眠。

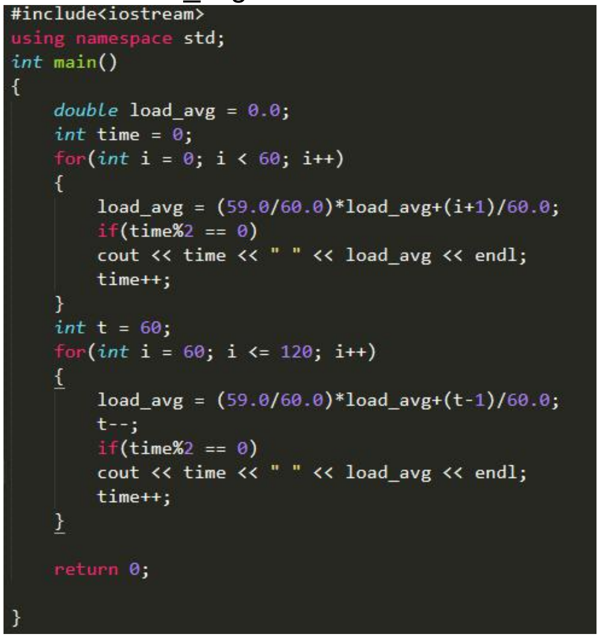
综上分析：

在 0~59s(参照物为 10s)，每隔 1s 子线程休眠结束，进入就绪队列中

在 60s~110s,每隔 1s 子线程会进入休眠状态。

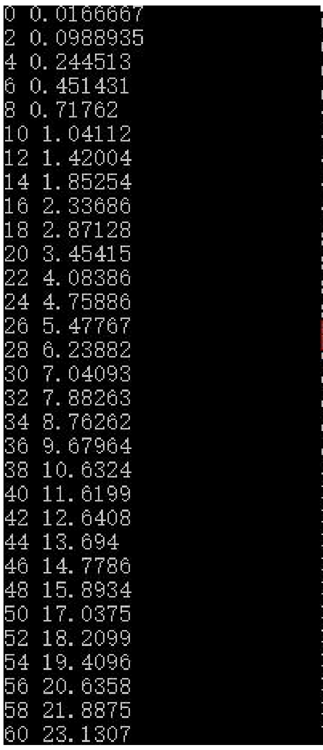
110s~120s 时，load51 到 load 59 依次每隔 1s 结束线程，load0 到 load50 线程结束休眠，结束进程。

理论计算 load_avg 的值：

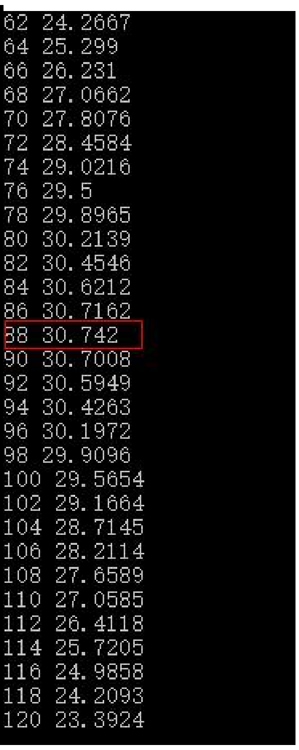


```
#include<iostream>
using namespace std;
int main()
{
    double load_avg = 0.0;
    int time = 0;
    for(int i = 0; i < 60; i++)
    {
        load_avg = (59.0/60.0)*load_avg+(i+1)/60.0;
        if(time%2 == 0)
        cout << time << " " << load_avg << endl;
        time++;
    }
    int t = 60;
    for(int i = 60; i <= 120; i++)
    {
        load_avg = (59.0/60.0)*load_avg+(t-1)/60.0;
        t--;
        if(time%2 == 0)
        cout << time << " " << load_avg << endl;
        time++;
    }

    return 0;

}
```

理论值计算结果为：



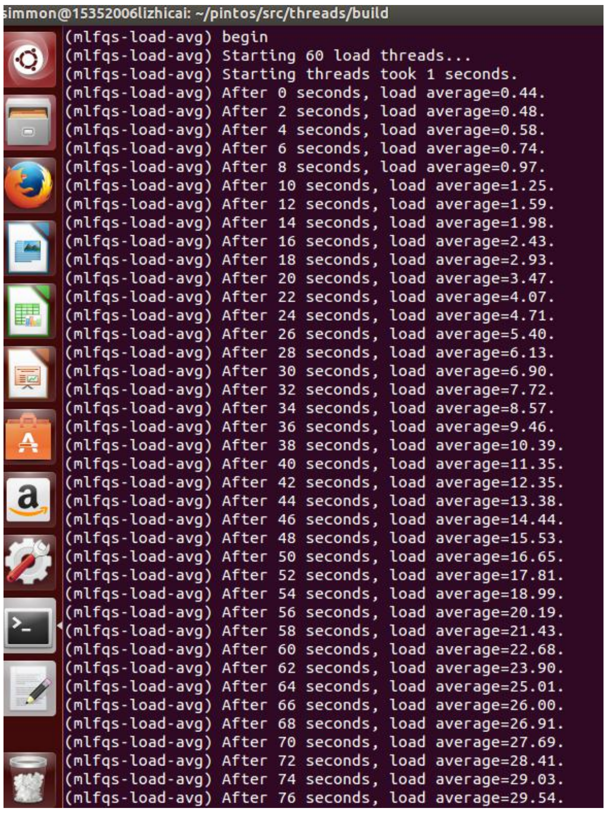
```
0 0.0166667
2 0.0988935
4 0.244513
6 0.451431
8 0.71762
10 1.04112
12 1.42004
14 1.85254
16 2.33686
18 2.87128
20 3.45415
22 4.08386
24 4.75886
26 5.47767
28 6.23882
30 7.04093
32 7.88263
34 8.76262
36 9.67964
38 10.6324
40 11.6199
42 12.6408
44 13.694
46 14.7786
48 15.8934
50 17.0375
52 18.2099
54 19.4096
56 20.6358
58 21.8875
60 23.1307
```



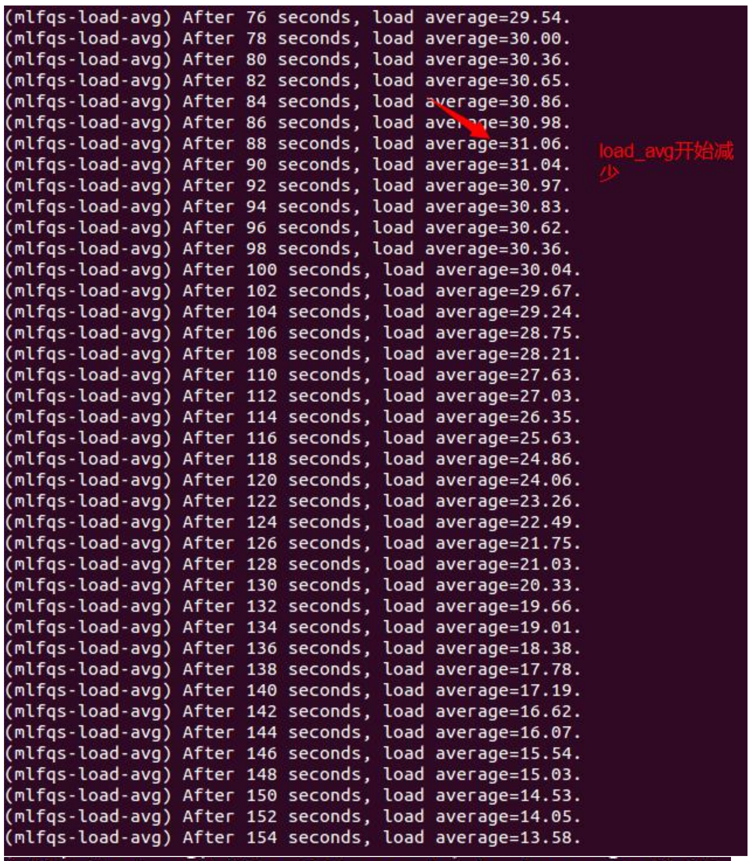
```
62 24.2667
64 25.299
66 26.231
68 27.0662
70 27.8076
72 28.4584
74 29.0216
76 29.5
78 29.8965
80 30.2139
82 30.4546
84 30.6212
86 30.7162
88 30.742
90 30.7008
92 30.5949
94 30.4263
96 30.1972
98 29.9096
100 29.5654
102 29.1664
104 28.7145
106 28.2114
108 27.6589
110 27.0585
112 26.4118
114 25.7205
116 24.9858
118 24.2093
120 23.3924
```

根据理论分析可知，在 88s 前，load_avg 逐渐增大，88s 后 load_avg 逐渐减小

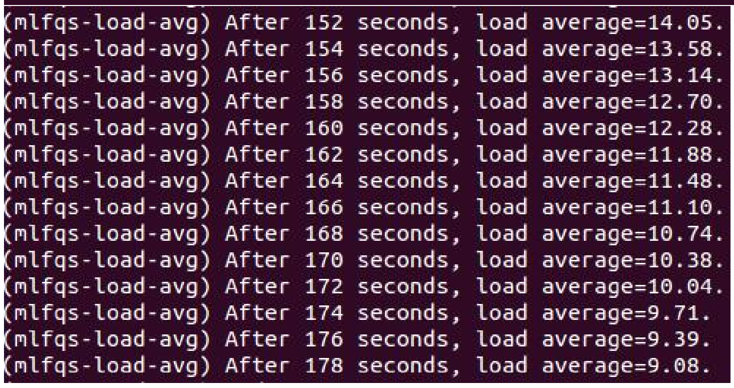
所以结果为：



```
simmon@15352006lizhicai: ~/pintos/src/threads/build
(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=0.44.
(mlfqs-load-avg) After 2 seconds, load average=0.48.
(mlfqs-load-avg) After 4 seconds, load average=0.58.
(mlfqs-load-avg) After 6 seconds, load average=0.74.
(mlfqs-load-avg) After 8 seconds, load average=0.97.
(mlfqs-load-avg) After 10 seconds, load average=1.25.
(mlfqs-load-avg) After 12 seconds, load average=1.59.
(mlfqs-load-avg) After 14 seconds, load average=1.98.
(mlfqs-load-avg) After 16 seconds, load average=2.43.
(mlfqs-load-avg) After 18 seconds, load average=2.93.
(mlfqs-load-avg) After 20 seconds, load average=3.47.
(mlfqs-load-avg) After 22 seconds, load average=4.07.
(mlfqs-load-avg) After 24 seconds, load average=4.71.
(mlfqs-load-avg) After 26 seconds, load average=5.40.
(mlfqs-load-avg) After 28 seconds, load average=6.13.
(mlfqs-load-avg) After 30 seconds, load average=6.90.
(mlfqs-load-avg) After 32 seconds, load average=7.72.
(mlfqs-load-avg) After 34 seconds, load average=8.57.
(mlfqs-load-avg) After 36 seconds, load average=9.46.
(mlfqs-load-avg) After 38 seconds, load average=10.39.
(mlfqs-load-avg) After 40 seconds, load average=11.35.
(mlfqs-load-avg) After 42 seconds, load average=12.35.
(mlfqs-load-avg) After 44 seconds, load average=13.38.
(mlfqs-load-avg) After 46 seconds, load average=14.44.
(mlfqs-load-avg) After 48 seconds, load average=15.53.
(mlfqs-load-avg) After 50 seconds, load average=16.65.
(mlfqs-load-avg) After 52 seconds, load average=17.81.
(mlfqs-load-avg) After 54 seconds, load average=18.99.
(mlfqs-load-avg) After 56 seconds, load average=20.19.
(mlfqs-load-avg) After 58 seconds, load average=21.43.
(mlfqs-load-avg) After 60 seconds, load average=22.68.
(mlfqs-load-avg) After 62 seconds, load average=23.90.
(mlfqs-load-avg) After 64 seconds, load average=25.01.
(mlfqs-load-avg) After 66 seconds, load average=26.00.
(mlfqs-load-avg) After 68 seconds, load average=26.91.
(mlfqs-load-avg) After 70 seconds, load average=27.69.
(mlfqs-load-avg) After 72 seconds, load average=28.41.
(mlfqs-load-avg) After 74 seconds, load average=29.03.
(mlfqs-load-avg) After 76 seconds, load average=29.54.
```



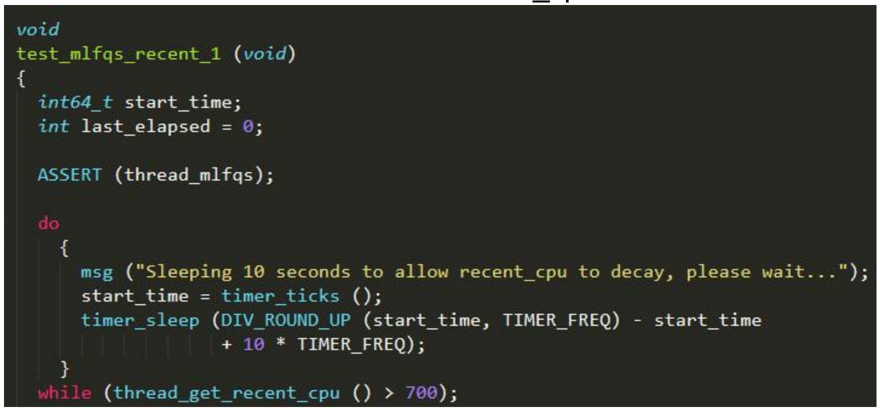
```
(mlfqs-load-avg) After 76 seconds, load average=29.54.
(mlfqs-load-avg) After 78 seconds, load average=30.00.
(mlfqs-load-avg) After 80 seconds, load average=30.36.
(mlfqs-load-avg) After 82 seconds, load average=30.65.
(mlfqs-load-avg) After 84 seconds, load average=30.86.
(mlfqs-load-avg) After 86 seconds, load average=30.98.
(mlfqs-load-avg) After 88 seconds, load average=31.06.
(mlfqs-load-avg) After 90 seconds, load average=31.04.
(mlfqs-load-avg) After 92 seconds, load average=30.97.
(mlfqs-load-avg) After 94 seconds, load average=30.83.
(mlfqs-load-avg) After 96 seconds, load average=30.62.
(mlfqs-load-avg) After 98 seconds, load average=30.36.
(mlfqs-load-avg) After 100 seconds, load average=30.04.
(mlfqs-load-avg) After 102 seconds, load average=29.67.
(mlfqs-load-avg) After 104 seconds, load average=29.24.
(mlfqs-load-avg) After 106 seconds, load average=28.75.
(mlfqs-load-avg) After 108 seconds, load average=28.21.
(mlfqs-load-avg) After 110 seconds, load average=27.63.
(mlfqs-load-avg) After 112 seconds, load average=27.03.
(mlfqs-load-avg) After 114 seconds, load average=26.35.
(mlfqs-load-avg) After 116 seconds, load average=25.63.
(mlfqs-load-avg) After 118 seconds, load average=24.86.
(mlfqs-load-avg) After 120 seconds, load average=24.06.
(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=20.33.
(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=13.58.
```



```
(mlfqs-load-avg) After 152 seconds, load average=14.05.
(mlfqs-load-avg) After 154 seconds, load average=13.58.
(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.
```

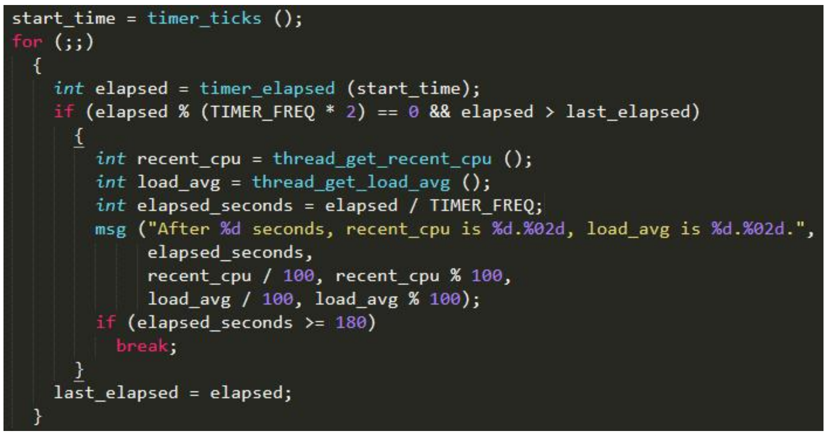
**Test 5：** mlfqs_recent_1

测试目的：检查单个就绪的线程其 recent_cpu 是否计算正确



```
void
test_mlfqs_recent_1 (void)
{
  int64_t start_time;
  int last_elapsed = 0;

  ASSERT (thread_mlfqs);

  do
    {
      msg ("Sleeping 10 seconds to allow recent_cpu to decay, please wait...");
      start_time = timer_ticks ();
      timer_sleep (DIV_ROUND_UP (start_time, TIMER_FREQ) - start_time
                   + 10 * TIMER_FREQ);
    }
  while (thread_get_recent_cpu () > 700);
```

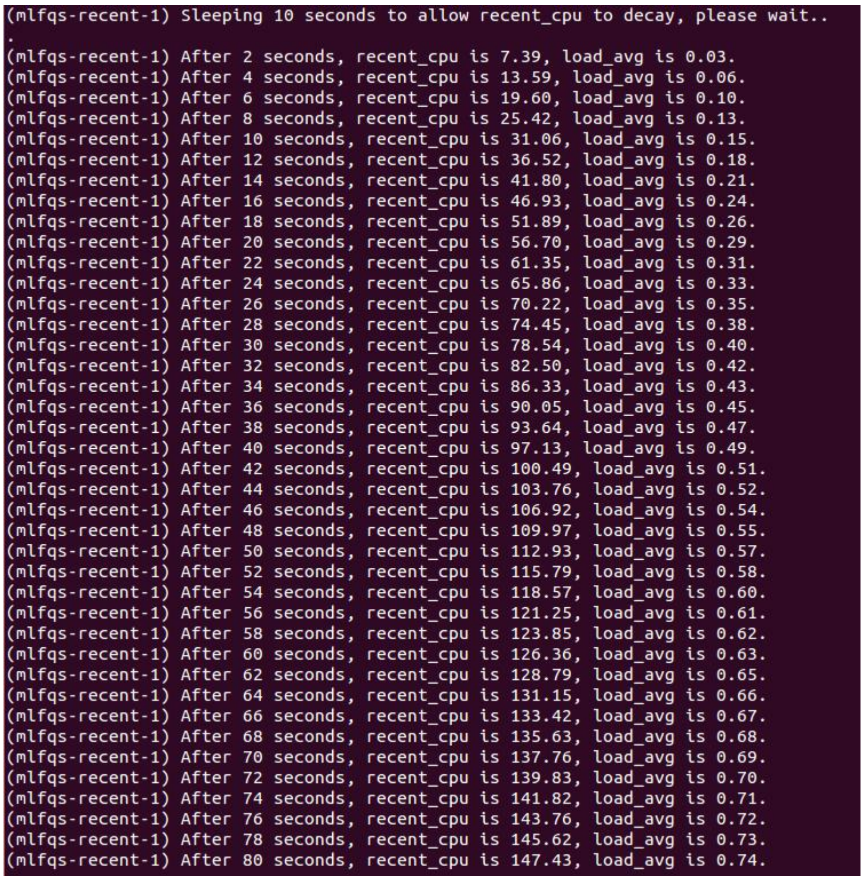
声明所需的变量 start_time 和 last_elapsed，进入一个循环，每次进入循环都会主线程都会休眠 10s，使得 recent_cpu 降低，当 recent_cpu 降低到小于获等于 700 时，主线程退出这个 while 循环



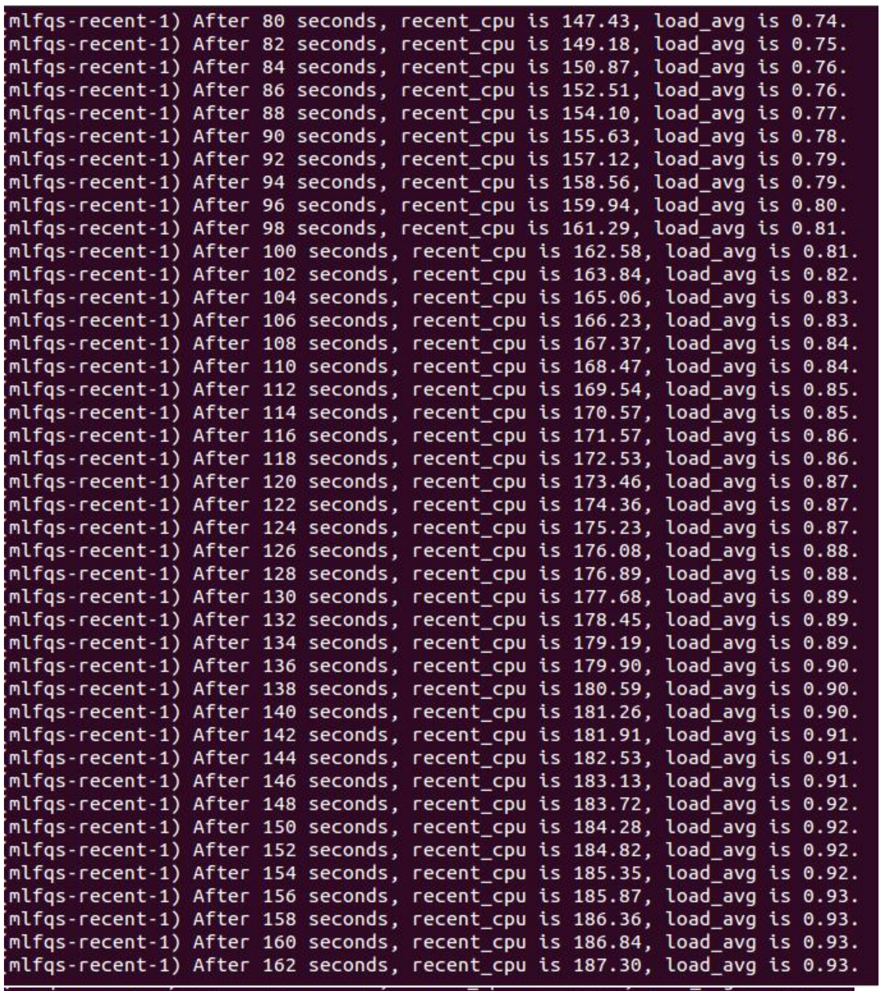
```
start_time = timer_ticks ();
for (;;)
  {
    int elapsed = timer_elapsed (start_time);
    if (elapsed % (TIMER_FREQ * 2) == 0 && elapsed > last_elapsed)
      {
        int recent_cpu = thread_get_recent_cpu ();
        int load_avg = thread_get_load_avg ();
        int elapsed_seconds = elapsed / TIMER_FREQ;
        msg ("After %d seconds, recent_cpu is %d.%02d, load_avg is %d.%02d.",
             elapsed_seconds,
             recent_cpu / 100, recent_cpu % 100,
             load_avg / 100, load_avg % 100);
        if (elapsed_seconds >= 180)
          break;
      }
    last_elapsed = elapsed;
  }
```

elapsed 记录了此时到进入循环前流逝的时间，一个判断语句，当流逝的时间是 2 秒时间的倍数时并且本次循环的流逝时间大于上一次循环的流逝时间，就执行判断语句内中的代码。判断语句内主要做的就是输出经过相应的偶数秒时间时，recent_cpu 和 load_avg 的值。
在就绪队列中就只有一个线程，因此只有主线程在 cpu 中运行，所以对应的 recent_cpu 会一直增大。

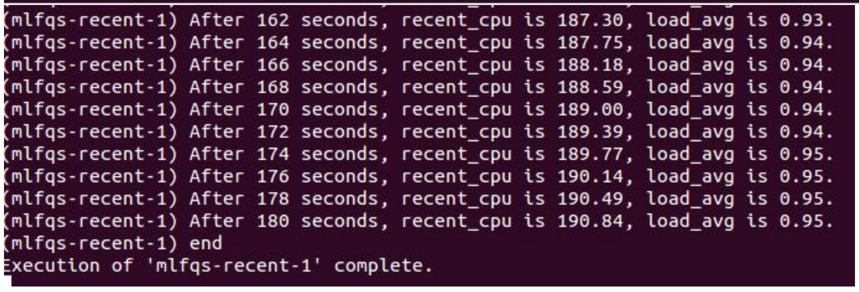
结果：



```
(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.
(mlfqs-recent-1) After 58 seconds, recent_cpu is 123.85, load_avg is 0.62.
(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.
```



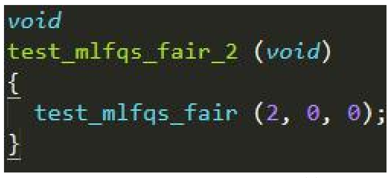
```
(mlfqs-recent-1) After 80 seconds, recent_cpu is 147.43, load_avg is 0.74.
(mlfqs-recent-1) After 82 seconds, recent_cpu is 149.18, load_avg is 0.75.
(mlfqs-recent-1) After 84 seconds, recent_cpu is 150.87, load_avg is 0.76.
(mlfqs-recent-1) After 86 seconds, recent_cpu is 152.51, load_avg is 0.76.
(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.
(mlfqs-recent-1) After 122 seconds, recent_cpu is 174.36, load_avg is 0.87.
(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 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.
```

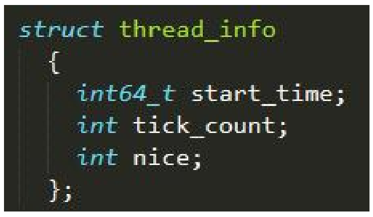
**Test 6：** mlfqs_fair_2

测试目的：测试当运行 2 个子线程，设定的 nice 都为 0 时，两个子线程得到的时间片是否相等。



```
void
test_mlfqs_fair_2 (void)
{
  test_mlfqs_fair (2, 0, 0);
}
```

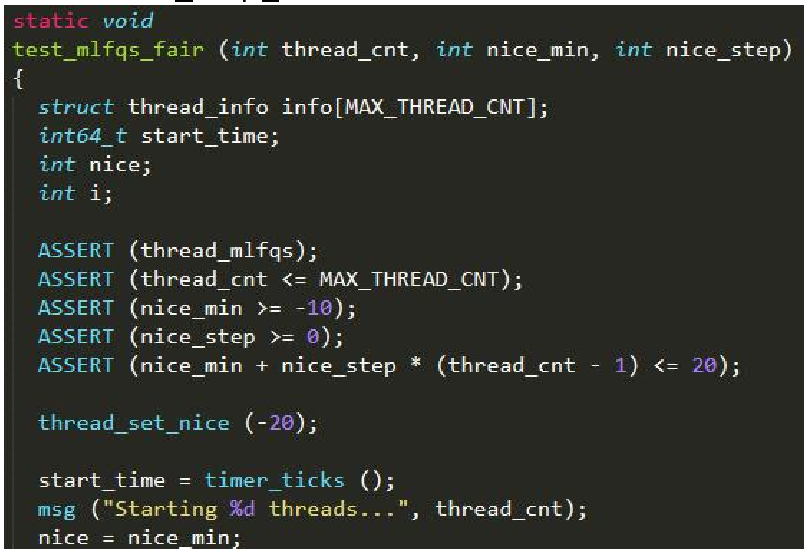
调用 test_mlfqs_fair 函数，传入 3 个参数，分别表示线程个数，nice 的最小值，和 nice_step 这里表示，共有 2 个线程，并且 2 个线程的 nice 值为 0



```
struct thread_info
  {
    int64_t start_time;
    int tick_count;
    int nice;
  };
```

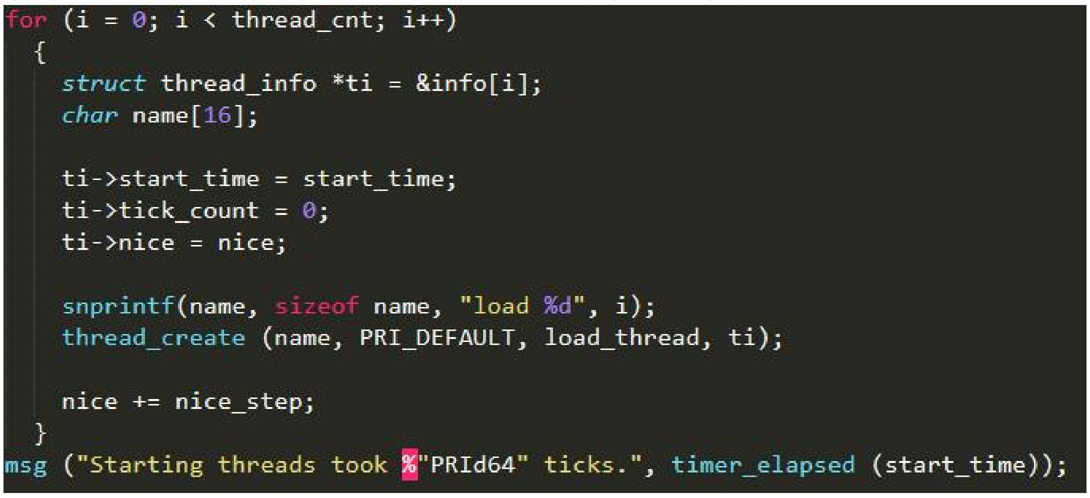
定义结构体 thread_info 记录线程的信息，包括时间片，开始时间，和 nice 值

首先分析 test_mlfqs_fair 函数



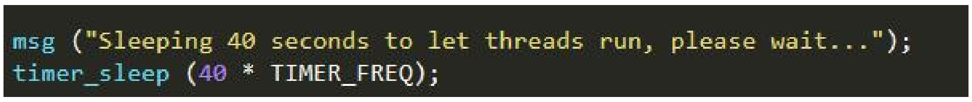
```c
static void
test_mlfqs_fair (int thread_cnt, int nice_min, int nice_step)
{
  struct thread_info info[MAX_THREAD_CNT];
  int64_t start_time;
  int nice;
  int i;

  ASSERT (thread_mlfqs);
  ASSERT (thread_cnt <= MAX_THREAD_CNT);
  ASSERT (nice_min >= -10);
  ASSERT (nice_step >= 0);
  ASSERT (nice_min + nice_step * (thread_cnt - 1) <= 20);

  thread_set_nice (-20);

  start_time = timer_ticks ();
  msg ("Starting %d threads...", thread_cnt);
  nice = nice_min;
```

声明了一个 thread_info 类型的 info 数组，记录每个线程的信息。调用 thread_set_nice 函数将主线程的 nice 值定为-20，提高主线程的优先级，获得更多的时间片。



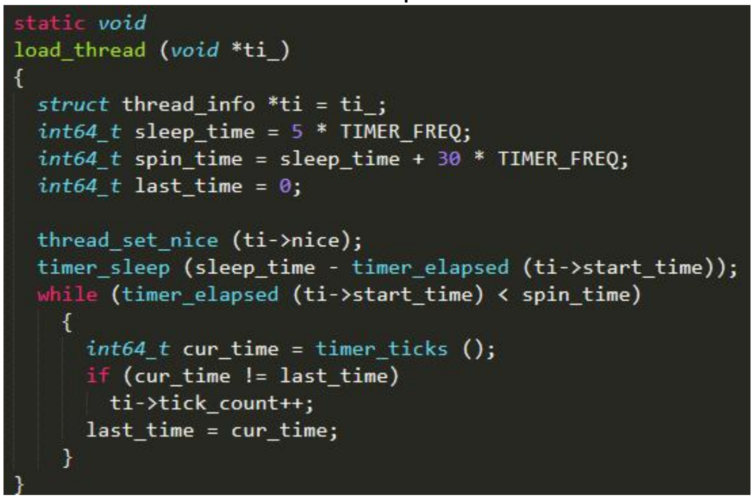
```c
for (i = 0; i < thread_cnt; i++)
  {
    struct thread_info *ti = &info[i];
    char name[16];

    ti->start_time = start_time;
    ti->tick_count = 0;
    ti->nice = nice;

    snprintf(name, sizeof name, "load %d", i);
    thread_create (name, PRI_DEFAULT, load_thread, ti);

    nice += nice_step;
  }
msg ("Starting threads took %"PRId64" ticks.", timer_elapsed (start_time));
```

进入一个循环，该循环会循环 thread_cnt 次，每一次都会将线程的信息记录到 info 数组中，创建一个子线程，并且更新 nice 值。



```c
msg ("Sleeping 40 seconds to let threads run, please wait...");
timer_sleep (40 * TIMER_FREQ);
```

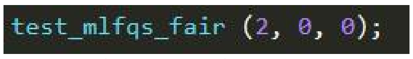
主线程休眠 40s，子线程获得 cpu 运行



```c
static void
load_thread (void *ti_)
{
  struct thread_info *ti = ti_;
  int64_t sleep_time = 5 * TIMER_FREQ;
  int64_t spin_time = sleep_time + 30 * TIMER_FREQ;
  int64_t last_time = 0;

  thread_set_nice (ti->nice);
  timer_sleep (sleep_time - timer_elapsed (ti->start_time));
  while (timer_elapsed (ti->start_time) < spin_time)
    {
      int64_t cur_time = timer_ticks ();
      if (cur_time != last_time)
        ti->tick_count++;
      last_time = cur_time;
    }
}
```

子线程会调用 thread_set_nice 函数将记录在 ti 中的 nice 值设置为自己的 nice 值，然后休眠到第 5s，休眠期间，其他的子线程获得 cpu 运行，同样也休眠到第 5s，获得 cpu 的子线程进入 while 循环，进入一个判断，目的是为了确保系统的 ticks 时间有正确运行。如果满足条件，就 tick_count++.

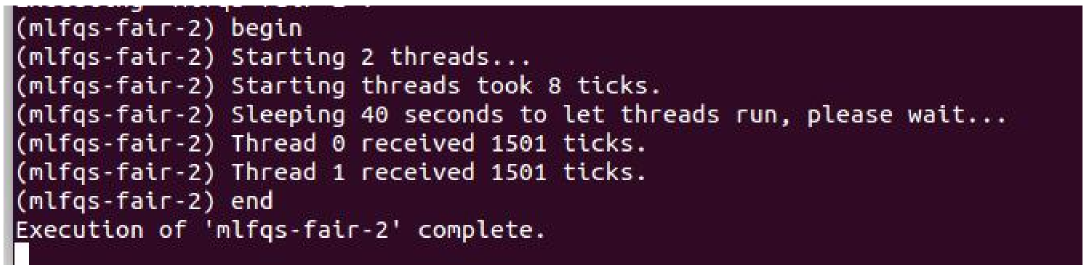
现在倒回来分析 test



```
test_mlfqs_fair (2, 0, 0);
```

主线程会创建 2 个子线程，并且 nice 值都为 0，根据公式可知两个子线程的优先级水平基本相同，所以两个子线程获得时间片基本相同。并且每个 test 都运行 30s，也就是 30*100=3000ticks. 所以两个子线程获得的时间片也就是 ticks_count 大概都是 3000ticks 的一半。

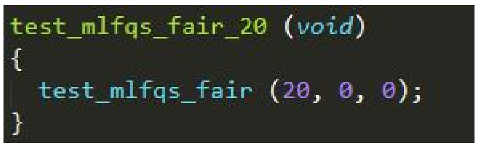
结果：



```
(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 1501 ticks.
(mlfqs-fair-2) Thread 1 received 1501 ticks.
(mlfqs-fair-2) end
Execution of 'mlfqs-fair-2' complete.
```

**Test 7**：mlfqs_fair_20

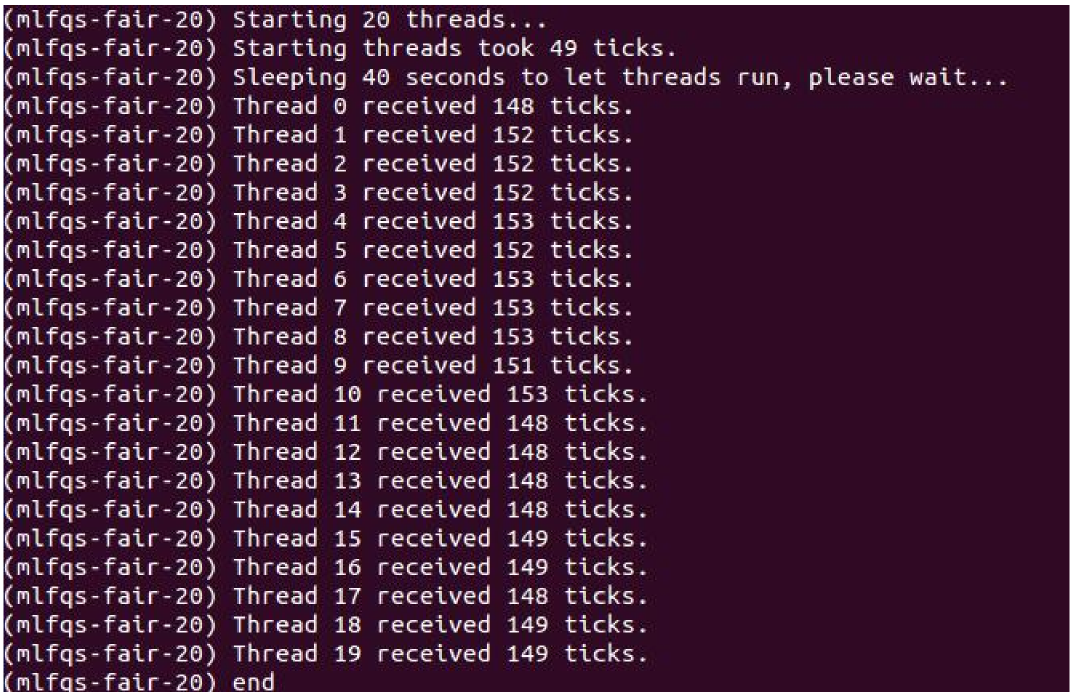
测试目的：测试当运行 20 个子线程，设定的 nice 都为 0 时，20 子线程得到的时间片是基本否相等。



```
test_mlfqs_fair_20 (void)
{
  test_mlfqs_fair (20, 0, 0);
}
```

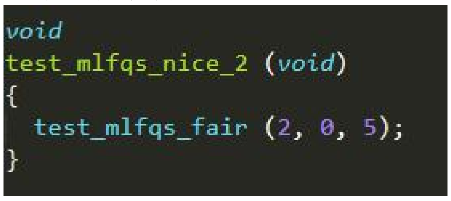
同理和 test6 的分析一致，因为这 20 个子线程的 nice 值都为 0，他们的优先级水平相同，所以获得的时间片基本都差不多。

结果：



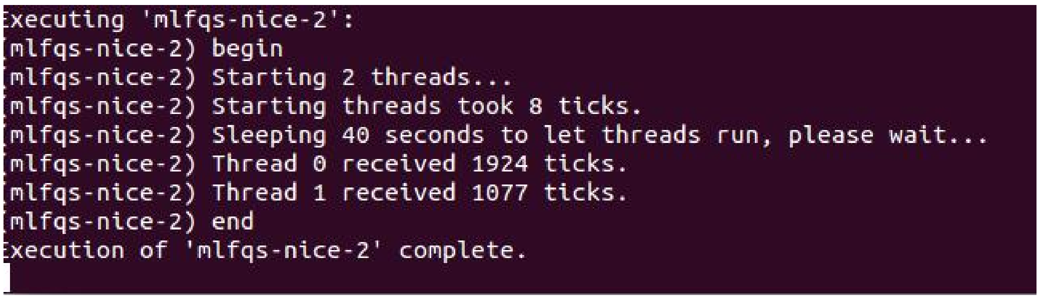
```
(mlfqs-fair-20) Starting 20 threads...
(mlfqs-fair-20) Starting threads took 49 ticks.
(mlfqs-fair-20) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-fair-20) Thread 0 received 148 ticks.
(mlfqs-fair-20) Thread 1 received 152 ticks.
(mlfqs-fair-20) Thread 2 received 152 ticks.
(mlfqs-fair-20) Thread 3 received 152 ticks.
(mlfqs-fair-20) Thread 4 received 153 ticks.
(mlfqs-fair-20) Thread 5 received 152 ticks.
(mlfqs-fair-20) Thread 6 received 153 ticks.
(mlfqs-fair-20) Thread 7 received 153 ticks.
(mlfqs-fair-20) Thread 8 received 153 ticks.
(mlfqs-fair-20) Thread 9 received 151 ticks.
(mlfqs-fair-20) Thread 10 received 153 ticks.
(mlfqs-fair-20) Thread 11 received 148 ticks.
(mlfqs-fair-20) Thread 12 received 148 ticks.
(mlfqs-fair-20) Thread 13 received 148 ticks.
(mlfqs-fair-20) Thread 14 received 148 ticks.
(mlfqs-fair-20) Thread 15 received 149 ticks.
(mlfqs-fair-20) Thread 16 received 149 ticks.
(mlfqs-fair-20) Thread 17 received 148 ticks.
(mlfqs-fair-20) Thread 18 received 149 ticks.
(mlfqs-fair-20) Thread 19 received 149 ticks.
(mlfqs-fair-20) end
```

**Test 8：**mlfqs_nice_2
测试目的：当跑 2 个线程的时候，一个线程的 nice 为 0，另一个线程的 nice 为 5，测试哪一个线程获得时间片多。



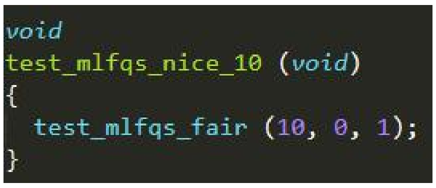
```
void
test_mlfqs_nice_2 (void)
{
  test_mlfqs_fair (2, 0, 5);
}
```

创建了 2 个线程，根据 test 6 的分析可知，第一个创建的子线程的 nice 值为 0，第二个创建的线程 nice 值为 5.根据公式：load_avg = (59/60)*load_avg + (1/60)*ready_threads
Recent_cpu = (2*load_avg)/(2*load_avg+1)*recent_cpu +nice
Priority = PRI_MAX - (recent_cpu/4)-(nice*2);
nice 值小的 priority 会大于 nice 值大的 priority，所以 load 0 获得更多的时间片，thread 1 获得较少的时间片，即 thread 0 获得 1904 ticks， load 1 获得 1096 ticks。



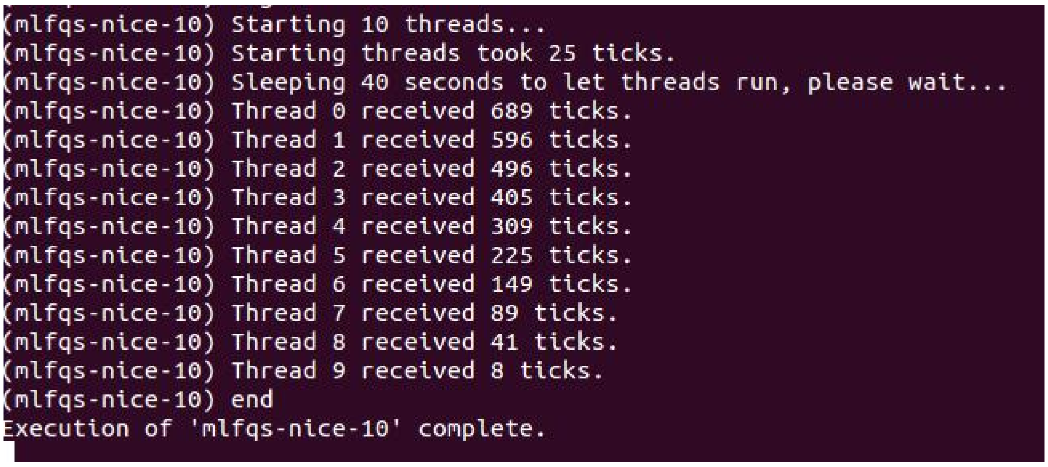
```
Executing 'mlfqs-nice-2':
(mlfqs-nice-2) begin
(mlfqs-nice-2) Starting 2 threads...
(mlfqs-nice-2) Starting threads took 8 ticks.
(mlfqs-nice-2) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-nice-2) Thread 0 received 1924 ticks.
(mlfqs-nice-2) Thread 1 received 1077 ticks.
(mlfqs-nice-2) end
Execution of 'mlfqs-nice-2' complete.
```

**Test 9：**mlfqs_nice_10
测试目的：当跑 10 个线程，这 10 个线程的 nice 依次为 0，1，2，。。9，测试这 10 个线程获得的时间片是否正常。



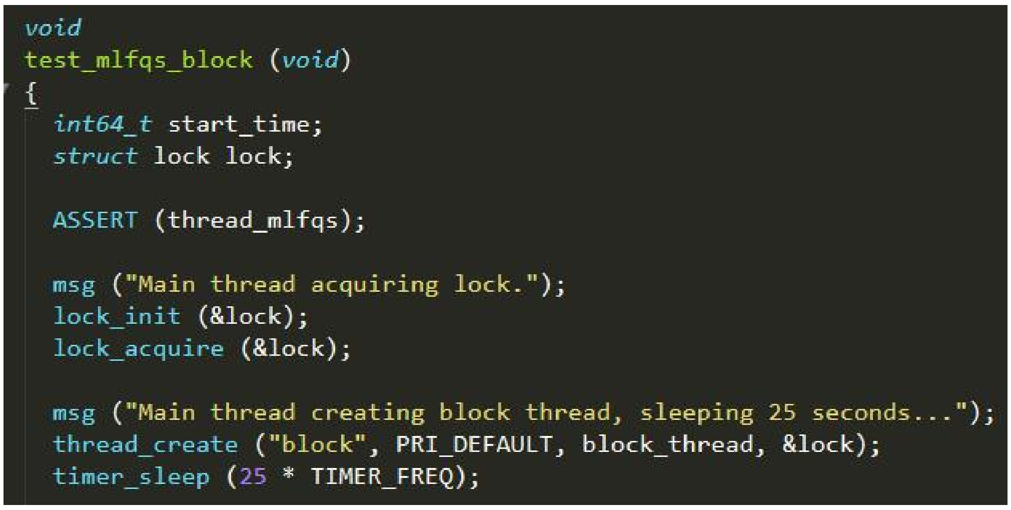
```
void
test_mlfqs_nice_10 (void)
{
  test_mlfqs_fair (10, 0, 1);
}
```

创建了 10 个线程，同理根据 test6 分析可知，这 10 个线程的 nice 值依次为 0，1，2，3，4.。。。9，根据 test 8 的公式分析可知，nice 越大，线程获得的时间片越少，因此，tjread 1 获得的时间片最多，然后一直到 thread 9 依次减小。



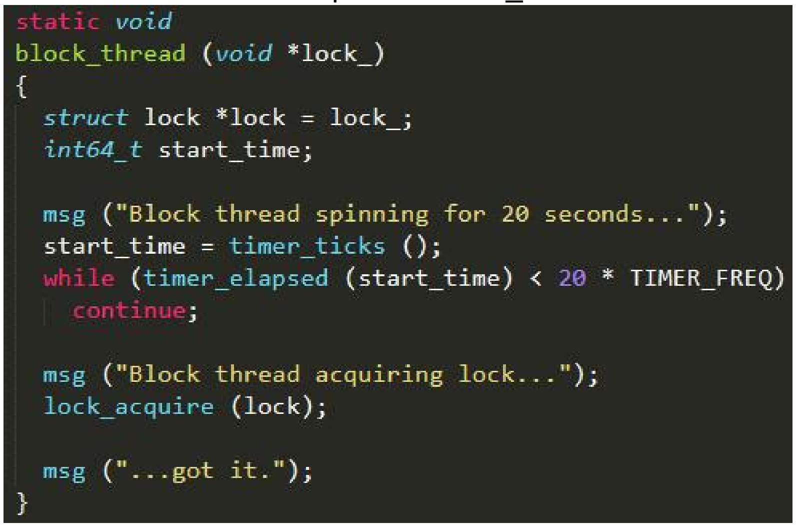
```
(mlfqs-nice-10) Starting 10 threads...
(mlfqs-nice-10) Starting threads took 25 ticks.
(mlfqs-nice-10) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-nice-10) Thread 0 received 689 ticks.
(mlfqs-nice-10) Thread 1 received 596 ticks.
(mlfqs-nice-10) Thread 2 received 496 ticks.
(mlfqs-nice-10) Thread 3 received 405 ticks.
(mlfqs-nice-10) Thread 4 received 309 ticks.
(mlfqs-nice-10) Thread 5 received 225 ticks.
(mlfqs-nice-10) Thread 6 received 149 ticks.
(mlfqs-nice-10) Thread 7 received 89 ticks.
(mlfqs-nice-10) Thread 8 received 41 ticks.
(mlfqs-nice-10) Thread 9 received 8 ticks.
(mlfqs-nice-10) end
Execution of 'mlfqs-nice-10' complete.
```

**Test 10：**mlfqs_block
测试目的：测试当线程被阻塞的时候，能否更新线程的优先级和 recent_cpu



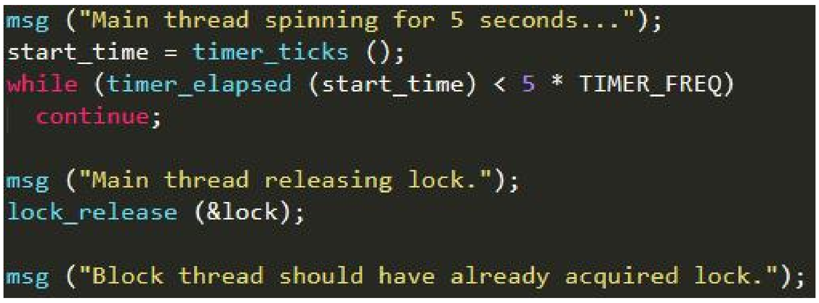
```c
void
test_mlfqs_block (void)
{
  int64_t start_time;
  struct lock lock;

  ASSERT (thread_mlfqs);

  msg ("Main thread acquiring lock.");
  lock_init (&lock);
  lock_acquire (&lock);

  msg ("Main thread creating block thread, sleeping 25 seconds...");
  thread_create ("block", PRI_DEFAULT, block_thread, &lock);
  timer_sleep (25 * TIMER_FREQ);
```

主线程得到锁 lock，创建一个线程 block，优先级和主线程一样大小。接着进入休眠，休眠 25s，子线程 block 获得 cpu 执行 block_thread 函数



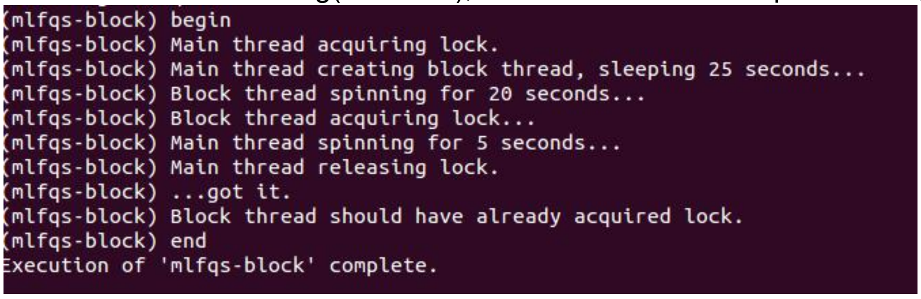
```c
static void
block_thread (void *lock_)
{
  struct lock *lock = lock_;
  int64_t start_time;

  msg ("Block thread spinning for 20 seconds...");
  start_time = timer_ticks ();
  while (timer_elapsed (start_time) < 20 * TIMER_FREQ)
    continue;

  msg ("Block thread acquiring lock...");
  lock_acquire (lock);

  msg ("...got it.");
}
```

子线程先进入 while 循环，等待 20s，接着申请锁被阻塞。主线程获得 cpu



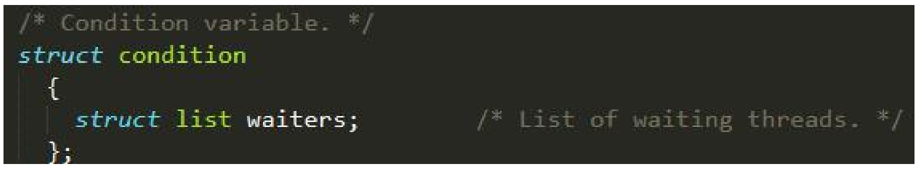
```c
msg ("Main thread spinning for 5 seconds...");
start_time = timer_ticks ();
while (timer_elapsed (start_time) < 5 * TIMER_FREQ)
  continue;

msg ("Main thread releasing lock.");
lock_release (&lock);

msg ("Block thread should have already acquired lock.");
```

主线程等待 5s，此时子线程 block 被 block 了 10s，子线程在 block 时依旧更新着 recent_cpu 和 priority，接着主线程释放锁 lock，因为主线程在过去的一段时间内一直占用了 cpu，所以主线程的 recent_cpu 高于子线程，其优先级会低于子线程 block，所以子线程会抢占 cpu，获得锁 lock，并输出 msg(“.. Got it”)，执行完后主线程获得 cpu 输出 msg 的信息。



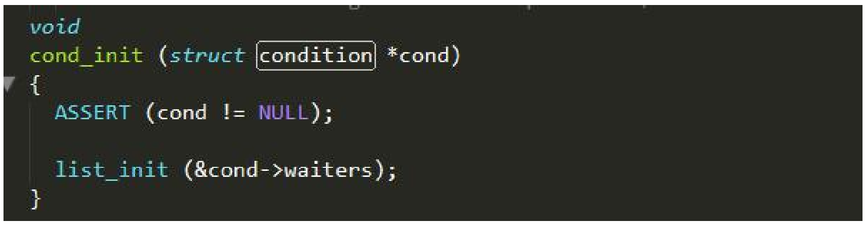
```
(mlfqs-block) begin
(mlfqs-block) Main thread acquiring lock.
(mlfqs-block) Main thread creating block thread, sleeping 25 seconds...
(mlfqs-block) Block thread spinning for 20 seconds...
(mlfqs-block) Block thread acquiring lock...
(mlfqs-block) Main thread spinning for 5 seconds...
(mlfqs-block) 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.
```

(二)分析及实现（30 分）
实验前未修改相关代码分析：



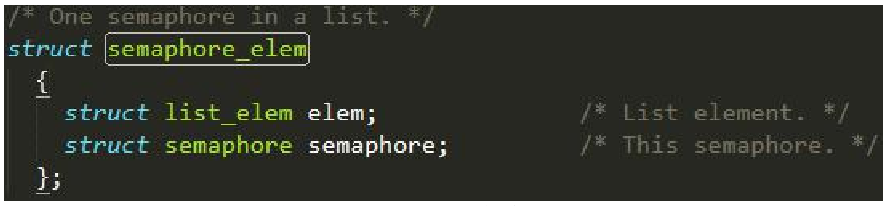
```
/* Condition variable. */
struct condition
  {
    struct list waiters;        /* List of waiting threads. */
  };
```

状态变量 condition 结构体，声明了一个 list 结构体的变量 waiters，waiters 表示的是被这个状态变量 condition 所阻塞的线程的链表。



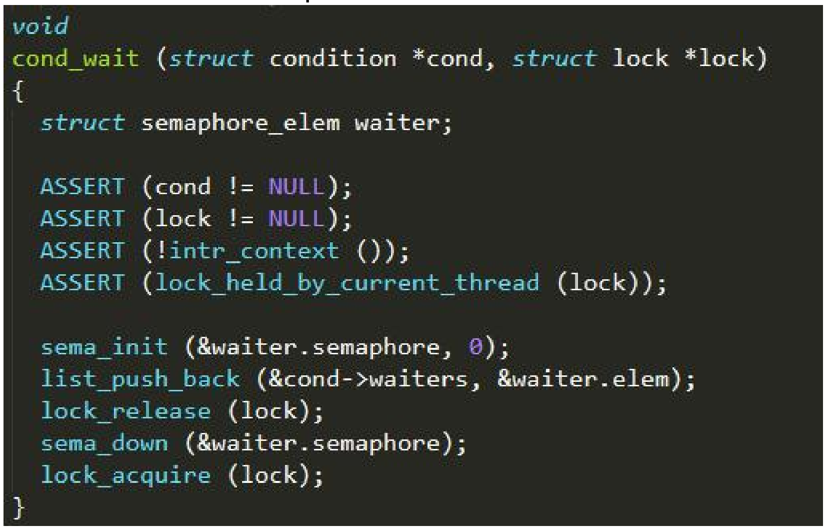
```
void
cond_init (struct condition *cond)
{
  ASSERT (cond != NULL);

  list_init (&cond->waiters);
}
```

初始化状态变量，将状态变量中的 waiters 链表初始化为空，表明一开始没有线程被状态变量所阻塞。



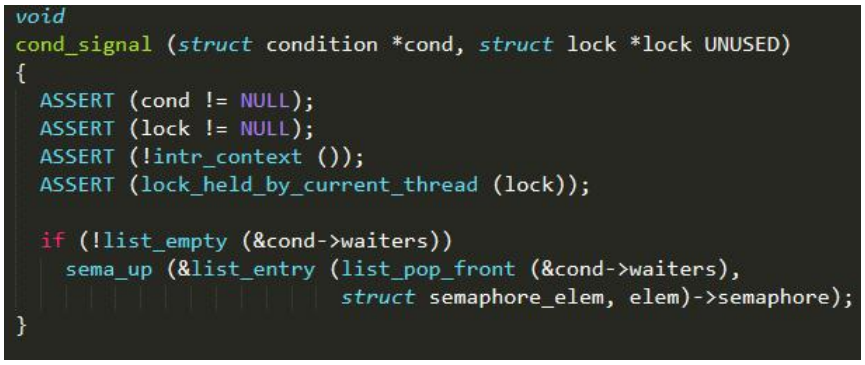
```
/* One semaphore in a list. */
struct semaphore_elem
  {
    struct list_elem elem;              /* List element. */
    struct semaphore semaphore;         /* This semaphore. */
  };
```

semaphore_elem 结构体里定义了两个变量，分别是 elem，semaphore，可见 semaphore_ellem 结构体其实就是一个信号量，当每个线程调用 cond_wait 函数时都会分配一个 semaphore_elem 结构体的变量，从而做到每个信号量对应一个线程。方便对线程进行阻塞和唤醒，使用 p，v 操作即可。



```
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

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

  sema_init (&waiter.semaphore, 0);
  list_push_back (&cond->waiters, &waiter.elem);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

首先给每个线程分配对应的信号量，断言调用 cond_wait 函数的线程是锁 lock 的主人，接着将 waiter 的 semaphore 初始化为 0 保证线程执行的先后顺序。调用 list_push_back 函数将线程对应的信号量插入到 condition 的阻塞队列中，然后在释放掉锁，调用 sema_down 函数把自己阻塞掉，直到这个线程被唤醒，再重新申请获得锁 lock。



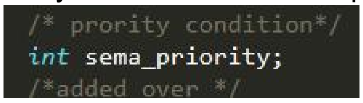
```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters))
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);
}
```

首先判断状态变量的阻塞队列是否为空，如果不为空，就选择队列中的第一个元素也就是第一个信号量进行 v 操作，同时删掉该信号量。

分析 test 中的 priority-condvar 可知，调用 cond_signal 函数的时候，是根据线程的优先级大小顺序唤醒的，优先级高的线程先被唤醒。分析未修改前的代码可知，状态变量的 watiers 链表是由线程对应的信号量组成，并且调用 cond_signal 函数是从 waiters 链表的头部获取信号量唤醒线程的，因此，首先我们需要再信号量的结构体定义中新增加一个变量优先级，用来记录保存对应线程的优先级。其次需要再 cond_signal 函数进行 v 操作前，对 waiters 链表中的信号量按照优先级大小的顺序排好队。

代码修改：

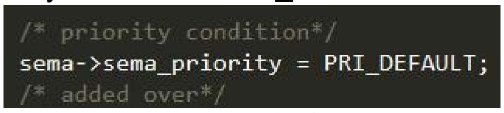
在 synch.h 中的 struct semphore 结构体中新增添一个变量



```
/* prority condition*/
int sema_priority;
/*added over */
```

表示相对应线程的优先级

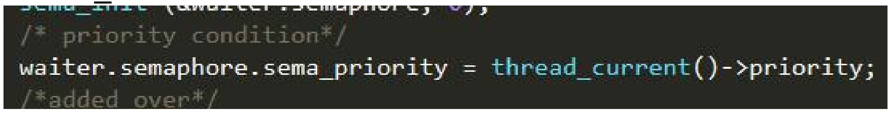
在 synch.c 中的 sema_inti 进行初始化



```
/* priority condition*/
sema->sema_priority = PRI_DEFAULT;
/* added over*/
```

将信号量的优先级设为默认的优先级

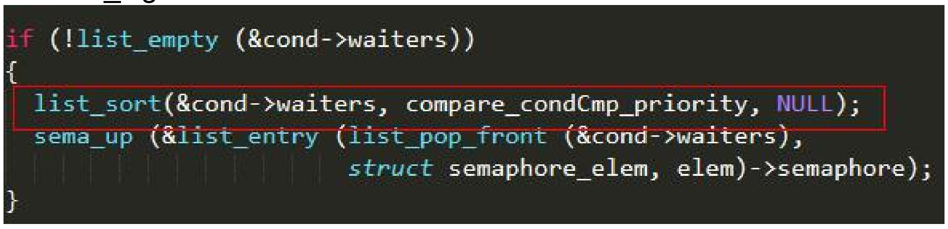
在 cond_wait 中



```
/* priority condition*/
waiter.semaphore.sema_priority = thread_current()->priority;
/*added over*/
```

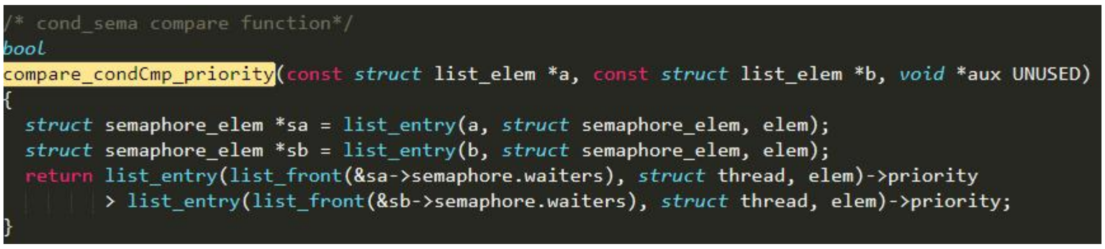
将当前正在 cpu 中运行的线程的优先级赋值给信号量的优先级

在 cond_signal 函数中



```
if (!list_empty (&cond->waiters))
{
  list_sort(&cond->waiters, compare_condCmp_priority, NULL);
  sema_up (&list_entry (list_pop_front (&cond->waiters),
                        struct semaphore_elem, elem)->semaphore);
}
```

在进行 v 操作前，对 waiters 链表进行优先级排序，这里设计到一个新写的函数 compare_condCmp_priority 函数



```
/* cond_sema compare function*/
bool
compare_condCmp_priority(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
  struct semaphore_elem *sa = list_entry(a, struct semaphore_elem, elem);
  struct semaphore_elem *sb = list_entry(b, struct semaphore_elem, elem);
  return list_entry(list_front(&sa->semaphore.waiters), struct thread, elem)->priority
         > list_entry(list_front(&sb->semaphore.waiters), struct thread, elem)->priority;
}
```

-----------------------------------------------------------------------------------------------------------

**伪代码：**

```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
   ........
 If(cond.waiters 链表不为空)
  {
     sort（waiters）  //按优先级大小排好序
     sema_up(waiters 的头元素);
   }
}
```

-----------------------------------------------------------------------------------------------------------

## 多级反馈调度

产生多级反馈调度的原因：当一个线程的优先级很高，并且需要运行很长时间的时候，其他一些优先级较低但是运行的时间很短的线程就必须要等到前面那个线程执行完后才能运行，这样会造成优先级较低的线程饿死。因此，通过考虑 cpu 的吞吐量，线程的 nice 属性，线程近期占用 cpu 的时间等线程运转情况来更新线程的优先级，从而实现动态调度。(参考 ppt)

实现思路：(参考 ppt)
我们需要实现多级反馈调度队列，在 pintos 中线程一共有 64 种优先级，分别是 PRI_MIN(0) 到 PRI_MAX(63),每个优先级会对应一条优先级队列。线程创建时被分配到一个优先级，线程会对应进入相应的优先级队列中，然后从此每隔四个时钟周期就需要更新线程的优先级。

**线程优先级计算：**

$$\text{priority} = PRI\_MAX - \frac{recent\_cpu}{4} - nice * 2$$

recent_cpu 表示的含义：记录线程近期占用 cpu 的情况
nice 表示的含义：该线程对其他线程的友好程度，当 nice 大于 0 时，表示线程愿意降低自己的优先级使得其他线程获得更多的时间片，并且 nice 值越大，其他线程获得的时间片可能会越多。当 nice 小于 0，则效果相反
priortity 是整数，并且是在 0 到 63 的范围。所以当计算的 priority 大于 PRI_MAX 时要降低到 PRI_MAX，当计算的 priority 的值小于 PRI_MIN，要升到 PRI_MIN。
分析 priority 公式可以直到，当前线程占用 cpu 的时间越长的时候，recent_cpu 会越大，此时线程的 priority 会越小直到这个占用 cpu 很久的线程退出 cpu 为止，这样就能是其他线程获得 cpu，从而避免饿死现象。同理一个没有进入 cpu 的线程，也会因为 recent_cpu 为 0，会更快地进入 cpu 中运行。

**线程 recent_cpu 计算：**

$$recent\_cpu = \frac{2 * load\_avg}{2 * load\_avg + 1} * recent\_cpu + nice$$

每隔 1s 钟，需要对所有线程除了空闲线程，更新 recent_cpu.

**线程 load_avg 计算：**

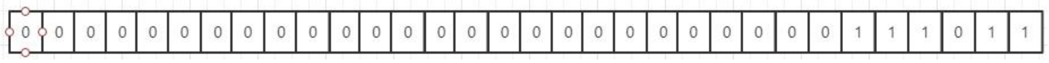
$$load\_avg = \frac{59}{60} * load\_avg + \frac{1}{60} * ready\_threads$$

Ready_threads 指的是就绪队列和运行线程中非空闲状态的线程数。

**浮点数运算：(参考 ppt)**

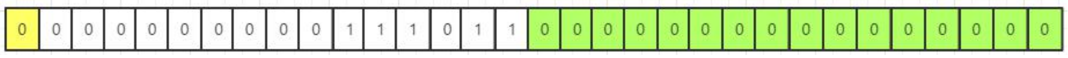
在上述的 3 个公式中，出现了浮点数和浮点数的运算，但是在 pintos 中并不能存放浮点数和进行浮点数运算。因此我们需要实现浮点数的运算。
Pintos 中只有整数，因此我们可以使用 32 位 int 的后 16 位表示小数部分，最高位表示正负，

中间的 15 位表示整数部分，然而这只是我们人为的假定是浮点数，实际上最后的浮点数仍然是整数。
当一个 32 位的整数 x，后面的 16 位当成是小数部分时，此时 x 所表示的实际值为 x/2^16,
如果难以理解的话，可以举 10 进制的例子，比如 1000，如果把 1000 的后两位当成小数位，则 1000 表示的实际值为 1000/10^2=10.00 ， 另 f = 2^16
(绿色表示小数位，黄色表示符号位)
① 有符号整数转化为浮点数
整数 59



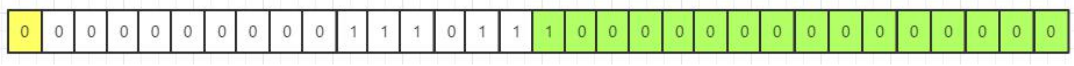
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |

乘以 f(左移 16 位)得到 59.0



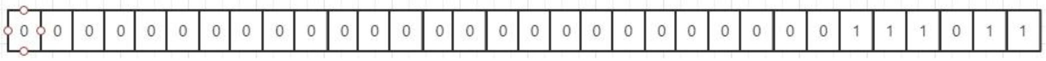
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

② 浮点数转化为整数(取整数部分，直接将小数部分舍弃)
浮点数 59.5



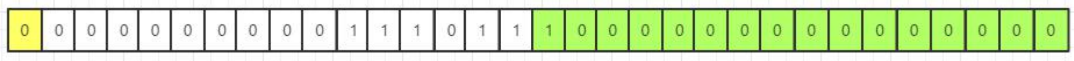
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

除以 f(右移 16 位)得到 59



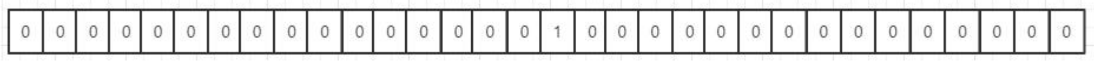
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |

③ 浮点数转化为整数(会进行四舍五入)



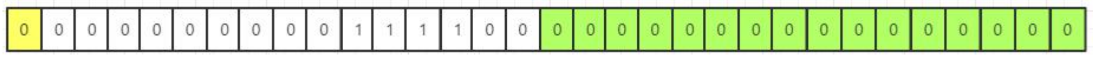
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

＋ (f/2)



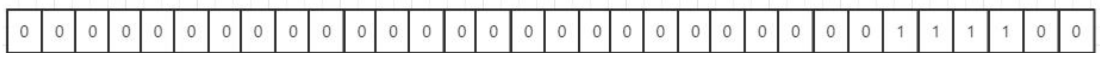
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

＝



| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

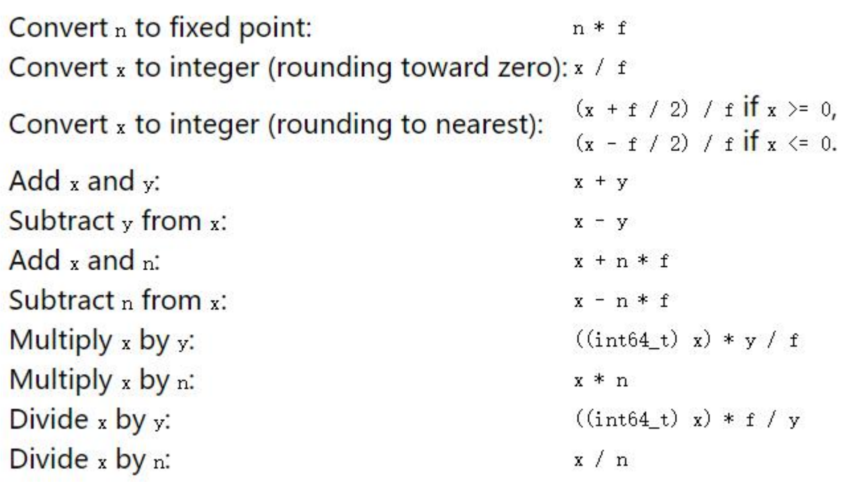
除以 f
=



| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

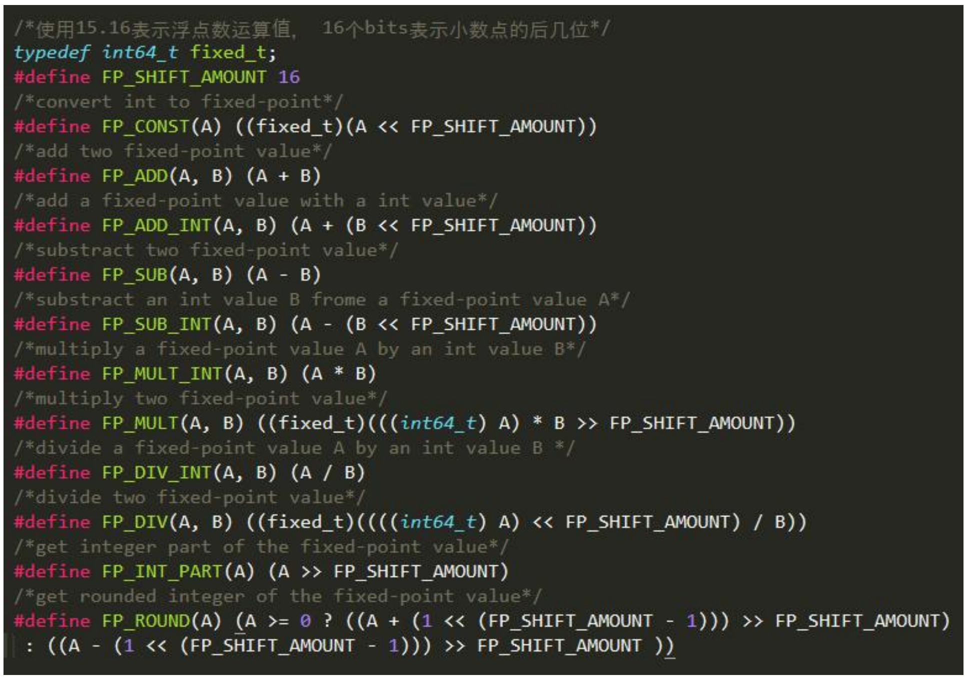
因此当 x>0 时，转化为整数 (x+f/2)/f
当 x<0 时，变成(x-f/2)/f 即可

在 ppt 中的链接资料中归纳了公式：



| | |
|---|---|
| Convert n to fixed point: | n * f |
| Convert x to integer (rounding toward zero): | x / f |
| Convert x to integer (rounding to nearest): | (x + f / 2) / f if x >= 0,<br>(x - f / 2) / f if x <= 0. |
| Add x and y: | x + y |
| Subtract y from x: | x - y |
| Add x and n: | x + n * f |
| Subtract n from x: | x - n * f |
| Multiply x by y: | ((int64_t) x) * y / f |
| Multiply x by n: | x * n |
| Divide x by y: | ((int64_t) x) * f / y |
| Divide x by n: | x / n |

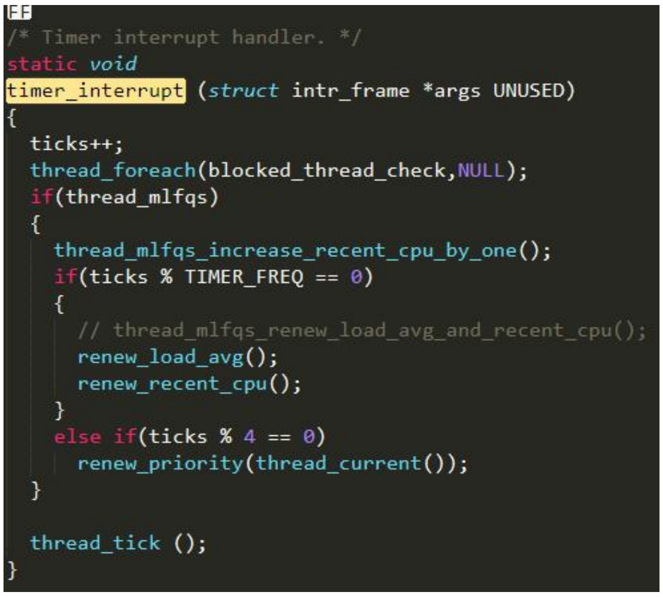
由上述公式中，我们可以新建一个文件 flxed_point.h 定义 pintos 下的浮点数运算



```c
/*使用15.16表示浮点数运算值，  16个bits表示小数点的后几位*/
typedef int64_t fixed_t;
#define FP_SHIFT_AMOUNT 16
/*convert int to fixed-point*/
#define FP_CONST(A) ((fixed_t)(A << FP_SHIFT_AMOUNT))
/*add two fixed-point value*/
#define FP_ADD(A, B) (A + B)
/*add a fixed-point value with a int value*/
#define FP_ADD_INT(A, B) (A + (B << FP_SHIFT_AMOUNT))
/*substract two fixed-point value*/
#define FP_SUB(A, B) (A - B)
/*substract an int value B frome a fixed-point value A*/
#define FP_SUB_INT(A, B) (A - (B << FP_SHIFT_AMOUNT))
/*multiply a fixed-point value A by an int value B*/
#define FP_MULT_INT(A, B) (A * B)
/*multiply two fixed-point value*/
#define FP_MULT(A, B) ((fixed_t)(((int64_t) A) * B >> FP_SHIFT_AMOUNT))
/*divide a fixed-point value A by an int value B */
#define FP_DIV_INT(A, B) (A / B)
/*divide two fixed-point value*/
#define FP_DIV(A, B) ((fixed_t)((((int64_t) A) << FP_SHIFT_AMOUNT) / B))
/*get integer part of the fixed-point value*/
#define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)
/*get rounded integer of the fixed-point value*/
#define FP_ROUND(A) (A >= 0 ? ((A + (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT)
 : ((A - (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT ))
```

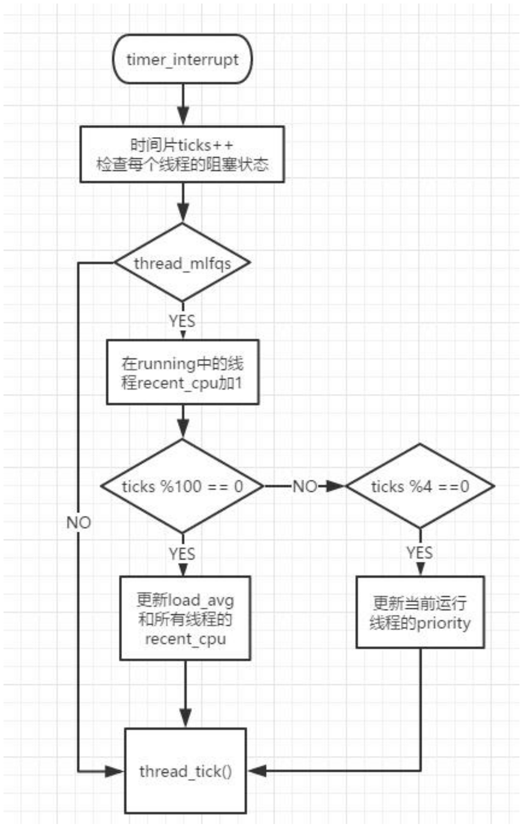
当一个浮点数和整数想加减时，将整数转化为浮点数。
两个浮点数相乘除时，避免上溢，需要拓展成 64 位

实现了基本的浮点运算后，在需要用到浮点数的地方加上头文件 "threads/fixed_point.h"即可
做好准备工作后，更新涉及到的各个变量.(参考 ppt)

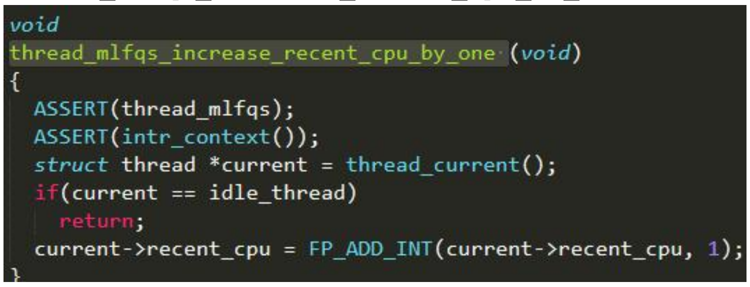


```c
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_foreach(blocked_thread_check,NULL);
  if(thread_mlfqs)
  {
    thread_mlfqs_increase_recent_cpu_by_one();
    if(ticks % TIMER_FREQ == 0)
    {
      // thread_mlfqs_renew_load_avg_and_recent_cpu();
      renew_load_avg();
      renew_recent_cpu();
    }
    else if(ticks % 4 == 0)
      renew_priority(thread_current());
  }

  thread_tick ();
}
```

每个 timer_tick 时刻，就将在 cpu 中运行的线程 recent_cpu 加 1
每过 1s 就更新系统的 load_avg 和所有线程的 recent_cpu
每过 4 个 timer_ticks，就更新一次当前运行线程的优先级
流程图：



thread_mlfqs_increase_recent_cpu_by_one 函数实现



```
void
thread_mlfqs_increase_recent_cpu_by_one (void)
{
  ASSERT(thread_mlfqs);
  ASSERT(intr_context());
  struct thread *current = thread_current();
  if(current == idle_thread)
    return;
  current->recent_cpu = FP_ADD_INT(current->recent_cpu, 1);
}
```

如果当前正在 cpu 中运行的线程不是空闲线程，就将其 recent_cpu 加 1，又因为 recent_cpu 是实数，所以需调用 FP_ADD_INT 函数进行运算

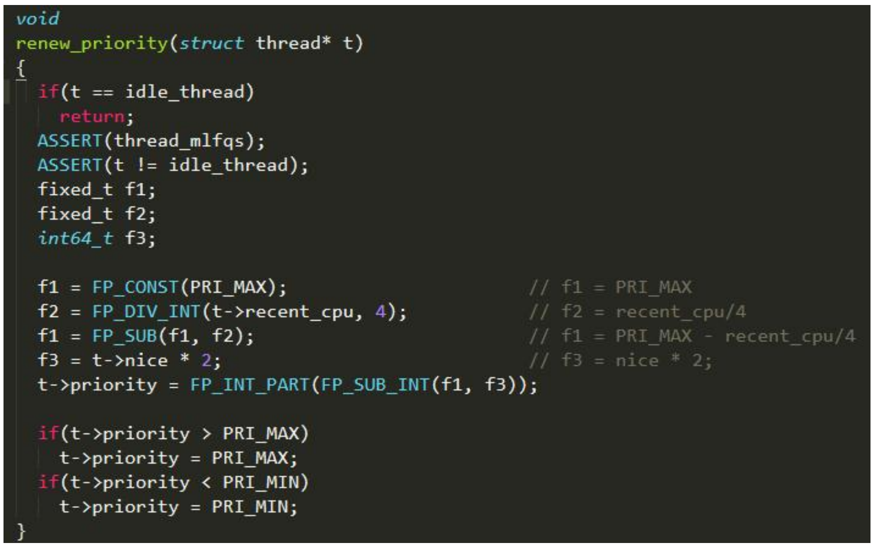
-----------------------------------------------------------------------------------------

伪代码：

```
if(thread_current()为空闲线程)
  Return;
else
  当前线程的 recent_cpu 加 1
```

-----------------------------------------------------------------------------------------

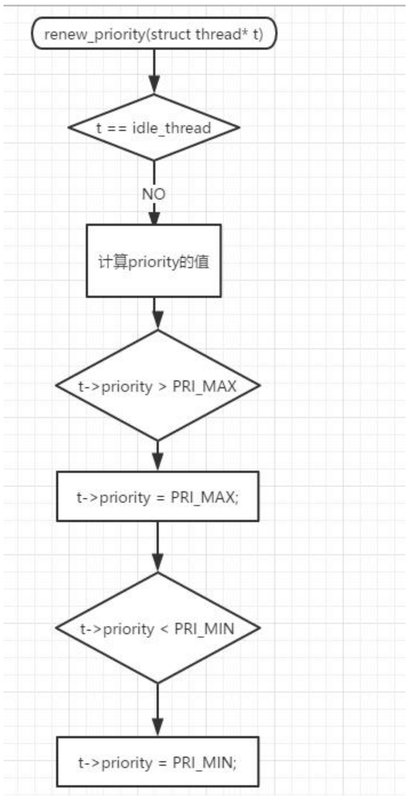
renew_priority 函数实现



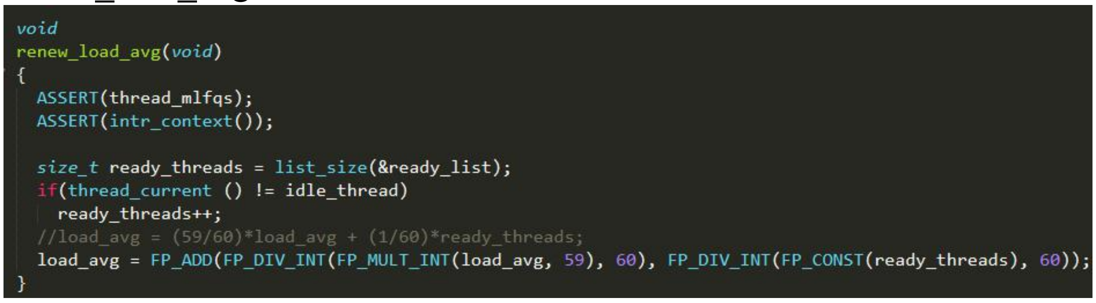
```c
void
renew_priority(struct thread* t)
{
  if(t == idle_thread)
    return;
  ASSERT(thread_mlfqs);
  ASSERT(t != idle_thread);
  fixed_t f1;
  fixed_t f2;
  int64_t f3;

  f1 = FP_CONST(PRI_MAX);                          // f1 = PRI_MAX
  f2 = FP_DIV_INT(t->recent_cpu, 4);               // f2 = recent_cpu/4
  f1 = FP_SUB(f1, f2);                             // f1 = PRI_MAX - recent_cpu/4
  f3 = t->nice * 2;                                // f3 = nice * 2;
  t->priority = FP_INT_PART(FP_SUB_INT(f1, f3));

  if(t->priority > PRI_MAX)
    t->priority = PRI_MAX;
  if(t->priority < PRI_MIN)
    t->priority = PRI_MIN;
}
```

根据公式 priority = PRI_MAX-(recent_cpu/4)-(nice * 2)，利用浮点运算算出 priority，如果线程的 priority 大于 PRI_MAX,就将线程的 Priority 设为 PRI_MAX，如果线程的 priority 小于 PRI_MIN，就将线程的 priority 设为 PRI_MIN.

流程图



## renew_load_avg 函数的实现



```c
void
renew_load_avg(void)
{
  ASSERT(thread_mlfqs);
  ASSERT(intr_context());

  size_t ready_threads = list_size(&ready_list);
  if(thread_current () != idle_thread)
    ready_threads++;
  //load_avg = (59/60)*load_avg + (1/60)*ready_threads;
  load_avg = FP_ADD(FP_DIV_INT(FP_MULT_INT(load_avg, 59), 60), FP_DIV_INT(FP_CONST(ready_threads), 60));
}
```

首先获取就绪队列中线程的元素并记录在 ready_threads 中，如果在 cpu 中运行的线程不为空闲线程的时候，就将 ready_threads 加 1.根据公式 load_avg = (59/60)*load_avg + (1/60)*ready_threads 进行相应的浮点数运算。

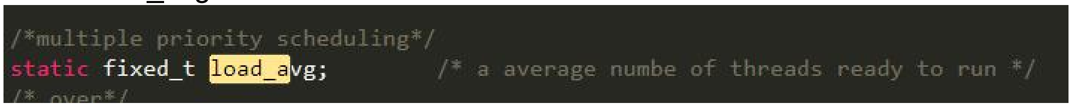
------------------------------------------------------------------------------------------------

伪代码：

```
Size_t ready_threads =  就绪队列中线程个数
If(thread_current() !=  空闲线程)
   read_threads++;
load_avg = (59/60)*load_avg + (1/60)*ready_threads;//  要调用浮点运算的函数。
```

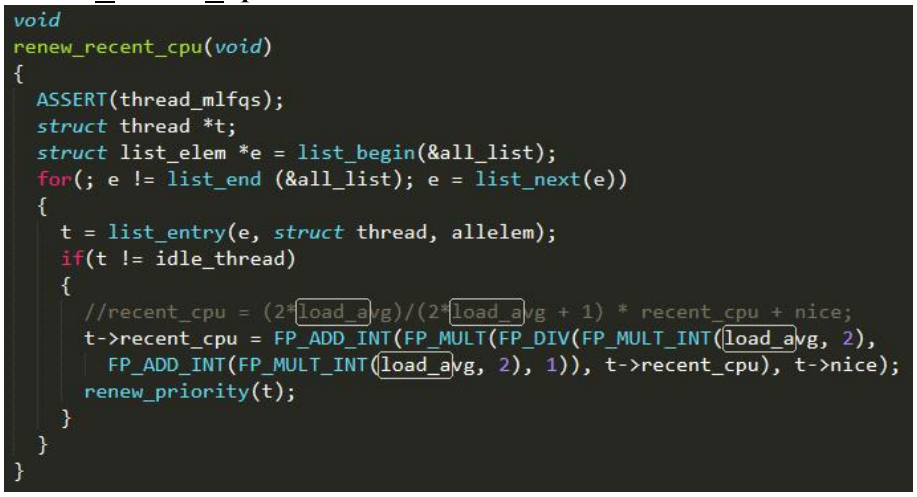
------------------------------------------------------------------------------------------------

注意：load_avg 是全局静态变量



```
/*multiple priority scheduling*/
static fixed_t load_avg;          /* a average numbe of threads ready to run */
/* over*/
```

## renew_recent_cpu 函数的实现



```
void
renew_recent_cpu(void)
{
  ASSERT(thread_mlfqs);
  struct thread *t;
  struct list_elem *e = list_begin(&all_list);
  for(; e != list_end (&all_list); e = list_next(e))
  {
    t = list_entry(e, struct thread, allelem);
    if(t != idle_thread)
    {
      //recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice;
      t->recent_cpu = FP_ADD_INT(FP_MULT(FP_DIV(FP_MULT_INT(load_avg, 2),
        FP_ADD_INT(FP_MULT_INT(load_avg, 2), 1)), t->recent_cpu), t->nice);
      renew_priority(t);
    }
  }
}
```

进入 for 循环，更新系统中除了空闲线程的所有线程的 recent_cpu 和 priority

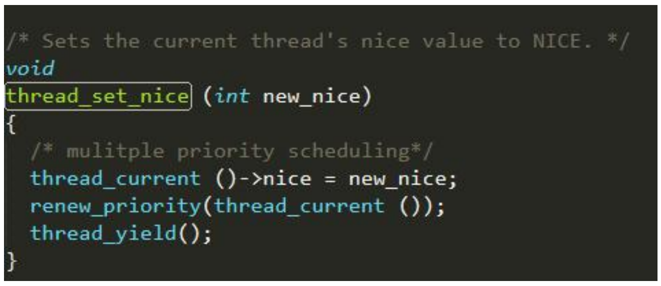
------------------------------------------------------------------------------------------------

伪代码：

```
for(; e!=list_end(&all_list); e = list_next(e))   //遍历一遍 all_list
{
    t =  在 all_list 链表中获取的 elem 对应的线程
    If( t 不为空闲线程)
    {
      //计算 recent_cpu 要进行浮点运算
      T-recent_cpu=recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice;
      Renew_priority(t);        //更新每一个线程的优先级
    }
}
```

------------------------------------------------------------------------------------------------

## thread_set_nice 函数



```
/* Sets the current thread's nice value to NICE. */
void
thread_set_nice (int new_nice)
{
  /* mulitple priority scheduling*/
  thread_current ()->nice = new_nice;
  renew_priority(thread_current ());
  thread_yield();
}
```

将新的 nice 值赋值给线程，并且调用 renew_priority 函数更新线程的优先级。调用 t
hread_yield 函数，将当前线程重新放回就绪队列中，如果当前线程的优先级变低的话，
就会被抢占

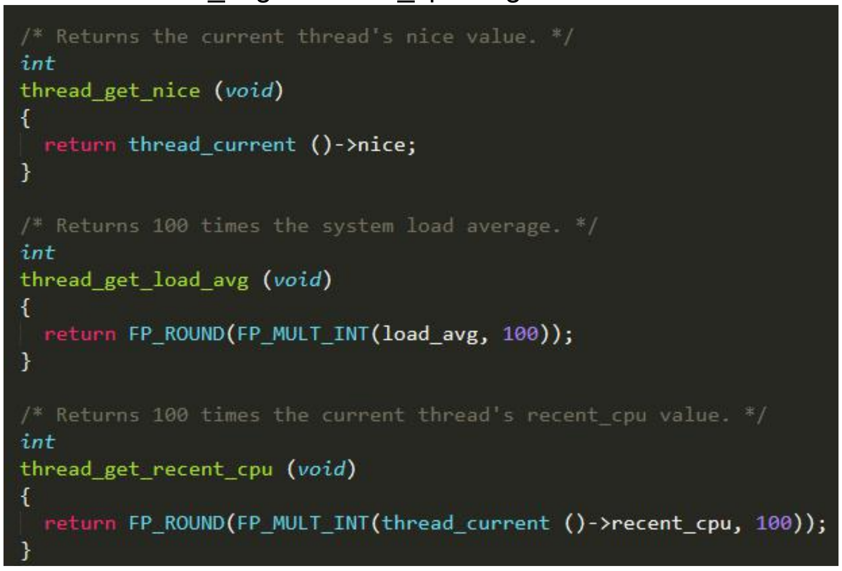
------------------------------------------------------------------------------------------------------------

伪代码：

```
Thread_set_nice(int new_nice)
{
   当前线程的 nice = new_nice
   renew_priority(当前线程)
   Thread_yield();
}
```

------------------------------------------------------------------------------------------------------------

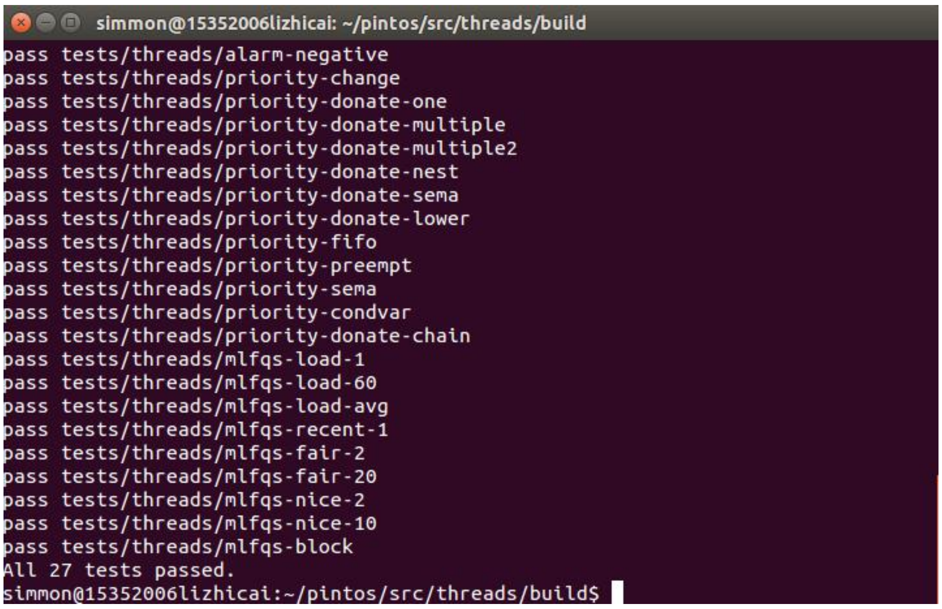
其余 nice，load_avg，  recent_cpu 的 get 函数



```
/* Returns the current thread's nice value. */
int
thread_get_nice (void)
{
  return thread_current ()->nice;
}

/* Returns 100 times the system load average. */
int
thread_get_load_avg (void)
{
  return FP_ROUND(FP_MULT_INT(load_avg, 100));
}

/* Returns 100 times the current thread's recent_cpu value. */
int
thread_get_recent_cpu (void)
{
  return FP_ROUND(FP_MULT_INT(thread_current ()->recent_cpu, 100));
}
```

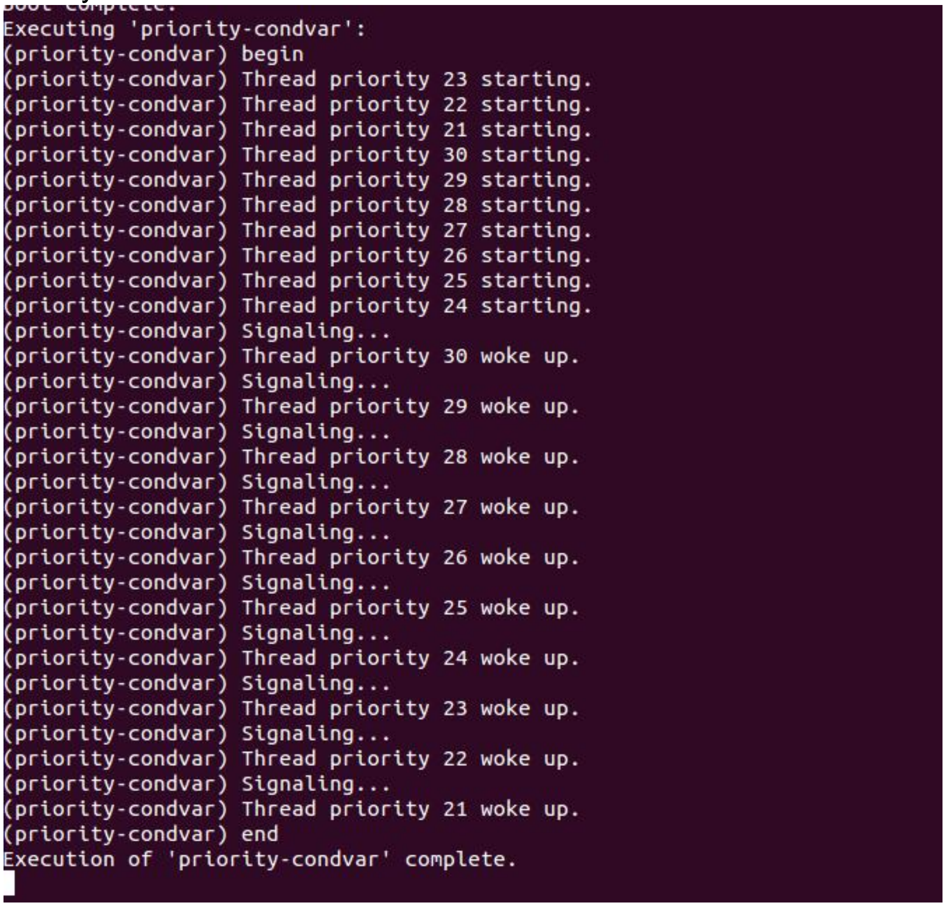
根据测试样例，我们需要将 load_avg 和 recent_cpu 乘以 100，并且因为 load_avg 和
 recent_cpu 都是浮点数，所以要调用浮点数运算的函数
最后，在所有涉及到优先级捐赠的地方加上判断语句 if(!thread_mlfqs)并括起来。

## 3. 实验结果



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

Priority-condvar 的结果



```
Boot complete.
Executing 'priority-condvar':
(priority-condvar) begin
(priority-condvar) Thread priority 23 starting.
(priority-condvar) Thread priority 22 starting.
(priority-condvar) Thread priority 21 starting.
(priority-condvar) Thread priority 30 starting.
(priority-condvar) Thread priority 29 starting.
(priority-condvar) Thread priority 28 starting.
(priority-condvar) Thread priority 27 starting.
(priority-condvar) Thread priority 26 starting.
(priority-condvar) Thread priority 25 starting.
(priority-condvar) Thread priority 24 starting.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 30 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 29 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 28 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 27 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 26 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 25 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 24 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 23 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 22 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 21 woke up.
(priority-condvar) end
Execution of 'priority-condvar' complete.
```

## 4. 回答问题（15 分）

➢ 1. 信号量，锁，条件变量的异同点？

答：相同点：常用于线程间的同步，锁是特殊的信号量，条件变量结合了信号量和锁

不同点：信号量中的 value 的初始值可以不为 1，而锁的 semaphore 的初始值为 1，也就①是说信号量用于多线程多任务同步的，在信号量中，可以有多个线程进入临界区中，当一个线程完成了某一个任务的时候，就通过信号量告诉其他的线程，其他线程去做一个动

作。比如我们熟悉的生产者消费者问题，当生产了一个东西，信号量的 value 值就加 1，当消费者消耗了一个东西，信号量的 value 值就减 1，当信号量的 value 小于等于 0 的时候，消费者还进行消费的话，就会被阻塞。
②锁是用在多线程多任务互斥的，当一个线程占有了某一个资源时，其他线程是无法访问的，只有当这个线程用完这个资源时，其他线程才能访问，一般用在对全局变量的访问，当一个线程要访问一个全局变量之前会进行加锁避免其他线程访问，访问完后释放锁，这时其他线程才能够访问这个全局变量
③条件变量结合了锁和信号量两者的优点。举一个例子：因为锁是死的，当一个线程 A 拿到锁，但是这个线程因为某一个条件为假而进入了一个 while 循环，一直在等这个条件变为真，才能退出循环，然而此时因为它占有锁，所以其他需要这把锁的线程也要陪它一起等条件变为真。这样其实是很浪费 cpu 的处理时间的。因此就出现了条件变量，当线程 A 发现条件为假的时候，就释放掉锁，然后通过 sema_down 阻塞自己，这样其他线程可以获得这把锁，过了一段时间，当条件变为真的时候，就通过 sema_up 唤醒线程 A，线程 A 又拿到锁，就可以继续执行了。

- 2. 实现多级反馈调度为什么要注释掉优先级捐赠？
答：从优先级的角度来说，实现多级反馈调度的时候，如果一个优先级低的线程 A 占有锁，随着线程 A 的执行，该线程的 recent_cpu 会增大从而导致优先级降低，如果没注释掉优先级捐赠，当优先级高的线程 B 申请锁的时候，会把自己的优先级捐赠给线程 A，此时线程 A 的优先级会变高，破坏了多级反馈调度的机制，所以在实现多级反馈调度的时候要注释掉优先级捐赠。

- 3. 多级反馈调度为什么能避免饥饿现象？
答：因为线程的优先级是动态变化的,根据公式 priority = PRI_MAX-(recent_cpu/4)-(nice * 2)可知当前占用 cpu 时间越长的线程的优先级会不断降低到当前线程退出 cpu 为止，并且由于 recent_cpu 表示的是在过去 1 分钟内该线程占用 cpu 的时间，所以上述线程让出 cpu 后，不会马上恢复优先级进行抢占。这时那些优先级相对较低但是运行时间很短的线程就可以获得 cpu，运行下去。从而避免了因为一个优先级很高但运行时间很久的线程长时间占用 cpu 而导致后面的线程一直进不了 cpu 的情况，也就是避免了饥饿现象

## 5. 实验感想

这次实验中主要遇到的问题是 pintos 下的浮点运算处理，因为我的二进制运算学得不好，所以出现有关二进制运算的题，我就会很懵逼，一开始我都不明白>>和<<的区别和含义，所以在理解将整数转化浮点数和浮点数运算那里花费了很多时间，其次就是 condition 条件变量的问题，我个人感觉有时候我太过于分析 test 钻一些牛角尖了，反而我不懂最根本的东西，实现信号量和锁的目的是什么。并且我觉得这些 test 样例太少了，有些粗暴。比如说和条件变量有关的那个 test 就直接调用 cond_wait 函数，并没有体现出条件变量的真正作用。但是 ppt 上谈到了条件变量相关的知识，所以我一开始看 test 就很懵逼，就觉得不就是在唤醒前排个序就好了吗，怎么 ppt 还说这么多知识点，后来自己上网查才有些明白条件变量的作用。最后就是在写函数的时候，编译的时候一开始遇到函数重定义的问题，在这个问题上纠结了一个晚上，后来问了同学才知道，在头文件那里出了问题。