中山大学移动信息工程学院本科生实验报告

(2017 学年春季学期)

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

年级+班级	1506	专业（方向）	移动信息工程
学号	15352116	姓名	洪子淇
电话	13726205766	Email	1102229410@qq.com
开始日期	2017.5.21	完成日期	2017.5.25

1. 实验目的

① 理解条件变量的含义以及作用

② 理解多级反馈队列优先级变化的机制以及实现多级反馈队列的调度

③ 区分信号量，锁以及条件变量

④ 利用整型实现浮点型运算

2. 实验过程

2.1 Test 分析

◆ **mlfqs-block 测试样例**

✧ 测试目的

测试在多级反馈机制下，线程占有 CPU 的时候优先级的更改以及线程在锁释放的时候能否根据优先级顺序执行。

✧ 过程分析

① test 线程优先级为 PRI_DEFAULT，首先 test 线程申请一把锁 lock，并创建了优先级为 PRI_DEFAULT 的 block 线程，由于优先级不高于 test 线程，test 线程继续运行，进入休眠状态，休眠时间为 25s，让出 CPU。block 线程获得 CPU，进入 block_thread 函数，然后空

转了 20s，一直占用 CPU，此时 block 线程的优先级会下降，然后申请锁 lock，由于该锁被 test 线程所占有，block 线程会被阻塞。block 线程被阻塞之后，test 线程还在休眠，那么还有 5s 时间 CPU 处于空闲状态（具体指运行队列和就绪队列都为空，ready_threads = 0），系统负载（load_avg）下降，recent_cpu 值下降，根据公式如下，两个线程的优先级不断被增大约为 PRI_MAX。

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)

test_mlfqs_block 函数：

```
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);
```

获得锁

休眠25秒

block_thread 函数：

```
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);
```

优先级下降

申请锁，被阻塞

② 5s 过后，block 线程被阻塞，test 线程被唤醒获得 CPU，此时两个线程的优先级都为 PRI_MAX，test 线程继续运行，空转了 5s，由于一直占有 CPU，recent_cpu 值会增大，所以 test 线程的优先级会下降，一旦 test 线程释放锁之后，唤醒 block 线程， block 线程由于优先级高于 test 线程会立马抢占 CPU。进入 block_thread 函数，输出 msg 语句“…got it”之后 block_thread 函数运行完，block 线程结束其生

命周期，test 线程因此获得 CPU 继续运行。

test_mlfqs_block 函数：

```
msg ("Main thread spinning for 5 seconds...");
start_time = timer_ticks ();
while (timer_elapsed (start_time) < 5 * TIMER_FREQ)
  continue;          空转5秒，优先级下降

msg ("Main thread releasing lock.");
lock_release (&lock);   释放锁，block线程抢占CPU
```

block_thread 函数：

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

test_mlfqs_block 函数：

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

✧ 结果分析

测试结果输出 msg 顺序和过程分析输出 msg 顺序一致，当线程一直占有 CPU 的时候能够正确地修改优先级，使得被阻塞的线程可以优先运行，测试通过。

```
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.

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

◆ mlfqs-load-1 测试样例

✧ 测试目的

测试单个线程在运行的时候 load_avg 是否可以在指定的时间（38~45s）内增长到 0.5，在 CPU 空闲的时候 load_avg 是否可以在制定的时间内

（10s）衰减到 0.5 以下。

✧ 过程分析

① 首先设置 test 线程的开始时间，然后进入 for 循环，在 for 循环中不断更新 load_avg 的值，当运行时间在 45 秒内，load_avg 大于 1 或者当运行时间超过 45 秒 load_avg 还不到 0.5 的时候失败输出 fail 语句，当 load_avg 能够在 45 秒内增加到 0.5~1.0 之间的时候，load_avg 增长速度符合要求退出 for 循环。

```
start_time = timer_ticks ();
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在45内增长到0.5~1.0之间，符合要求

② 接着进入 if 判断条件，如果 load_avg 在 for 循环增长过快，即在 38 秒之内增长到了 0.5，也是不符合增长的要求，输出 fail 语句。{结合 for 循环一起判断 load_avg 的增长要求就是，当单线程在运行的时候，load_avg 需要在 38 秒到 45 秒之间增长到 0.5 到 1.0 之间。}然后线程进入休眠状态，休眠时间为 10 秒。10 秒之后获得 load_avg 的值，如果此时 load_avg 小于 0，衰减过快，不符合要求并输出 fail 语句，若是 load_avg 大于 0.5，衰减过慢，也不符合要求输出 fail 语句。当 load_avg 在 0~0.5 之间的时候，衰减速度符合要求并输出 msg 语句。{load_avg 的衰减要求就是，当 CPU 处于空闲的时候（具体指就绪队列和运行队列都为空），load_avg 需要在 10 秒之内从 0.5 衰减到小于

0.5 且大于 0 之间}

```
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 ();
```

增长符合要求

衰减符合要求

✧ 结果分析

由输出语句分析可知，load_avg 能够在 38~45 秒之内增长到 0.5，也能在 10 秒之内从 0.5 衰减到小于 0.5 到大于 0 之间，测试通过。

```
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.

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

◆ mlfqs-load-60 测试样例

✧ 测试目的

通过测试 60 个线程，每个线程休眠 10 秒之后，再空转 60 秒，然后再睡眠 60 秒，每两秒输出 load_avg，看 load_avg 的增长和衰减是否符合要求。

✧ 过程分析

① 首先设置了 test 线程的开始时间点，然后进入了 for 循环创建名为

load i 的 THREAD_CNT 个优先级为 PRI_DEFAULT 的子线程，本来每一个线程的优先级都和 test 线程的优先级一样，可是由于 test 线程一直占有 CPU，优先级会下降，然后在创建子线程的时候，子线程会抢占 CPU，进入 load_thread 函数，发现子线程的 nice 值被设置得很高，所以子线程的优先级下降的幅度会比 test 线程大的多。即每次创建线程的时候子线程都会抢占 CPU，抢占完之后进入休眠状态（休眠时间为 10 秒）主动让出 CPU 给 test 线程，然后当优先级更新之后会由于 nice 值所以优先级会一直低于 test 线程，所以可以看成 test 线程一直在占有 CPU。

test_mlfqs_load_60 函数：

```
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);
  }
```

优先级高于test线程，抢占CPU

load_thread 函数：

```
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));
```

休眠时间10s

空转时间60s

再次休眠60s

进入休眠状态，让出CPU，并且优先级变低

② 创建子线程完成之后进入下一个 for 循环，第一次循环 test 线程休眠（10s-创建线程的时间）s，即有 （10s-创建线程的时间）秒的时间，所有线程都在休眠状态，load_avg 会在这个时间衰减。（10 秒-创建线程的时间）s 之后所有线程被唤醒，此时子线程的优先级大致是一样的，而 test 线程的优先级最高继续占有 cpu，输出 0s 之后（这里的 0 秒距离开始的时候间隔了 10s）的 load_avg，然后

进入下一次循环，下面的每次循环中 test 线程都会休眠 2s，在 2s 的过程中，子线程会获得 cpu，然后进入了空转的状态，直到 spin_time 的到来（即空转时间为 60s）。根据 load_avg 的计算公式：

load_avg = (59 / 60) * load_avg + (1 / 60) * ready_threads

ready_threads 等于 THREAD_CNT，会发现此时的 load_avg 不断增加。

两秒之后，test 线程会再次被唤醒抢占 cpu 输出 load_avg。

③ 每次 test 线程让出 cpu 而子线程在空转状态的时候 load_avg 都会增加，直到子线程结束了空转状态，即 60 秒之后，（第 30 次循环之后）当 test 线程再次休眠 2s，让出 cpu 的时候，此刻子线程全部进入休眠状态直到 exit_time 的到来，然后子线程会同步结束，ready_threads 的值由 60 跌至 0，即子线程不会再影响到系统的负载，所以 60s 之后 load_avg 开始降低。

test_mlfqs_load_60 函数：

```
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);
  }
}
```

load_thread 函数：

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

✧ 结果分析

测试样例示范的部分输出结果和实际样例输出结果比较，可以发现误

差不大，而且在 60s 时刻内，load_avg 处于增长状态，60s 之后 load_avg 处于衰减状态，与过程分析一致，测试通过。

```
After 40 seconds, load average=29.88.
After 42 seconds, load average=30.87.
After 44 seconds, load average=31.84.
After 46 seconds, load average=32.77.
After 48 seconds, load average=33.67.
After 50 seconds, load average=34.54.
After 52 seconds, load average=35.38.
After 54 seconds, load average=36.19.
After 56 seconds, load average=36.98.
After 58 seconds, load average=37.74.
After 60 seconds, load average=37.48.
After 62 seconds, load average=36.24.
After 64 seconds, load average=35.04.
After 66 seconds, load average=33.88.
After 68 seconds, load average=32.76.
After 70 seconds, load average=31.68.
After 72 seconds, load average=30.63.
After 74 seconds, load average=29.62.
After 76 seconds, load average=28.64.
After 78 seconds, load average=27.69.
After 80 seconds, load average=26.78.
```

```
(mlfqs-load-60) After 40 seconds, load average=29.39.
(mlfqs-load-60) After 42 seconds, load average=30.40.
(mlfqs-load-60) After 44 seconds, load average=31.38.
(mlfqs-load-60) After 46 seconds, load average=32.33.
(mlfqs-load-60) After 48 seconds, load average=33.24.
(mlfqs-load-60) After 50 seconds, load average=34.12.
(mlfqs-load-60) After 52 seconds, load average=34.98.
(mlfqs-load-60) After 54 seconds, load average=35.81.
(mlfqs-load-60) After 56 seconds, load average=36.61.
(mlfqs-load-60) After 58 seconds, load average=37.38.
(mlfqs-load-60) After 60 seconds, load average=38.13.
(mlfqs-load-60) After 62 seconds, load average=36.87.
(mlfqs-load-60) After 64 seconds, load average=35.65.
(mlfqs-load-60) After 66 seconds, load average=34.47.
(mlfqs-load-60) After 68 seconds, load average=33.33.
(mlfqs-load-60) After 70 seconds, load average=32.23.
(mlfqs-load-60) After 72 seconds, load average=31.16.
(mlfqs-load-60) After 74 seconds, load average=30.13.
(mlfqs-load-60) After 76 seconds, load average=29.14.
(mlfqs-load-60) After 78 seconds, load average=28.17.
(mlfqs-load-60) After 80 seconds, load average=27.24.
```

◆ mlfqs-load-avg 测试样例

✧ 测试目的

该测试的目的和上一个测试即 mlfqs-load-60 目的差不多，也是观察

load_avg 的增长和衰减是否符合要求。

✧ 过程分析

① 首先设置了初始时间进入 for 循环创建子线程，在 load_thread 函数传入了一个参数 seq_no_，该参数控制了每个线程的休眠的时间，根据代码分析可知，越早创建的线程休眠时间越少，被唤醒也越早（即创建的子线程会按照创建的顺序被唤醒）。与 mlfqs-load-60 测试不同的是，mlfqs-load-60 通过设置子线程的 nice 值降低子线程的优先级；这个测试则是通过降低 test 线程的 nice 值来提高 test 线程的优先级，从而使得 test 线程的优先级高于子线程。

test_mlfqs_load_avg 函数：

```
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);
```

test线程优先级一直高于子线程

load_thread 函数：

```
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));
```

② 创建完线程之后，同样进入 for 循环，函数内容和上一个测试一样；主要是子线程的状态不一样，根据上面分析可知，每个子线程会间隔 1s 被唤醒进入就绪队列，即在开始后的 10s 之后，就绪队列在 THREAD_CNTs 内每秒增加 1 。而每个被唤醒的子线程

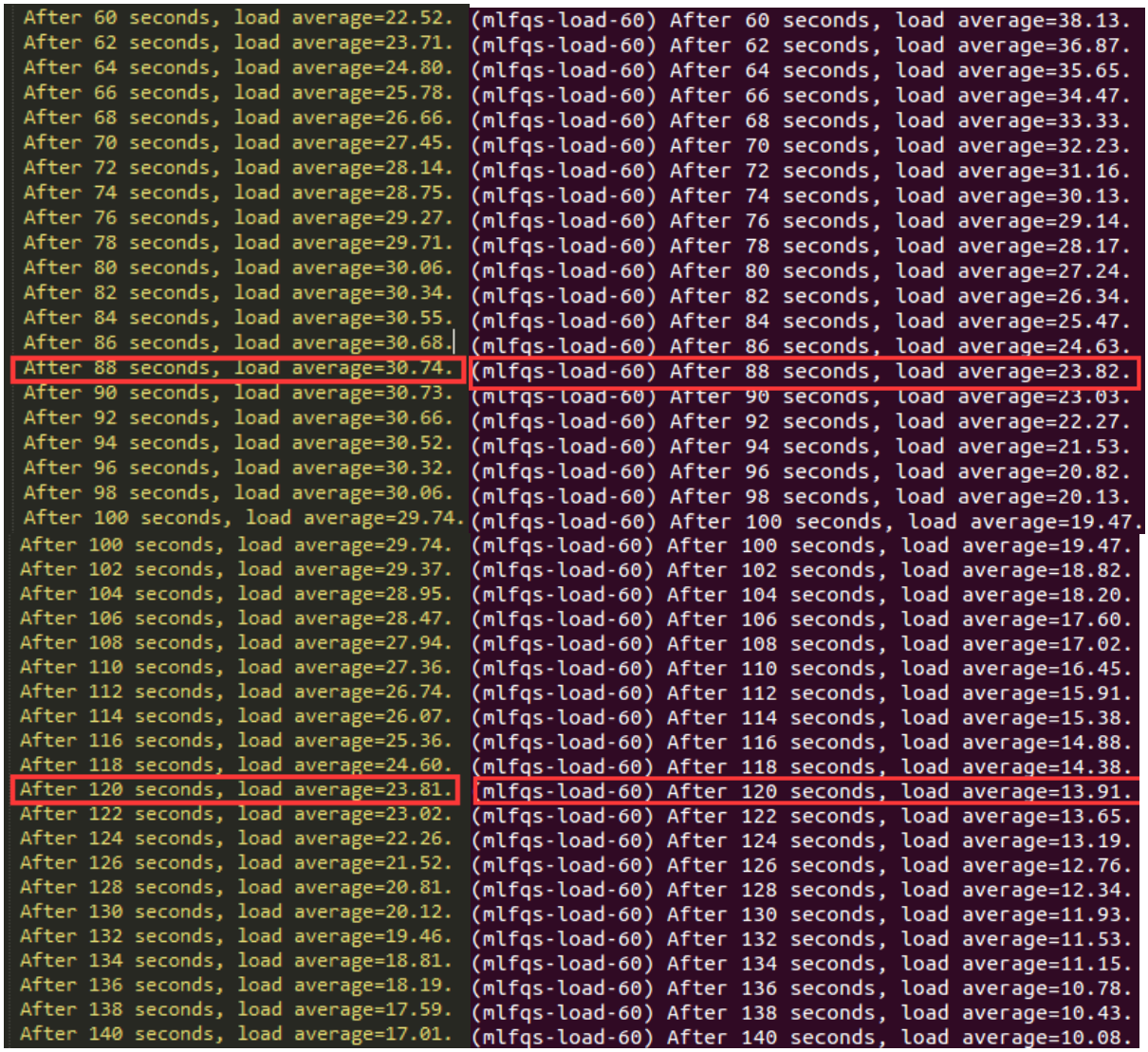
只要拿到 cpu 之后就会进入空转的状态，空转时间都为 60s，再度进入休眠，当 exit_time 时刻到来，结束休眠的同时也结束该线程的生命周期。总体来说就是子线程隔 1s 被唤醒，也隔 1s 再次进入休眠，不过子线程都是同步结束的。

因此，在子线程依次被唤醒进入就绪队列的时候，系统的负载是不断地增加的，随着时间的增加，比较早被唤醒的线程结束了空转的状态再次进入休眠，系统的负载又会下降。所以在测试过程中 load_avg 的值应该是先增长后降低。而且当所有的线程都进入休眠状态后，load_avg 衰减的速度会明显提高。

```
int seq_no = (int) seq_no_;
int sleep_time = TIMER_FREQ * (10 + seq_no);空转时间段相同
int spin_time = sleep_time + TIMER_FREQ * THREAD_CNT;
int exit_time = TIMER_FREQ * (THREAD_CNT * 2);
   结束时间点相同
timer_sleep (sleep_time - timer_elapsed (start_time));
while (timer_elapsed (start_time) < spin_time)
  continue;
timer_sleep (exit_time - timer_elapsed (start_time));
```

✧ 结果分析

和 mlfqs-load-60 测试对比，由于 mlfqs-load-60 测试的子线程是同步唤醒并且同步空转同步再次休眠的，而这个测试不是同步休眠也不是同步唤醒和同步再次休眠的，所以对比系统负载来说，该测试的 load_avg 的增长速度和衰减速度都会比上一个测试要慢，分割点也不一致。对比示范结果，实际结果的误差值不大。而且可以发现 90s 之前 load_avg 处于增长状态，90s 之后 load_avg 处于衰减状态，在所有的线程都进入了休眠状态，即 120s 之后会发现 load_avg 下降速度加快，符合过程分析，测试通过。



◆ **mlfqs-recent-1 测试样例**

✧ 测试目的

该测试主要是用来极端单线程的时候 recent_cpu 和 load_avg 的计算是否符合要求。

✧ 过程分析

① 首先 test 线程进入休眠状态，将自身的 recent_cpu 的值降低到 7.0 以下。

```
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);
```

② 然后设置了开始的时间，并且进入 for 循环，每隔 2s 就会获得新的 recent_cpu 以及 load_avg 的值并且输出来，当循环时间超过 180s 时退出循环。

```
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;
  }
```

✧ 结果分析

根据过程分析，首先 test 线程的 recent_cpu 会降到 7.0 以下，以这个时间点为开始，之后由于一直占用 cpu，所以 recent_cpu 和 load_avg 也会一直增长。与示范的 recent_cpu 和 load_avg 的增长比较，实际的 recent_cpu 和 load_avg 增长的速度符合要求。测试通过。

```
After 2 seconds, recent_cpu is 6.40, load_avg is 0.03.
After 4 seconds, recent_cpu is 12.60, load_avg is 0.07.
After 6 seconds, recent_cpu is 18.61, load_avg is 0.10.
After 8 seconds, recent_cpu is 24.44, load_avg is 0.13.
After 10 seconds, recent_cpu is 30.08, load_avg is 0.15.
After 12 seconds, recent_cpu is 35.54, load_avg is 0.18.
After 14 seconds, recent_cpu is 40.83, load_avg is 0.21.
After 16 seconds, recent_cpu is 45.96, load_avg is 0.24.
After 18 seconds, recent_cpu is 50.92, load_avg is 0.26.
After 20 seconds, recent_cpu is 55.73, load_avg is 0.29.
After 22 seconds, recent_cpu is 60.39, load_avg is 0.31.
After 24 seconds, recent_cpu is 64.90, load_avg is 0.33.
After 26 seconds, recent_cpu is 69.27, load_avg is 0.35.
After 28 seconds, recent_cpu is 73.50, load_avg is 0.38.
After 30 seconds, recent_cpu is 77.60, load_avg is 0.40.
```

```
After 150 seconds, recent_cpu is 183.41, load_avg is 0.92.
After 152 seconds, recent_cpu is 183.96, load_avg is 0.92.
After 154 seconds, recent_cpu is 184.49, load_avg is 0.92.
After 156 seconds, recent_cpu is 185.00, load_avg is 0.93.
After 158 seconds, recent_cpu is 185.49, load_avg is 0.93.
After 160 seconds, recent_cpu is 185.97, load_avg is 0.93.
After 162 seconds, recent_cpu is 186.43, load_avg is 0.93.
After 164 seconds, recent_cpu is 186.88, load_avg is 0.94.
After 166 seconds, recent_cpu is 187.31, load_avg is 0.94.
After 168 seconds, recent_cpu is 187.73, load_avg is 0.94.
After 170 seconds, recent_cpu is 188.14, load_avg is 0.94.
After 172 seconds, recent_cpu is 188.53, load_avg is 0.94.
After 174 seconds, recent_cpu is 188.91, load_avg is 0.95.
After 176 seconds, recent_cpu is 189.27, load_avg is 0.95.
After 178 seconds, recent_cpu is 189.63, load_avg is 0.95.
After 180 seconds, recent_cpu is 189.97, load_avg is 0.95.
```

```
(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 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.

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

◆ mlfqs-fair-2 测试样例

✧ 测试目的

测试两个 nice 值同为 0 的线程获得的 ticks 的统计特性。

✧ 过程分析

测试 mlfqs-fair-2，mlfqs-fair-20，mlfqs-nice-2，mlfqs-nice-10 执行的都是 test_mlfqs_fair 函数，只不过传入参数不一致。在这四个测试中首先定义了一个新的结构体 thread_info（包括 start_time，线程开始时间；tick_count，线程持有 CPU 的时间；nice，线程初始的 nice 值）

① test_mlfqs_fair 传入参数 2，0，0。首先将 test 线程的 nice 值设置为-20，然后进入 for 循环创建了两个 nice 值为 0 优先级为 PRI_DEFAULT 的子线程,创建完毕之后 test 线程输出 msg 语句，然后休眠 40s

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

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

nice_step = 0,nice = 0

② test 线程休眠，子线程获得 CPU，进入 load_thread 函数，首先设置了三个时间点：sleep_time，线程苏醒时刻；spin_time，线程停止空转时刻；last_time，上一个记录的时刻。然后将结构体的 nice 值设置到线程的 nice 值之后，进入休眠状态（保证两个线程同步苏醒，而且优先级一致）。然后子线程进入 30s 的空转状态。每次空转 tick_count++，由于这两个子线程同步唤醒，而且 nice 值一样，那么优先级也几乎一样，因此每个子线程在每个 tick

获得 CPU 的概率是相等的，都为 1/2。

```
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;
    }
}
```

实现同步唤醒

③ 30s 之后，两个子线程结束，test 线程被唤醒，输出每个线程获得的 ticks 大小

```
for (i = 0; i < thread_cnt; i++)
  msg ("Thread %d received %d ticks.", i, info[i].tick_count);
```

✧ 结果分析

由结果可知,每个子线程获得一半的时间,即获得tick的概率为1/2,与过程分析一致，测试通过。

```
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 1500 ticks.
(mlfqs-fair-2) Thread 1 received 1501 ticks.
(mlfqs-fair-2) end
Execution of 'mlfqs-fair-2' complete.

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

◆ **mlfqs-fair-20 测试样例**

✧ 测试目的

该测试主要时测试 20 个 nice 值同为 0 的线程获得的 ticks 的统计特性。

✧ 过程分析

与上一个测试样例 mlfqs-fair-2 实现流程完全一致，只不过创建子线程的数目由 2 变为了 20，nice 值也一样，所以在同步唤醒的时候，这 20 个子线程的优先级也一样，每个子线程获得 tick 的概率一样，为 1/20。

✧ 结果分析

根据输出结果分析，每个线程获得的 ticks 的数目之间相差不大，而且约为总 ticks 的 1/20，与过程分析一致，测试通过。

```
Executing 'mlfqs-fair-20':
(mlfqs-fair-20) begin
(mlfqs-fair-20) Starting 20 threads...
(mlfqs-fair-20) Starting threads took 50 ticks.
(mlfqs-fair-20) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-fair-20) Thread 0 received 152 ticks.
(mlfqs-fair-20) Thread 1 received 153 ticks.
(mlfqs-fair-20) Thread 2 received 152 ticks.
(mlfqs-fair-20) Thread 3 received 149 ticks.
(mlfqs-fair-20) Thread 4 received 148 ticks.
(mlfqs-fair-20) Thread 5 received 149 ticks.
(mlfqs-fair-20) Thread 6 received 148 ticks.
(mlfqs-fair-20) Thread 7 received 148 ticks.
(mlfqs-fair-20) Thread 8 received 152 ticks.
(mlfqs-fair-20) Thread 9 received 153 ticks.
(mlfqs-fair-20) Thread 10 received 153 ticks.
(mlfqs-fair-20) Thread 11 received 153 ticks.
(mlfqs-fair-20) Thread 12 received 153 ticks.
(mlfqs-fair-20) Thread 13 received 148 ticks.
(mlfqs-fair-20) Thread 14 received 149 ticks.
(mlfqs-fair-20) Thread 15 received 148 ticks.
(mlfqs-fair-20) Thread 16 received 151 ticks.
(mlfqs-fair-20) Thread 17 received 152 ticks.
(mlfqs-fair-20) Thread 18 received 148 ticks.
(mlfqs-fair-20) Thread 19 received 149 ticks.
(mlfqs-fair-20) end
Execution of 'mlfqs-fair-20' complete.

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

◆ **mlfqs-nice-2 测试样例**

✧ 测试目的

测试 2 个具有不同 nice 值的线程获得 ticks 的统计特性。

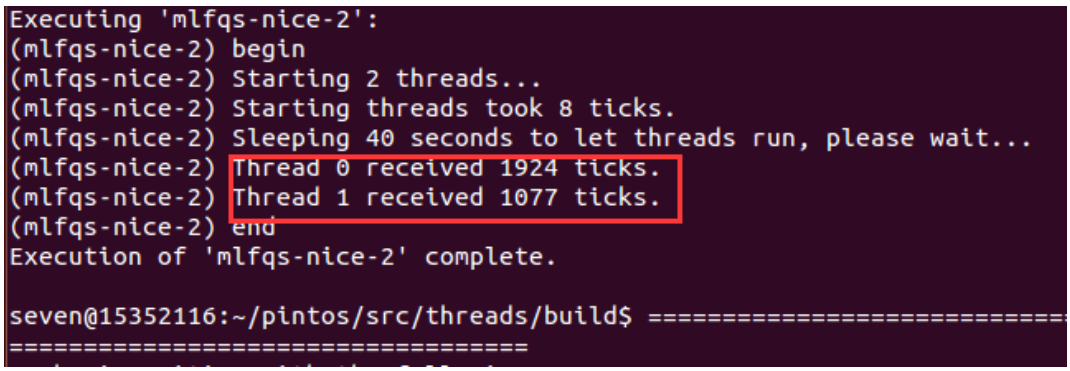
✧ 过程分析

由于传入的 nice_step 是 5，所以创建了两个不同 nice 值的子线程，

第一个 nice 值为 0，第二个为 5。所以当两个线程同步唤醒之后，优先级不再相等，nice 值越高的优先级会越低，所以两个子线程获得 tick 的概率也不相等，nice 值较低的线程优先级较高，获得优先级的概率会更大（概率之所以不是 1 是因为，当获得更多 ticks 之后相应的优先级会有所下降），不过两个子线程的 ticks 总数应该为 3000。

✧ 结果分析

由结果可知，越早创建的子线程获得 tick 概率越大，和过程分析一致，测试通过。



◆ **mlfqs-nice-10 测试样例**

✧ 测试目的

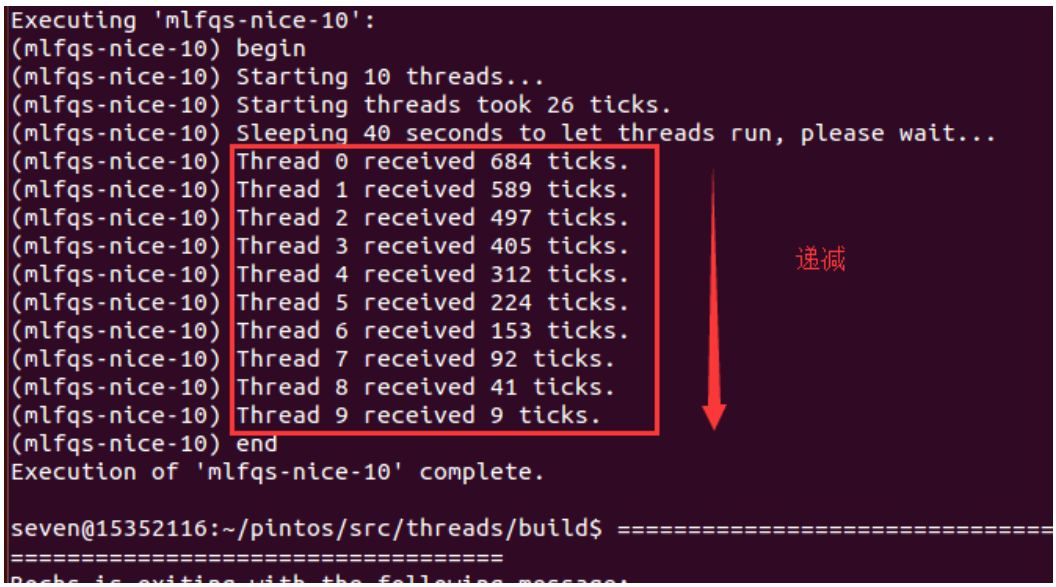
测试 10 个具有不同 nice 值的线程获得 ticks 的统计特性。

✧ 过程分析

和 mlfqs-nice-2 的分析过程也一致，由于传入的 thread_cnt 为 10，nice_step 为 1，所以创建了一组 nice 值为 0，1 ···，9 的子线程。所以在这 10 个子线程同步唤醒的时候，优先级也不一样，获得 tick 的概率也不一样，nice 值低的获得 tick 的概率更大，即越早创建的线程获得的 tick 的概率越大。

✧ 结果分析

根据结果输出可知，越早创建的线程获得 tick 数越大，即获得的概率越大，与过程分析一致，测试通过。



◆ priority-condvar 测试样例

✧ 测试目的

测试对于条件变量的操作，能否按照优先级大小的顺序唤醒被条件变量阻塞的线程。

✧ 过程分析

① 首先初始化 lock 锁和 condition 条件变量，并把 test 线程的优先级设置为 PRI_MIN。然后进入 for 循环中，创建了 10 个优先级不同的子线程。由于子线程优先级大于 test 线程，抢占 CPU 进入 priority_condvar_thread 函数。在这个函数里面首先子线程会申请锁，并且发出条件变量的等待信号。又由于在等待过程中会释放掉锁，所以每次创建新的线程的时候，每个子线程都可以成功地申请到锁，并且由于条件变量为假，有序地插入条件变量地等待队列中。

test_priority_condvar 函数：

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

thread_set_priority (PRI_MIN);
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);
  }
```

优先级大于test线程

priority_condvar_thread 函数：

```
msg ("Thread %s starting.", thread_name ());
lock_acquire (&lock);
cond_wait (&condition, &lock);
```

释放锁，进入条件变量的阻塞队列

② 所有的线程创建完之后都进入了条件变量的阻塞队列中，test 线程重新获得 CPU，进入下一个 for 循环，首先会申请锁 lock，紧接着会发出通知信号，唤醒在条件变量的阻塞队列中优先级最高的线程，抢占 test 的 CPU，又由于唤醒的时候会申请锁 lock，而此时锁 lock 被 test 线程占有，所以又进入了锁 lock 的等待队列中，同时捐赠优先级给 test 线程。

③ test 线程重新获得 CPU，然后执行释放锁的语句，test 线程优先级又变为 PRI_MIN，同时唤醒锁的等待队列的（优先级最高）线程，进入 priority_condvar_thread 函数输出 msg 语句之后，释放锁并结束生命周期。然后 test 重新获得 CPU 进入下一个循环。

test_priority_condvar 函数：

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

唤醒子线程，子线程进入锁的等待队列

释放锁，唤醒子线程，子线程抢占CPU

priority_condvar_thread 函数：

```
msg ("Thread %s woke up.", thread_name ());
lock_release (&lock);
```
输出msg，释放锁，结束线程

✧ 结果分析

由结果可知，子线程可以按照优先级的顺序依次被唤醒。测试通过。

```
(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
```

2.2 实验思路与代码分析

◆ 实验思路

✧ 实现条件变量相关操作

➢ 条件变量的含义：

条件变量是利用线程间共享的全全局变量进行同步的一种机制，

有两种基本的操作，等待&通知。当条件不满足的时候会发出等待信号，并且会阻塞当前的线程，阻塞的同时会释放用于互斥条件的锁。当条件满足的时候，由另一个线程发出通知信号给关联的条件变量，唤醒等待该条件信号的优先级最高的线程。

- 实现条件变量的 waiter 队列的优先级排序

新增的参数变量：

参数	类型	作用	位置
sema_priority	int	实现条件信号阻塞队列中的有序插入	semaphore 结构体（synch.h）

改进 con_wait 函数：

在 semaphore_elem 结构体中定义了 list_elem 和 semaphore 类型的参数，前者是作为排序的小弟用，后者存储了该条件信号的相关内容。在 semaphore 结构体加入 sema_priority 参数之后，每次等待信号，将该信号的优先级设置了线程的优先级，然后再进行有序插入条件信号的等待队列当中,伪代码如下。

```
cond_wait(cond, lock) {
…
//实现有序插入，信号量的优先级和线程本身具有的优先级一样
waiter.semaphore.sema_priority = thread_current ()->priority;
list_insert_ordered (&cond->waiters, &waiter.elem, cond_sema_cmp_priority);
}
cond_sema_cmp_priority (a, b){
cond_a = list_entry (a, struct semaphore_elem, elem);
cond_b = list_entry (b, struct semaphore_elem, elem);
return cond_a->semaphore.sema_priority>cond_ b->semaphore.sema_priority ;
}
```

✧ 实现整型模拟浮点型运算

由于 pintos 不支持浮点运算，而在多级反馈队列中需要用到浮点型运算，所以使用 32 位 int 的后 16 表示小数部分，最高位表示正负，中间 15 位表示整数部分来模拟浮点型运算。最后的浮点数实际上还是用 int 存储的，只是我们默认在中间加了一个小数点。所以在有些运算的时候我们可以用原有的 int 运算。

本实验需要实现四则运算、四舍五入。取整、进制转换等功能。伪代码如下：

#define f　1<<16	
Convert n to fixed point	n*f
Convert x to integer (rounding toward zero) ：	x / f
Convert x to integer (rounding to nearest) ：	if(x >= 0) (x + f / 2) / f if(x <= 0) (x - f / 2) / f
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

✧ 实现多级反馈调度

➢ 多级反馈调度的含义

多级反馈队列调度算法允许进程在队列之间移动。主要思想是根据不同的CPU区间的特点来区分线程,如果进程使用过多的CPU时间，那么它会被转移到更低优先级队列。这种方案将I/O约束和交互进程留在更高优先级队列。此外，在较低优先级队列中等待时间过长的进程会被转移到更高优先级队列。这种形式的老化

阻止饥饿的发生。

为了解决一个高优先级且需要运行很久的线程 A 一直占有 CPU，其他较低优先级可能运行时间较短的线程一直处于等待状态的问题，多级反馈调度考虑 CPU 的吞吐量，根据线程的属性（nice 值），线程占用 CPU 的时间（recent_cpu）等线程运转情况来定期（每 4 个 timer_tick）更新所有线程的优先级，实现动态调度。

- 实现过程

1. 新增的参数变量

参数	类型	作用	位置
nice	int	对其他线程的友好程度，nice 值越高优先级越低	thread 结构体（thread.h）
recent_cpu	int	近期占有 CPU 情况，占有 CPU 时间越长，优先级越低	thread 结构体（thread.h）
load_avg	int	CPU 的平均负载，一般其值越高，优先级越低	thread.c 中静态全局变量

2. 具体函数实现

实现 thread_set_nice，thread_get_nice，thread_get_load_avg，thread_get_recent_cpu 函数：

```
thread_set_nice (nice) {    thread_current () -> nice = nice;    }
thread_get_nice (void){    return thread_current () -> nice;    }
thread_get_load_avg (void) {    return load_avg;    }
thread_get_recent_cpu (void) {    return thread_current()->recent_cpu;  }
```

实现对 recent_cpu,load_avg 以及线程优先级的计算：

```
load_avg = (59/60)*load_avg + (1/60)*ready_threads
//ready_threads 指就绪队列和运行线程中非 idle 状态的线程数
recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice
priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)
```

修改 thread_tick 函数，定时更新多级反馈队列的相关函数；实现每 100 个 ticks 更新依次系统的 load_avg 和所有线程的 recent_cpu，每 4 个 ticks 更新一次所有线程的优先级（注意每个 tick，运行的线程的 recent_cpu 加 1）：

```
thread_tick（void）{
……..
//如果该线程不是空闲线程，cpu+1
if (t != idle_thread)
    t->recent_cpu = t->recent_cpu + 1;
//每隔 100 个 ticks 更新一次 load_avg 并且更新所有线程的 recent_cpu
if (timer_ticks () % 100 == 0){
    renew_load_avg ();
    thread_foreach (renew_recent_cpu);
}
//每 4 个 ticks 更新一次优先级,就绪队列需要重新排序
if (timer_ticks () % 4 == 0){
    thread_foreach (renew_priority);
    list_sort (&ready_list, cmp_priority);
}
}
```

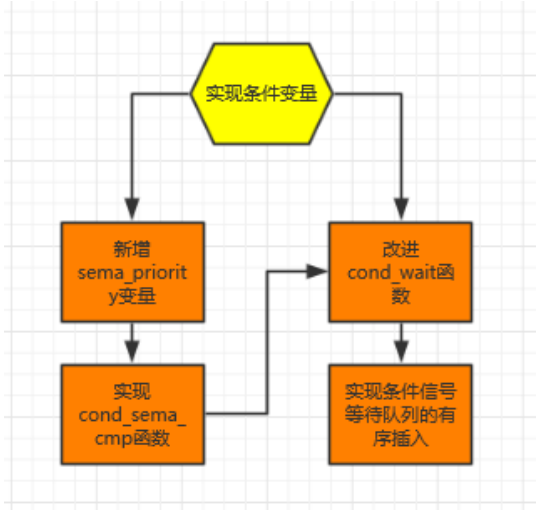
3. 修改与多级反馈调度冲突的其他函数

由于多级反馈调度过程不需要优先级捐赠，所以需要在申请锁的函数和释放锁的函数中实现优先级捐赠部分需要增加 if（！thread_mlfq）判断条件；只有多级反馈调度才需要每隔一段时间

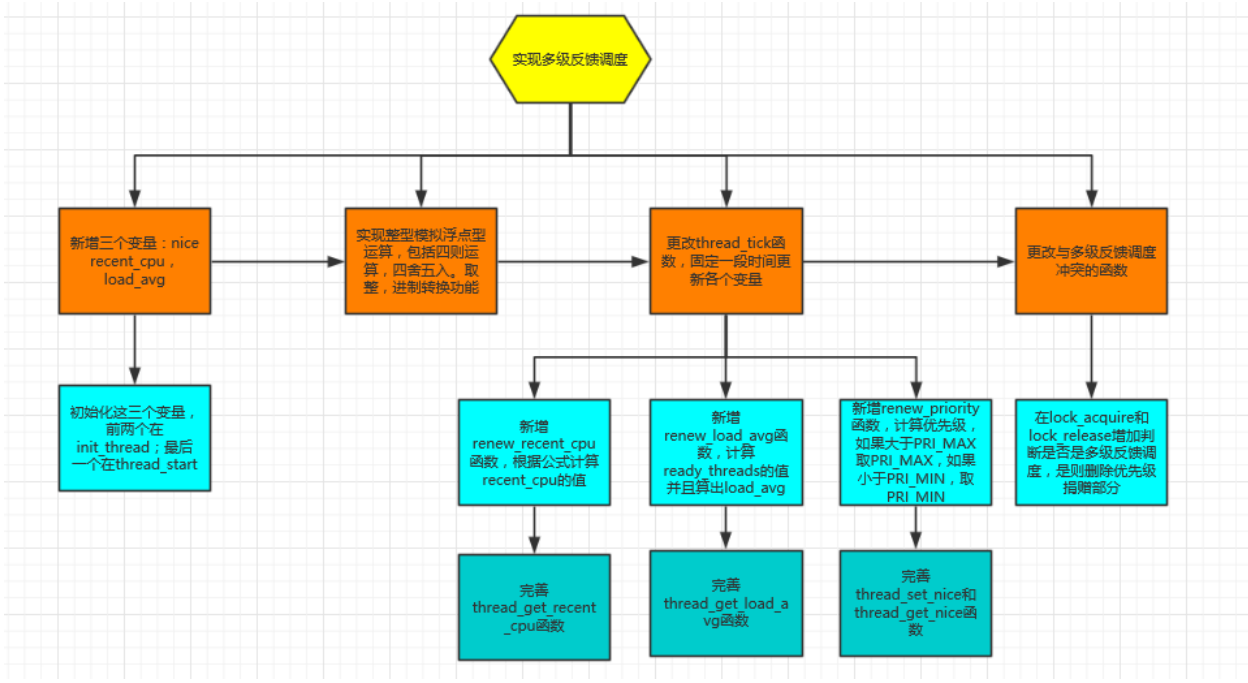
计算系统的平均负载，以及对 recent_cpu，优先级相关参数的更新，所以在这个部分需要增加 if（thread_mlfq）的判断条件。

✧ 思路流程图

➢ 条件变量的流程图：



➢ 多级反馈调度的流程图：



◆ 代码分析

根据实验分析，今次修改以及增加的函数如下：

✧ 增加 cond_sema_cmp_priority 函数

```
//////////////////////// change lab4//////////////////////////////
bool cond_sema_cmp_priority (struct list_elem* a, struct list_elem* b, void* aux)
{
  struct semaphore_elem *cond_sema_a;
  cond_sema_a = list_entry (a, struct semaphore_elem, elem);
  struct semaphore_elem *cond_sema_b;
  cond_sema_b = list_entry (b, struct semaphore_elem, elem);
  return cond_sema_a->semaphore.sema_priority > cond_sema_b->semaphore.sema_priority ;
}
//////////////////////// change lab4//////////////////////////////
```

✧ 修改 cond_wait 函数

```
//////////////////// change lab4 ////////////////////////////
//实现有序插入，信号量的优先级和线程本身具有的优先级一样
waiter.semaphore.sema_priority = thread_current ()->priority;
list_insert_ordered (&cond->waiters, &waiter.elem, cond_sema_cmp_priority, NULL);
//////////////////// change lab4 ////////////////////////////
```

✧ 对新增的参数变量进行初始化

thread_start 函数：

```
////////////// change lab4/////////
load_avg = 0;
////////////// change lab4/////////
```

init_thread 函数：

```
//////////////////////////////change lab4//////////////////////////
t->nice = 0;
t->recent_cpu = 0;          //recent_cpu是浮点型
//////////////////////////////change lab4//////////////////////////
```

✧ 完善没有实现的 thread_set_nice，thread_get_nice，thread_get_recent_cpu 以及 thread_get_load_avg 函数，在后面两个函数里面由于是模拟的浮点型，为了保留两位小数，所以要先乘上 100。

thread_set_nice 函数：

```
thread_current () -> nice = nice;
```

thread_get_nice 函数：

```
return thread_current () -> nice;
```

thread_get_recent_cpu 函数：

```
return FP_ROUND(FP_MUL_MIX(load_avg, 100));
```

thread_get_load_avg 函数：

```
return FP_ROUND(FP_MUL_MIX(thread_current()->recent_cpu, 100));
```

✧ 增加 renew_priority 函数

注意计算的结果，最大的优先级不大于 PRI_MAX；最小的优先级不小

于 PRI_MIN。

```
renew_priority(struct thread*t, void *aux)
{
    //priority = PRI_MAX - (recent_cpu/4) - (nice*2)
    t->priority = PRI_MAX - FP_ROUND(FP_DIV_MIX(t->recent_cpu, 4)) - t->nice * 2;
    t->priority = t->priority < PRI_MIN ? PRI_MIN : t->priority;
    t->priority = t->priority > PRI_MAX ? PRI_MAX : t->priority;
}
```

✧ 增加 renew_recent_cpu 函数

```
renew_recent_cpu (struct thread* t, void *aux) //和foreach的函数参数表一致
{
   //recent_cpu = (2*load_avg) / (2*load_avg + 1) *recent_cpu + nice
   //load_avg是浮点型，recent_cpu是浮点型，nice是整型
   t->recent_cpu = FP_ADD_MIX(FP_MUL(FP_DIV(FP_MUL_MIX(load_avg, 2), FP_ADD_MIX(FP_MUL_MIX(load_avg, 2), 1)), t->recent_cpu), t->nice);
}
```

✧ 增加 renew_load_avg 函数

```
renew_load_avg(void)
{
    //load_avg = (59/60) *load_avg + (1/60) *ready_threads
    //ready_threads = 等待队列数目 + 运行的线程数
    int ready_threads = list_size (&ready_list);    //等待队列数目
    //如果现在的线程不是空闲的，等待序列数目+1      //运行的线程数一般是1
    if (thread_current () != idle_thread)
    {
        ready_threads += 1;
    }
    //将ready_threads从整型转化为浮点型。
    load_avg = FP_ADD(FP_DIV_MIX(FP_MUL_MIX(load_avg, 59), 60), FP_DIV_MIX(FP_CONST(ready_threads), 60));
}
```

✧ 修改 thread_tick 函数

```
///////////////////////////// CHANGE lab4 ////////////////////////////
  enum intr_level old_level = intr_disable ();  //原子操作
  if (thread_mlfqs)
  {
    //  如果该线程不是空闲线程，cpu+1
    if (t != idle_thread)
    {
        t->recent_cpu = FP_ADD_MIX (t->recent_cpu, 1);
    }
    //每隔100个ticks更新一次load_avg
    if (timer_ticks () % 100 == 0)
    {
        renew_load_avg ();
        thread_foreach (renew_recent_cpu, NULL);
    }
    //每4个ticks更新一次优先级,就绪队列重新排序
    if (timer_ticks () % 4 == 0)
    {
        thread_foreach (renew_priority, NULL);
        list_sort (&ready_list, cmp_priority, NULL);
    }
  }
  intr_set_level(old_level);    //原子操作
///////////////////////////// CHANGE lab4 ////////////////////////////
```

✧ 修改 lock_acquire 和 lock_release 函数

lock_acquire 函数：

如果是多级反馈调度，就进入下面的语句，如果不是则进入上一次修改后的语句。

```
if (thread_mlfqs)
{
  sema_down (&lock->semaphore); //P操作
  lock->holder = thread_current();
  list_insert_ordered (&lock->holder->locks, &lock->holder_elem, lock_cmp_priority, NULL);
}
```

lock_release 函数：

如果是多级反馈调度则只运行下面这些语句，如果是其他就运行上一次修改后的语句。

```
ASSERT(current_thread->blocked == NULL);
struct thread *curr = thread_current();

lock->holder = NULL;  //释放锁
list_remove(&lock->holder_elem);  //将线程的锁的队列删除这把锁
sema_up (&lock->semaphore); //V操作
```

3. 实验结果

```
block tests/threads/mlfqs-block.result
pass tests/threads/mlfqs-block
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.
seven@15352116:~/pintos/src/threads/build$
```

4. 回答问题（15 分）

- 1.信号量，锁，条件变量的异同点?

① 信号量和锁都是通过 seme_up 和 sema_down 来实现的，只不过锁的 value 值只是 1，而信号量可以有多个实例。对于申请锁的操作，一旦这把锁被一个线程所占有，所有其他申请该锁的线程都会被阻塞。而信号量则不同，由于有多个资源，只要还剩下资源，其他的线程依旧可以成功地申请资源，直到资源全部被申请完之后，之后申请资源的线程才会进入等待资源释放的队列中。简单来说，锁可以是一种特殊的信号量。可是两者之间的意义不一样，信号量具有多个可以实现多个线程之间的通信；而锁只能被一个线程拥有，其他线程都不可以访问该锁，另一个角度也是一种被保护的资源。

② 条件变量则是在互斥锁的保护下进行的。如果目前条件为假，则需要该条件的线程发出等待信号后被阻塞，阻塞的同时会释放用于互斥条件的锁。而当条件为真的时候，由其他线程发出通知信号，并且唤醒等待该条件信号的优先级最高的线程。和锁以及信号量释放的机制不一样，它是通过其他信号发出通知信号而被唤醒的，本身拥有的那把锁在阻塞过程中会被其他线程占有，也可能不会。而锁和信号量唤醒的机制都是拥有那把锁或资源的线程主动释放锁，唤醒被锁或信号量阻塞的线程。即条件为假时，被阻塞的线程用于互斥条件的锁处于游离状态；而对于锁和信号量阻塞的线程，他们申请的锁一定处于被占有的状态。从意义上讲，条件变量是利用线程间共享的全局变量进行同步的一种机制。由于是通过通知信号，而不是通过锁的释放来唤醒被阻塞的线程，条件变量更好维护线程的

状态。

- 2. 实现多级反馈调度为什么要注释掉优先级捐赠?

  假设有一个拥有锁的较低优先级的线程一直占有 CPU，而且有高优先级的线程在申请这把锁，如果实现优先级捐赠的话，那么那个拥有锁的较低优先级的线程的优先级先会增加，然后在运行的过程中优先级降低（降低的不是自己原来的优先级），等到释放锁的时候又会恢复了自己原来的优先级。那么虽然这个线程一直在占有 CPU，可是由于优先级捐赠问题，相当于没有降低自己的优先级，也就是没有实现反馈调度算法。

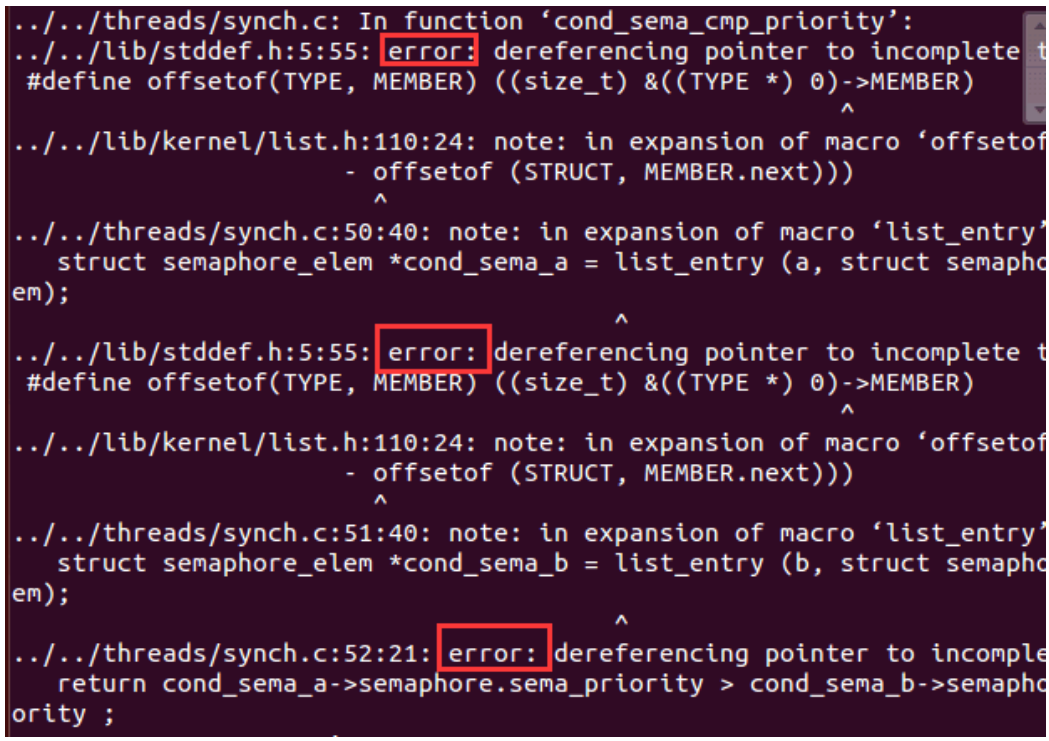
- 3. 多级反馈调度为什么能避免饥饿现象?

  由于一个线程的优先级会根据系统的平均负载以及运行的 CPU 的时间和 nice 值进行更改，即使有一个高优先级却要运行很久的线程在占有 CPU，而 有一些较低的优先级运行时间很短的线程在等待状态由于多级反馈调度，高优先级的线程运行时间越长优先级就会降得越低，而且如果当前就绪队列的线程数越多，优先级降低的速度也会增快，当优先级降到比就绪队列线程中最高的优先级还要低的时候，等到下一个轮转时刻，该线程就会被剥夺 CPU，被原本较低优先级的线程抢占 CPU，避免了饥饿现象。

5. 实验感想

在这次实验过程中，首先是对于 define 语句的使用，由于多了一个空格，程序在运行的时候失败了。又由于之前在写 cmp 函数的时候都是直接放到.c 文件所有函数的前面，没有在.h 文件中声明，这次也是保留了这个坏习惯，

导致在写 cond_sema_cmp_priority 函数的时候用了下面才声明的结构 semaphore_elem，导致报错，而且根据报错消息指向了没有修改过的地方导致了 debug 很久，所以说要改掉不太好的代码风格。



在这次实验中也懂得了很多知识，比如信号量，锁以及条件变量之间的联系和意义。还有对于多级反馈调度算法的实现以及与优先级捐赠算法之间存在的一些矛盾。