

中山大学本科生实验报告

(2018 学年春季学期)

课程名称: Operationg System

任课教师: 饶洋辉

年级+班级	2016 级	专业 (方向)	信息与计算科学
学号	16339038	姓名	舒展

1. 实验目的

- 掌握条件变量的含义、用法；明确条件变量与锁和信号量的异同点
- 使用整数模拟实现定点数及其相关运算
- 学习并实现多级反馈调度机制，通过所有的测试

2. 实验过程

(一) Test 源文件分析

• priority-condvar

测试目的：

- 乱序创建 10 个线程，让它们进入条件变量的 waiters 队列中进行排队，等待信号量通知唤醒。然后主线程进行十次循环让信号量通知线程唤醒，检查线程唤醒的顺序是否按照优先级从大到小排列，若满足要求，则测试通过。
- 本测试主要考察的是是否成功实现“cond_signal 函数是否每次唤醒的都是等待着的优先级最高的线程”

过程分析：

结构体 condition:

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

- condition 中只有一个成员变量：waiters 队列，线程等待的信号量 semaphore 在这个队列中排队
- condition 结构体的对象为全局变量，所有线程可以共享

主线程 test_priority_condvar:

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

thread_set_priority (PRI_MIN);
```

- 初始化锁和条件变量
- 将主线程自己的优先级调为最小：0

```

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);
}

```

- 10 次循环，按照优先级 23、22、21、30、29、28...24 的顺序创建 10 个子线程
- 以第一次循环为例，创建一个优先级为 23 的子线程，发生优先级抢占：

子线程开始执行 priority_condvar_thread 函数：

```

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);
}

```

- 子线程首先输出信息：“Thread 23 starting.”
- 成功请求到锁。注意 lock 为全局变量
- 调用 cond_wait 函数

进入 cond_wait 函数：

```

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);

```

- 注意首先要断言：锁是被当前线程所拥有的。否则的话临界区未被成功加锁，可能会有多个线程同时进入临界区进行读写，造成数据混乱
- 然后进入临界区，将创建的 waiter 对象插入全局变量：cond 的 waiters 队列中。
- 因为结束了对缓冲区的读写，所以可以释放锁，方便其他线程进入缓冲区；另一方面，也必须在 P 操作前释放锁，否则会造成死锁
- 然后进行 P 操作。因为 waiter 的信号量的值初始化为 0，所以当前线程被阻塞，回到主线程

回到主线程，结束第一个循环，然后接着下一个循环，依此类推...10 次循环结束后，条件变量的 waiters 队列中按照优先级 30-21 的顺序排着 10 个线程

接着主线程又是 10 个循环：

```

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

```

- 以第一次循环为例，首先申请全局变量 lock；由前面的分析可知，此时锁是不被任何线程所拥有的，所以主线程成功申请到锁
- 输出信息“Signaling”
- 调用 cond_singal 函数：

进入 cond_singal 函数：

```

ASSERT (lock_held_by_current_thread (lock));

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

```

- 同样，也要先断言当前线程是拥有锁的，因为接下来要对公共资源进行读写
- 首先需要 sort 一下条件变量的 waiters 队列，这是自己实现的部分，目的是为了使 waiters 保持优先级有序。注意这里说的优先级是 semaphore 的优先级，是本次实验添加的成员变量，它的值是阻塞的线程中最大的优先级，在之后的代码分析部分会进行说明
- 然后唤醒 waiters 队首的信号量所阻塞的线程；对第一次循环而言也就是线程 30，发生优先级抢占：

线程 30 继续执行未运行完的 cond_wait 函数：

```
lock_acquire (lock);
```

- 然后重新请求全局变量 lock；由前面分析可知：此时 lock 被主线程所拥有，所以线程 30 又被阻塞，并将自己的优先级 30 捐赠给主线程
- 注意这个 acqire 是与 priority_condvar_thread 中的最后一条语句 release 配对的

回到主线程：

```
lock_release (&lock);
```

主线程释放 lock，优先级又降为 0；子线程 30 重新被唤醒，成功申请到 lock

子线程 30 继续执行 priority_condvar_thread 函数：

```

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 “Thread 30 woke up.”
- 然后释放 lock
- 回到主线程，第一次循环完成
- 接下来就次循环同理
- 10 次循环完成后，测试完成
- 思考：对于 cond_wait 与 cond_signal 函数，为什么要将 lock_acquire 函数与 lock_release 函数放在它们的外面，而不直接写在函数的里面？

我的想法是临界区可能不只这么大，即在进行完 cond_wait/cond_signal 函数后，可能还需要接着对其它的共享资源继续进行写操作，所以这么设计使得代码更加灵活。

结果分析：

```
(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.

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

如上图所示，输出的信息与分析的一致，测试通过

• mlfqs-load-1:

测试目的：

- 证实当只有一个线程在跑的时候，load_avg 从 0 升至 0.5 以上需要 38 秒到 45 秒的时间；如果[38, 45]这个时间段内，load_avg 的值未达到 0.5、或者超过了 1、或者还没到 38 秒就已经达到了 0.5，测试均不通过
- 然后验证单个线程休眠 10 秒之后，load_avg 的值又会跌倒 0.5 之下；若不满足要求，则测试失败
- 要注意在该测试中取的浮点数 load_avg 是四舍五入保留小数点后两位的

过程分析：

test_mlfqs_load_1:

```
int64_t start_time;
int elapsed;
int load_avg;
```

测试的开始先定义了三个变量，它们的含义如下：

- start_time：用于记录下面的 for 循环的开始时间
- elapsed：用于记录从 start_time，到当前时刻所经历的秒数
- load_avg：即表示平均负载，要与全局变量的 load_avg 区别开

```
ASSERT (thread_mlfqs);

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

start_time = timer_ticks ();
```

- 要断言 thread_mlfqs 的值为 true，否则在优先级+轮转发的调度机制下，下面的测试是没有意义的！
- 输出一条语句，提示接下来要 spinning 至多 45 秒（如果运转正常的话）
- 然后将当前的 ticks 数赋值给 start_time

```
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");
}
```

开始 for 循环，至多 45 秒

- load_avg 得到当前的平均负载。注意它的逻辑意义实际上是全局变量 load_avg 乘以 100 然后四舍五入取整数部分
- 算出从 start_time 到当前的秒数，赋值给 elapsed

- 做判断，如果 load_avg 的值大于 100，也就是逻辑上此时系统的平均负载的值大于 1，则不满足要求，测试失败。（这显然是违背常理的，因为当前只有一个线程在跑，且不超过 45 秒，平均负载的值只能在 0, 1 之间。）
- 如果 load_avg > 50，则满足测试要求，可以退出循环
- 否则若时间以经过去 45 秒（但平均负载还没有到到 0.5），则测试失败

```
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);
```

- 退出循环后，还要检查 load_avg 达到 0.5 的时间下限，如果小于 38 秒，也是不满足要求的，测试会失败。
- 结束打印出 elapsed，即达到 0.5 的实际用时。系统的预期为 42 秒，我的实际为 41 秒（采用 15.16fp），结果会在下面展示

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

- 线程主动休眠 10 秒，让 load_avg 下降

```
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 %.2f",
     | | | load_avg / 100, load_avg % 100);
```

- 十秒过后，按照公式推算，平均负载理论上应该低于 5.0（要大于 0），测试通过；若出现其它情况，则测试失败。（10s 后我的实际值降至 0.43）

结果分析：

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

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

如图，按照 15.16fixed-point 的标准，41 秒后平均负载的值升至 0.5 以上；然后线程休眠 10 秒，平均负载的值降低至 0.43

• mlfqs-load-60:

测试目的：

- 主线程创建 60 个线程，然后让它们睡眠一定时间（在距 start_time 10 秒的时候同时被唤醒）；然后让这 60 个线程轮流自旋 60 秒；60 秒后又让这 60 个线程睡眠一定时间（在距 start_time +170 秒时同时被唤醒）。在这个过程

中，让主线程每个 2 秒打印一次 load_avg 的值（其余时间在睡觉），观察 load_avg 的变化。

- 如果 load_avg 的值是先增加再下降，并且是在规定的误差范围内的，则测试通过，否则测试失败。

过程分析：

```
static int64_t start_time;
```

定义全局变量 start_time，作为后面循环开始的一个时间点。

主线程 test_mlfqs_load_60:

```
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);
```

- start_time 记录当前时间，为接下来 for 循环开始的时间点
- 在每一次 for 循环中，创建一个名为 load i 的子线程。
- 注意在多级反馈调度队列中，按照规则刚创建出来的线程直接放入最高优先级队列中，即线程的初始化优先级为 63，所以有可能会抢占主线程的 cpu 资源；但是不影响接下来的测试，因为这些子线程拿到 cpu 后会马上 sleep，在 start_time + 10s 时才会同时被唤醒
- 创建完线程后输出 for 循环耗时，在我的实际输出中为 3 秒，即为 start_time + 3s 这个时间点

```
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);
}
```

- 90 次 for 循环。每次循环休眠 2 秒，然后被唤醒，输出当前 load_avg 的值
- 需要注意的是，第一次的 for 循环比较特殊（红框部分），它是休眠到 start_time + 10s 这个时间点被唤醒，然后之后都是间隔 2 秒；这么做的原因会在结果分析中说明。

子线程 load_thread:

在主线程进行循环：休眠-唤醒-休眠-唤醒...的过程中，子线程会轮流执行：

```
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;
```

- 首先定义了三个时间段：
- sleep_time：表示子线程一开始的休眠时间，10秒
- spin_time：表示子线程不断轮流自旋的时间，60秒
- exit_time：表示子线程接受自旋后再次睡眠的时间，60秒

```
thread_set_nice (20);
```

- 接着提高子线程的 nice 值；这么做的原因是防止主线程被唤醒后，无法抢占子线程的 cpu 资源，这样就无法输出 load_avg 的值了

```
timer_sleep (sleep_time - timer_elapsed (start_time));
```

- 然后每个子线程都睡眠一定时间
- 需要注意的是，它们的休眠时间都是不同的；但是再次被唤醒的时间点相同，为 start_time + 10s 的时候

```
while (timer_elapsed (start_time) < spin_time)
    continue;
```

- 子线程被同时唤醒后，这 60 个子线程轮流自旋；60 秒后所有子线程结束自旋

```
timer_sleep (exit_time - timer_elapsed (start_time));
```

- 所以子线程休眠一定时间
- 在 start_time + 130s 时，所有子线程被同时唤醒，然后结束运行
- 此时主线程的 for 循环还未结束，再经过几十秒后，for 循环结束，测试结束

结果分析：

```
Executing 'mlfqsl-load-60':
(mlfqsl-load-60) begin
(mlfqsl-load-60) Starting 60 niced load threads...
(mlfqsl-load-60) Starting threads took 2 seconds.
(mlfqsl-load-60) After 0 seconds, load average=0.00.
(mlfqsl-load-60) After 2 seconds, load average=1.98.
(mlfqsl-load-60) After 4 seconds, load average=3.90.
(mlfqsl-load-60) After 6 seconds, load average=5.75.
(mlfqsl-load-60) After 8 seconds, load average=7.55.
(mlfqsl-load-60) After 10 seconds, load average=9.28.
(mlfqsl-load-60) After 12 seconds, load average=10.96.
(mlfqsl-load-60) After 14 seconds, load average=12.58.
(mlfqsl-load-60) After 16 seconds, load average=14.14.
(mlfqsl-load-60) After 18 seconds, load average=15.66.
(mlfqsl-load-60) After 20 seconds, load average=17.12.
(mlfqsl-load-60) After 22 seconds, load average=18.54.
(mlfqsl-load-60) After 24 seconds, load average=19.91.
(mlfqsl-load-60) After 26 seconds, load average=21.23.
```

- 如上图，首先注意到一个细节：既然一开始所有子线程都休眠了大约 10 秒，那么为何前 10 秒 load_avg 还在持续增长？这其实与第一次 for 循环有关：

```
int64_t sleep_until = start_time + TIMER_FREQ * (2 * i + 10);
```

+10 操作使得主线程第一次被唤醒的时间点为：start_time + 10s，即与子线程同时被唤醒，然后主线程才开始打印 load_avg 的值，所以之后 load_avg 的值自然是一直在上升的

```
(mlfq -> load -> 60) After 42 seconds, load average=30.37.
(mlfq -> load -> 60) After 44 seconds, load average=31.34.
(mlfq -> load -> 60) After 46 seconds, load average=32.29.
(mlfq -> load -> 60) After 48 seconds, load average=33.21.
(mlfq -> load -> 60) After 50 seconds, load average=34.09.
(mlfq -> load -> 60) After 52 seconds, load average=34.95.
(mlfq -> load -> 60) After 54 seconds, load average=35.77.
(mlfq -> load -> 60) After 56 seconds, load average=36.57.
(mlfq -> load -> 60) After 58 seconds, load average=37.34.
(mlfq -> load -> 60) After 60 seconds, load average=38.09.
(mlfq -> load -> 60) After 62 seconds, load average=36.83.
(mlfq -> load -> 60) After 64 seconds, load average=35.01.
```

- load_avg 一直上升，在 60 秒时开始下降，这是因为子线程的自旋结束，开始睡眠；此时自由主线程一个线程在（间歇）运行

```
(mlfq -> load -> 60) After 112 seconds, load average=15.89.
(mlfq -> load -> 60) After 114 seconds, load average=15.36.
(mlfq -> load -> 60) After 116 seconds, load average=14.85.
(mlfq -> load -> 60) After 118 seconds, load average=14.36.
(mlfq -> load -> 60) After 120 seconds, load average=13.89.
(mlfq -> load -> 60) After 122 seconds, load average=13.74.
(mlfq -> load -> 60) After 124 seconds, load average=13.28.
(mlfq -> load -> 60) After 126 seconds, load average=12.84.
(mlfq -> load -> 60) After 128 seconds, load average=12.42.
(mlfq -> load -> 60) After 130 seconds, load average=12.01.
(mlfq -> load -> 60) After 132 seconds, load average=11.61.
(mlfq -> load -> 60) After 134 seconds, load average=11.23.
(mlfq -> load -> 60) After 136 seconds, load average=10.86.
(mlfq -> load -> 60) After 138 seconds, load average=10.50.
```

- 再过 60 秒，子线程同时被唤醒，但是为何之后 load_avg 还会继续下降？这是因为子线程被唤醒后就会结束运行，所以之后还是只有主线程一个线程在（间歇）运行

```
(mlfq -> load -> 60) After 170 seconds, load average=6.13.
(mlfq -> load -> 60) After 172 seconds, load average=5.93.
(mlfq -> load -> 60) After 174 seconds, load average=5.73.
(mlfq -> load -> 60) After 176 seconds, load average=5.54.
(mlfq -> load -> 60) After 178 seconds, load average=5.36.
(mlfq -> load -> 60) end
Execution of 'mlfq -> load -> 60' complete.

shuzhan@16339038shuzhan:~/PINTOS/pintos_final/src/threads/build$
```

- load_avg 一直持续下降，直到测试结束

• mlfqs-load-avg:

测试目的：

- 主线程创建 60 个子线程，每个子线程按照 $10+i$ ($i = 0 \rightarrow 59$) 的规律休眠相应

秒数；醒来后均自旋 60 秒；然后进入休眠状态，直到 start_time + 120s 这个时间点，所有子线程被同时唤醒，结束运行。期间主线程（间歇）运行，每隔 2 秒打印出 load_avg 的值

- 分析 load_avg 值的变化规律，若满足 0->90 秒上升，之后下降的规律，且在误差范围之内，则测试通过；否则测试失败

过程分析：

```
static int64_t start_time;
```

- start_time 用于表示一个特定的时间点，从 start_time 开始，创建子线程

```
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);
```

- 接下来为 60 次 for 循环：
- 对于每一次循环，创建一个子线程 load i
- 创建 60 个子线程耗时 2 秒

```
thread_set_nice (-20);
```

- 这主线程将自己的 nice 值置为最低，为的是防止每次被唤醒时被子线程抢占 cpu 资源，导致 load_avg 无法输出

```
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);
}
```

- 90 次 for 循环：
- 对于每一次 for 循环，主线程休眠 2 秒，然后被唤醒打印 load_avg 的信息
- 与前一个 test 分析的一样，第一次 for 循环比较特殊，当主线程第一次被唤醒时，子线程 load 1 也同时被唤醒（下面会说明）；这个时间点为 start_time + 10s

子线程 load_thread:

```
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);
```

- seq_no: 表示子线程的编号
- sleep_time: 表示子线程的休眠时间
- spin_time: 表示子线程的自旋时间, 为固定的 60 秒
- exit_time: 表示子线程再次休眠的时间

```
timer_sleep (sleep_time - timer_elapsed (start_time));
```

- 子线程首先休眠一定时间; 每个子线程休眠的时间不定, 被唤醒的时间点为 start_time + (10 + i) 秒, i 为子线程的编号; 也就是说, 从 start_time + 10s 开始, 每隔 1s 就有一个子线程被唤醒

```
while (timer_elapsed (start_time) < spin_time)
| continue;
```

- 子线程被唤醒后, 便开始 (轮流) 自旋
- 每个子线程自旋的时间都是一样的: 60 秒

```
timer_sleep (exit_time - timer_elapsed (start_time));
```

- 对于每一个子线程, 60 秒自旋结束后, 便再次进入睡眠状态; 即 start_time + 70s 开始, 每隔 1 秒就有一个子线程进入休眠状态
- 在 start_time + 120s 这个时间点, 所有子线程同时被唤醒, 然后依次结束运行
- 此时主线程的 for 循环还没有结束, 还要接着 (间歇) 运行几十秒, 然后测试才结束

结果分析:

```
Executing 'mlfqsload-avg':
(mlfqsload-avg) begin
(mlfqsload-avg) Starting 60 load threads...
(mlfqsload-avg) Starting threads took 2 seconds.
(mlfqsload-avg) After 0 seconds, load average=0.00.
(mlfqsload-avg) After 2 seconds, load average=0.05.
(mlfqsload-avg) After 4 seconds, load average=0.16.
(mlfqsload-avg) After 6 seconds, load average=0.34.
(mlfqsload-avg) After 8 seconds, load average=0.58.
(mlfqsload-avg) After 10 seconds, load average=0.87.
(mlfqsload-avg) After 12 seconds, load average=1.22.
(mlfqsload-avg) After 14 seconds, load average=1.63.
(mlfqsload-avg) After 16 seconds, load average=2.09.
(mlfqsload-avg) After 18 seconds, load average=2.60.
(mlfqsload-avg) After 20 seconds, load average=3.16.
(mlfqsload-avg) After 22 seconds, load average=3.76.
(mlfqsload-avg) After 24 seconds, load average=4.41.
(mlfqsload-avg) After 26 seconds, load average=5.11.
(mlfqsload-avg) After 28 seconds, load average=5.85
```

```
(mlfq -load-avg) After 78 seconds, load average=29.91.  
(mlfq -load-avg) After 80 seconds, load average=30.28.  
(mlfq -load-avg) After 82 seconds, load average=30.58.  
(mlfq -load-avg) After 84 seconds, load average=30.79.  
(mlfq -load-avg) After 86 seconds, load average=30.95.  
(mlfq -load-avg) After 88 seconds, load average=31.00.  
(mlfq -load-avg) After 90 seconds, load average=31.01.  
(mlfq -load-avg) After 92 seconds, load average=30.94.  
(mlfq -load-avg) After 94 seconds, load average=30.81.  
(mlfq -load-avg) After 96 seconds, load average=30.62.  
(mlfq -load-avg) After 98 seconds, load average=30.38.  
(mlfq -load-avg) After 100 seconds, load average=30.07.  
(mlfq -load-avg) After 102 seconds, load average=29.69.  
(mlfq -load-avg) After 104 seconds, load average=29.25.  
(mlfq -load-avg) After 106 seconds, load average=28.76.
```

- 如上图所示，系统的 load_avg 一直时上升的，直到 90seconds 开始才下降
- 经过上面的代码分析，可以知道：在 0 seconds 这个时间点，第一个子线程被唤醒；在 59 seconds 这个时间点，最后一个子线程被唤醒，接着在 60 seconds 第一个子线程进入休眠状态，然后接下来每隔一秒休眠一个子线程。那么，为什么 60 seconds 到 90 seconds 这个时间段，就绪队列中的线程一直在减少，但是 load_avg 却一直在增加呢？原因很简单，这是因为这个时间段内，就绪队列中的子线程数 (> 30) 始终要比 load_avg 的值 (<= 30) 大，所以 load_avg 还是会一直上升的
- 而 90seconds 之后，就绪队列中的线程数小于 30 个，比 load_avg 小，所以 load_avg 开始下降

```
(mlfq -load-avg) After 168 seconds, load average=10.76.  
(mlfq -load-avg) After 170 seconds, load average=10.41.  
(mlfq -load-avg) After 172 seconds, load average=10.06.  
(mlfq -load-avg) After 174 seconds, load average=9.73.  
(mlfq -load-avg) After 176 seconds, load average=9.41.  
(mlfq -load-avg) After 178 seconds, load average=9.10.  
(mlfq -load-avg) end  
Execution of 'mlfq -load-avg' complete.  
shuzhan@16339038shuzhan:~/PINTOS/pintos_final/src/threads/build$
```

- 而在 start_time + 120s 这个时间点，虽然所有线程同时被唤醒（**其实此时还有最后几个子线程还在自旋，比较细节，但不影响**），但因为立马就会结束，所以 load_avg 并不会增加，一直下降直到测试结束

• mlfqs-recent-1:

测试目的：

- 测试只有一个线程在 cpu 中运转时，其 recent_cpu 与 load_avg 的变化规律
- 若在误差范围内，recent_cpu 与 load_avg 的值按照预期的增加，则测试通过，否则测试失败

过程分析：

```

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);

```

- 首先要不断循环睡眠，让当前线程的 recent_cpu 降至 7 以下
- Q: 为什么测试刚开始线程的 recent_cpu 就会大于 7? A: 因为测试函数其实是被 test.c 文件中的 run_test 函数（如下图）所调用的，run_test 线程已经跑了很长一段时间了，所以 recent_cpu 自然是远比 7 大的：

```

/* Runs the test named NAME. */
void
run_test (const char *name)
{
    const struct test *t;

    for (t = tests; t < tests + sizeof tests / sizeof *tests; t++)
        if (!strcmp (name, t->name))
        {
            test_name = name;
            msg ("begin");
            t->function ();
            msg ("end");
            return;
        }
    PANIC ("no test named \"%s\"", name);
}

```

```

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 %.2d, load_avg is %.2d.",
             elapsed,
             recent_cpu / 100, recent_cpu % 100,
             load_avg / 100, load_avg % 100);
        if (elapsed_seconds >= 180)
            break;
    }
    last_elapsed = elapsed;
}

```

- start_time 记录 for 循环开始的时间点
- elapsed 为 200 的整数（即秒数为 2 的倍数）时，就打印出 recent_cpu 和 load_avg 的 值，精度均为四舍五入取小数点后两位
- 180 秒后退出循环，测试结束

结果分析：

```
Executing 'mlfqs-recent-1':
(mlfqs-recent-1) begin
(mlfqs-recent-1) Sleeping 10 seconds to allow recent_cpu to decay, please wait...
.
(mlfqs-recent-1) After 2 seconds, recent_cpu is 7.39, load_avg is 0.03.
(mlfqs-recent-1) After 4 seconds, recent_cpu is 13.59, load_avg is 0.06.
(mlfqs-recent-1) After 6 seconds, recent_cpu is 19.60, load_avg is 0.10.
(mlfqs-recent-1) After 8 seconds, recent_cpu is 25.41, load_avg is 0.13.
(mlfqs-recent-1) After 10 seconds, recent_cpu is 31.05, load_avg is 0.15.
(mlfqs-recent-1) After 12 seconds, recent_cpu is 36.51, 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 46.92, load_avg is 0.24.
(mlfqs-recent-1) After 18 seconds, recent_cpu is 51.88, load_avg is 0.26.
(mlfqs-recent-1) After 20 seconds, recent_cpu is 56.68, load_avg is 0.29.
(mlfqs-recent-1) After 22 seconds, recent_cpu is 61.34, load_avg is 0.31.
(mlfqs-recent-1) After 24 seconds, recent_cpu is 65.84, load_avg is 0.33.
(mlfqs-recent-1) After 26 seconds, recent_cpu is 70.21, load_avg is 0.35.
(mlfqs-recent-1) After 28 seconds, recent_cpu is 74.43, load_avg is 0.38.

• • • • •

(mlfqs-recent-1) After 90 seconds, recent_cpu is 155.58, load_avg is 0.78.
(mlfqs-recent-1) After 92 seconds, recent_cpu is 157.06, load_avg is 0.79.
(mlfqs-recent-1) After 94 seconds, recent_cpu is 158.49, load_avg is 0.79.
(mlfqs-recent-1) After 96 seconds, recent_cpu is 159.88, load_avg is 0.80.
(mlfqs-recent-1) After 98 seconds, recent_cpu is 161.22, load_avg is 0.81.
(mlfqs-recent-1) After 100 seconds, recent_cpu is 162.52, load_avg is 0.81.
(mlfqs-recent-1) After 102 seconds, recent_cpu is 163.77, load_avg is 0.82.
(mlfqs-recent-1) After 104 seconds, recent_cpu is 164.99, load_avg is 0.82.
(mlfqs-recent-1) After 106 seconds, recent_cpu is 166.17, load_avg is 0.83.
(mlfqs-recent-1) After 108 seconds, recent_cpu is 167.30, load_avg is 0.84.
(mlfqs-recent-1) After 110 seconds, recent_cpu is 168.40, load_avg is 0.84.
(mlfqs-recent-1) After 112 seconds, recent_cpu is 169.47, load_avg is 0.85.
(mlfqs-recent-1) After 114 seconds, recent_cpu is 170.49, load_avg is 0.85.

• • • • •

(mlfqs-recent-1) After 160 seconds, recent_cpu is 186.74, load_avg is 0.93.
(mlfqs-recent-1) After 162 seconds, recent_cpu is 187.21, load_avg is 0.93.
(mlfqs-recent-1) After 164 seconds, recent_cpu is 187.65, load_avg is 0.94.
(mlfqs-recent-1) After 166 seconds, recent_cpu is 188.08, load_avg is 0.94.
(mlfqs-recent-1) After 168 seconds, recent_cpu is 188.50, load_avg is 0.94.
(mlfqs-recent-1) After 170 seconds, recent_cpu is 188.91, load_avg is 0.94.
(mlfqs-recent-1) After 172 seconds, recent_cpu is 189.29, load_avg is 0.94.
(mlfqs-recent-1) After 174 seconds, recent_cpu is 189.67, load_avg is 0.95.
(mlfqs-recent-1) After 176 seconds, recent_cpu is 190.03, load_avg is 0.95.
(mlfqs-recent-1) After 178 seconds, recent_cpu is 190.38, load_avg is 0.95.
(mlfqs-recent-1) After 180 seconds, recent_cpu is 190.73, load_avg is 0.95.
(mlfqs-recent-1) end
Execution of 'mlfqs-recent-1' complete.

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

- 如图：recent_cpu 与 load_avg 持续上升，但增长速度都是越来越慢；这是因为此时只有一个线程在运行，根据公式 $load_avg$ 越接近真实值 (1)，其增长速率就越慢；recent_cpu 的增长速率与 $load_avg$ 的增长速率呈正相关，所以也会越来越慢

• mlfqs-fair-2:

测试目的：

- 创建两个子线程，nice 值都设置为 0，然后让它们同时自旋 30 秒，测试这两个子线程占用 cpu 的时间，看看是否与模拟结果相同。（模拟结果由 mlfqs.pm 文件计算出）

- 因为两个子线程的 nice 值相同，且初始优先相同，所以按照公式，两个线程的优先级基本上都会保持相等，最终的两个线程占用 cpu 时间应该大致相等

过程分析：

test_mlfqs_fair_2 函数：

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

- 使用两个初始优先级均为 63、nice 值均为 0 的子线程进行测试

test_mlfqs_fair 函数：

```
ASSERT (nice_min + nice_step * (thread_cnt - 1) <= 20);
```

- 如上断言保证了创建的子线程的 nice 值不会超过规定的最大 nice 值

```
thread_set_nice (-20);

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

- 主线程首先将自己的 nice 值调至最低，为的是保证创建子线程时，子线程不会抢占主线程的 cpu 资源
- start_time 记录 for 循环开始的时间点
- nice 值初始为设置的最小值

```
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;

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

    nice += nice_step;
}
```

- 创建 thread_cnt 个线程，每个子线程的优先级均为 31
- nice 值按照 nice_min + i * nice_step 的规律来设置；在本测试中，两个子线程的 nice 值均为 0

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

- 主线程休眠 40 秒，让两个子线程有足够的时间执行完

子线程 load_thread:

```
thread_set_nice (ti->nice);
```

- 首先按照之前的规律设置子线程的 nice 值（均为 0）

```
timer_sleep (sleep_time - timer_elapsed (ti->start_time));
```

- 休眠一段时间，两个子线程均在 start_time + 5s 这个时间点被唤醒

```
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;
}
```

- 然后在接下来的 30 秒内，两个线程轮流执行；
- 每当一个线程进入一个新的 tick，相应的计数器就会加 1

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

- 子线程运行完后，最终回到主线程，输出记录的 tick_count 信息
- 如果代码正确，两个子线程的 tick_count 的值应该都大约为 1500（15 秒）

结果分析：

```
Executing 'mlfqs-fair-2':
(mlfqsfair-2) begin
(mlfqsfair-2) Starting 2 threads...
(mlfqsfair-2) Starting threads took 8 ticks.
(mlfqsfair-2) Sleeping 40 seconds to let threads run, please wait...
(mlfqsfair-2) Thread 0 received 1500 ticks.
(mlfqsfair-2) Thread 1 received 1501 ticks.
(mlfqsfair-2) end
Execution of 'mlfqs-fair-2' complete.

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

- 如图，一个线程的 tick_count 值为 1500，另一个为 1501，符合预期

未修改前结果分析：

```
Executing 'mlfqs-fair-2':
(mlfqsfair-2) begin
(mlfqsfair-2) Starting 2 threads...
(mlfqsfair-2) Starting threads took 8 ticks.
(mlfqsfair-2) Sleeping 40 seconds to let threads run, please wait...
(mlfqsfair-2) Thread 0 received 1501 ticks.
(mlfqsfair-2) Thread 1 received 1501 ticks.
(mlfqsfair-2) end
Execution of 'mlfqs-fair-2' complete.

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

- 如图，未实现多级反馈调度机制前，mlfqs-fair-2 这个测试仍然能够通过：两个线程都运行了约 1500 个 ticks。这是因为若未实现多级反馈调度，那么系统使用的

仍然是优先级+轮转发的调度机制，而这两个线程的优先级均为 31；所以它们一共轮流自旋 30 秒，每个线程占用 cpu 的时间自然也就约为 1500 个 ticks，这与多级反馈调度产生的结果是一致的。mlfq-fair-20 能够通过，也是相同原因。

- **mlfq-fair-20:**

测试目的：

- 测试目的与 fair-2 一样，创建 20 个初始优先级相同、nice 值为 0 的子线程，然后让它们轮流运行共三十秒；查看各线程 cpu 占用情况，如果符合预期的话，每个子线程占用的 cpu 时间应该都约为 1.5 秒

过程分析：

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

- 使用 20 个初始优先级均为 63、nice 值均为 0 的子线程进行测试
- 接下来的流程与 fair-2 同，不在详细分析

结果分析：

```
Executing 'mlfq-fair-20':
(mlfq-fair-20) begin
(mlfq-fair-20) Starting 20 threads...
(mlfq-fair-20) Starting threads took 50 ticks.
(mlfq-fair-20) Sleeping 40 seconds to let threads run, please
(mlfq-fair-20) Thread 0 received 148 ticks.
(mlfq-fair-20) Thread 1 received 148 ticks.
(mlfq-fair-20) Thread 2 received 148 ticks.
(mlfq-fair-20) Thread 3 received 148 ticks.
(mlfq-fair-20) Thread 4 received 148 ticks.
(mlfq-fair-20) Thread 5 received 148 ticks.
(mlfq-fair-20) Thread 6 received 148 ticks.
(mlfq-fair-20) Thread 7 received 148 ticks.
(mlfq-fair-20) Thread 8 received 149 ticks.
(mlfq-fair-20) Thread 9 received 147 ticks.
(mlfq-fair-20) Thread 10 received 152 ticks.
(mlfq-fair-20) Thread 11 received 152 ticks.
(mlfq-fair-20) Thread 12 received 152 ticks.
(mlfq-fair-20) Thread 13 received 153 ticks.
(mlfq-fair-20) Thread 14 received 152 ticks.
(mlfq-fair-20) Thread 15 received 152 ticks.
(mlfq-fair-20) Thread 16 received 152 ticks.
(mlfq-fair-20) Thread 17 received 152 ticks.
(mlfq-fair-20) Thread 18 received 153 ticks.
(mlfq-fair-20) Thread 19 received 150 ticks.
(mlfq-fair-20) end
Execution of 'mlfq-fair-20' complete.

shuzhan@16339038shuzhan:~/PINTOS/pintos_final/src/threads/b
=====
```

- 如上图，20 个线程所占用的 cpu 时间均约为 150 个 ticks
- 线程之间占用 ticks 数有略微不同的原因，可能是因为在实际运行时不同各个线程的 recent_cpu 会有略微差别；但是因为 nice 值以及初始优先级相同，所以总体上各个线程还是维持一个相同的优先级

- **mlfq-nice-2:**

测试目的:

- 创建两个子线程，初始优先级均为 31，一个线程的 nice 值为 0，一个为 5；让这两个子线程一共运行 30 秒，分别记录两个子线程占用 cpu 的时间，查看是否与 mlfqs.pm 文件模拟出的结果相同
- 若在误差范围内结果一致，则测试通过；否则测试失败

过程分析:

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

- 使用两个初始优先级均为 63，nice 值分别为 0 和 5 的子线程进行测试
- 后面的创建子线程过程与前面的 test 一致，不详细分析

```
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;
}
```

- 当这两个线程一开始进行休眠时，由于 nice 值不同，所以子线程 1 的优先级会逐渐大于子线程 2 的优先级
- 当它们在 start_time + 5s 这个时间点同时被唤醒时，子线程 1 由于优先级更高，会一直运行；子线程 2 没有运行的机会
- 但在子线程 1 运行的过程中，由于 recent_cpu 的不断增长，优先级会逐渐降低，最终小于等于子线程 2 的优先级，使得子线程 2 有机会运行
- 但由于子线程 1 的 nice 值要更低，所以一段时间后，其优先级又会高于子线程 2 的优先级，如此进行下去...
- 总的来说，由于多级反馈调度机制的存在，使得两个子线程都有轮流运行的机会，但是由于子线程 1 的 nice 值更低，所以其被分配的 cpu 资源要更多

结果分析:

```
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 1928 ticks.
(mlfqs-nice-2) Thread 1 received 1072 ticks.
(mlfqs-nice-2) end
Execution of 'mlfqs-nice-2' complete.

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

- 如上图，测试结果与模拟结果基本一致

- **mlfq-nice-10:**

测试目的:

- 测试目的与 nice-2 相同

过程分析:

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

- 使用 10 个初始优先级均为 63, nice 值为 0-9 的子线程进行测试
- 过程与 nice-10 类似, 不在再详细分析

结果分析:

```
Executing 'mlfq-nice-10':
(mlfq-nice-10) begin
(mlfq-nice-10) Starting 10 threads...
(mlfq-nice-10) Starting threads took 26 ticks.
(mlfq-nice-10) Sleeping 40 seconds to let threads run, please wait...
(mlfq-nice-10) Thread 0 received 684 ticks.
(mlfq-nice-10) Thread 1 received 597 ticks.
(mlfq-nice-10) Thread 2 received 496 ticks.
(mlfq-nice-10) Thread 3 received 404 ticks.
(mlfq-nice-10) Thread 4 received 309 ticks.
(mlfq-nice-10) Thread 5 received 224 ticks.
(mlfq-nice-10) Thread 6 received 152 ticks.
(mlfq-nice-10) Thread 7 received 89 ticks.
(mlfq-nice-10) Thread 8 received 40 ticks.
(mlfq-nice-10) Thread 9 received 9 ticks.
(mlfq-nice-10) end
Execution of 'mlfq-nice-10' complete.

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

- 实际测试结果与模拟结果一致
- 以之前分析的一样, 在初始优先级一样的情况下, nice 值越低, 拿到的 cpu 资源也就越多

- **mlfq-block:**

测试目的:

- 测试当一个线程被阻塞时, 它的 recent_cpu 和优先级是会不断更新的
- 测试开始先让主线程拥有一把锁, 然后让其休眠 25 秒, 再自旋 5 秒; 同时创建一个子线程, 让它先自旋 20 秒, 请求所, 然后被阻塞 10 秒; 在第 30 秒时, 主线程释放锁; 如果子线程在被阻塞时 recent_cpu 正确第下降, 那么按照设计, 主线程释放锁时, 子线程的优先级会比主线程优先级高, 发生优先级抢占

过程分析:

主线程 test_mlfqs_block:

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

- 主线程先成功申请到锁

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

- 然后创建一个初始优先级为 31 的子线程；由于此时主线程优先级比 31 高，所以不会发生优先级抢占
- 接着主线程休眠 25 秒

子线程 block_thread:

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

- 子线程先自旋 20 秒；在这个过程中，按照公式它的优先级是会不断下降的（通过设置 test 的输出，得知子线程的优先级在自旋前为 61，在自旋后为 38）

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

- 子线程自旋完毕后，申请锁；此时锁被主线程所拥有，于是子线程被阻塞

25 秒后，主线程被唤醒：

```
start_time = timer_ticks ();
while (timer_elapsed (start_time) < 5 * TIMER_FREQ)
| continue;
```

- 主线程开始自旋 5 秒，在这个过程中主线程的优先级应该是不断降低的
- 而此时被阻塞的子线程的优先级应该是在不断上升的

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

- 主线程自旋结束，释放锁，子线程被唤醒
- 按照预期，此时子线程的优先级应该比主线程优先级高，发生优先级抢占

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

- 子线程拿到锁，输出该信息，然后结束运行

回到主线程：

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

- 主线程最终输出该信息，然后结束整个测试

结果分析：

```

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.

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

```

如图，输出信息与分析的一致，说明线程在被阻塞时，它的 recent_cpu 和优先级的更新方式的确是正确的，测试通过。

(二) 实验思路与代码分析

条件变量部分：

(1) 结构体 semaphore:

```

struct semaphore
{
    unsigned value;           /* Current value. */
    struct list waiters;      /* List of waiting threads. */
    int lock_priority;
};

```

- 添加成员变量 lock_priority，它表示的是 semaphore 的优先级
- 在 sema_init 中初始为 0，储存的值为信号量的 waiters 队列中线程最大的优先级

(2) cond_wait 函数:

```

waiter.semaphore.lock_priority = thread_current ()->priority;
sema_down (&waiter.semaphore);

```

- 在 P 操作前添加如图语句，信号量的优先级为它阻塞的线程的优先级

(3) cond_signal 函数:

```

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

```

- 要维持条件变量的 waiters 队列的优先级，可以在 cond_wait 中按优先级插入；也可以在 cond_signal 中使用 sort 函数，我选择了后者
- 这里新定义了一个 cmp_sema 函数，用以比较信号量的优先级

(4) cmp_sema 函数:

```

bool
cmp_sema(const struct list_elem *sema1, const struct list_elem *sema2, void *aux UNUSED)
{
    if (list_entry(sema1, struct semaphore_elem, elem)->semaphore.lock_priority >
        list_entry(sema2, struct semaphore_elem, elem)->semaphore.lock_priority)
        return true;
    else
        return false;
}

```

- 需要特别注意，虽然传经来的两个指针指向的对象是 semaphore_elem 类型，但正真要比较的是它的成员变量 semaphore 中的成员变量 lock_priority 的值

多级反馈调度部分：

(1) 变量设置与初始化工作

首先需要新增一些宏定义、成员变量，并将它们初始化

下面为思路与具体的代码实现：

```

/* 15.16 fixed-point number */
#define F (1 << 16)

```

- 在 thread.c 文件中，宏定义 F；F 为等下模拟定点数运行所需要使用的操作数
- 我采用的定点数格式为 15.16 format represents

```

/* 定义nice的范围 */
#define NICE_DEFAULT 0
#define NICE_MAX 20
#define NICE_MIN -20

```

- 接下来还要宏定义 nice 的默认初始值、最大值与最小值

```

/* 定义recent_cpu的默认初始值 */
#define RECENT_CPU_DEFAULT 0

/* 定义load_avg的默认初始值 */
#define LOAD_AVG_DEFAULT 0

```

- 宏定义 recent_cpu 和 load_avg 的默认初始值

```

/* 静态全局变量，平均负载量 */
/* 定点数 */
static int load_avg;

```

- 定义静态全局变量 load_avg，它逻辑上为定点小数

在结构体 thread 中，新增成员变量：

```

int nice;
int recent_cpu;

```

- nice 逻辑上为整数； recent_cpu 逻辑上为定点小数

在 init_thread 函数中给新增的成员变量赋值：

```

if (thread_mlfqs)
    t->priority = PRI_MAX;
else
    t->priority = priority;

t->nice = NICE_DEFAULT;
t->recent_cpu = RECENT_CPU_DEFAULT;

/* 初始化全局变量load_avg */
load_avg = LOAD_AVG_DEFAULT;

```

- 要注意：若为多级反馈队列机制，线程的初始化优先级应为 63
- nice 值和 recent_cpu 值都初始化为 0
- 还要在该函数中完成全局变量 load_avg 的初始化（只初始化一次）

(2) 定点小数运算操作实现

实现定点小数，是为了满足 load_avg、recent_cpu、priority 三个变量的计算需要，下面为思路与具体代码实现：

int 转为定点数：

```

/* int转为fp */
int int_to_fp(int n)
{
    return n * F;
}

```

- 直接左移 16 位即可

定点数转为 int：

a. 直接舍去小数部分：

```

int fp_to_int_round_zero (int x)
{
    return x / F;
}

```

- 右移 16 位

b. 四舍五入：

```

int fp_to_int_round_nearest (int x)
{
    if (x >= 0)
        return (x + F / 2) / F;
    else
        return (x - F / 2) / F;
}

```

- 对于正数，在第 15 位加上 1，那么小数部分若 ≥ 0.5 ，会进一位到第 16 位；否则不会。再将结果右移 16 位即可达到四舍五入的效果
- 对于负数，第 15 位减去 1，再右移 16 位即可

两个定点数相加：

```
int fp_add (int x, int y)
{
    return x + y;
}
```

- 因为小数点是对齐的，直接相加即可

两个定点数相减：

```
int fp_sub (int x, int y)
{
    return x - y;
}
```

定点数与整数相加：

```
int fp_add_int (int x, int n)
{
    return x + n * F;
```

- 整数左移 16 位，使得小数点对齐，然后直接相加即可

定点数减整数：

```
int fp_sub_int (int x,int n)
{
    return x - n * F;
```

定点数乘整数：

```
int fp_mul_int (int x, int n)
{
    return x * n;
```

定点数除以整数：

```
int fp_div_int (int x, int n)
{
    return x / n;
```

定点小数相乘：

```
int fp_mul (int x, int y)
{
    return ((int64_t) x) * y / F;
```

- 要考虑到结果溢出问题，引用 pintos 中给出的例子进行说明：

在 17.14 格式的定点数中，逻辑上的 64 实际值为 2^{**20} ；64 的平方为 2^{**16} ，这个值在 17.14 格式的定点数中是可以被表示到的。但是实际上 $(2^{**20})^{**2}$ 值为

2^{**40} , 这是 32 位整数所不能表示的, 这就造成了溢出问题, 使得模拟的定点数结果不正确。

- 发生溢出问题的本质很简单, 其实是两个定点小数相乘, 运算结果的小数的在逻辑上左移了 16 位, 使得可能溢出。解决方法: 先将相乘的结果扩展到 64 位, 这样就可以防止溢出, 然后再将运算结果右移 16 位对齐小数位即可

定点数相除:

```
int fp_div (int x ,int y)
{
    return ((int64_t) x) * F / y;
```

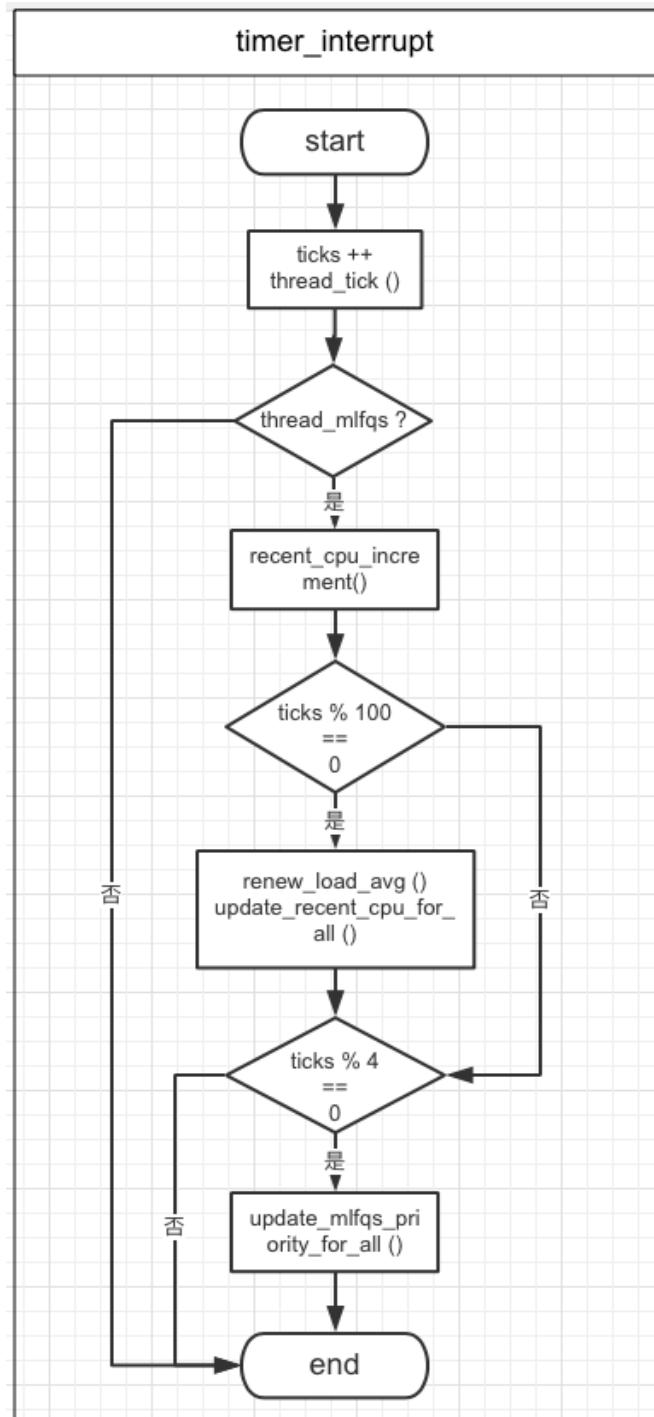
- 不用考虑溢出问题, 但是两个定点数相除后, **逻辑上小数点会向右移动了 16 位**, 所以需要将结果再左移 16 位
- 以上便实现了 pintos 中所需要的全部定点数相关的运算操作, 另外还需要在 thread.h 文件中声明这些函数

(3) 多级反馈队列的实现

- 前面进行了这么多准备工作, 都是为接下来实现多级反馈队列而服务的;
- 按照规则, 每一个 tick 正在 running 的线程的 recent_cpu+1; 每 4 个 ticks 更新一次所以线程的优先级; 每 100 个 ticks 更新一些系统 load_avg 和所有线程的 recent_cpu
- timer_interrupt () 函数为核心部分, 每次中断, 由它来调度其它函数进行参数的更新:

```
if (thread_mlfqs)
{
    recent_cpu_increment ();
    if (ticks % TIMER_FREQ == 0)
    {
        renew_load_avg ();
        update_recent_cpu_for_all ();
    }
    if (ticks % UPDATE_FREQ == 0)
    {
        update_mlfqs_priority_for_all ();
    }
}
```

- 首先要明确, 以上工作只有在 thread_mlfqs 为真的前提下才能进行
- 每一个 tick, 调用一次自定义的 recent_cpu_increment () 函数, 使当前正在运行的线程的 recent_cpu 增 1
- 若过了 100 个 ticks, 则调用 renew_load_avg () 更新系统的平均负载; 调用 update_recent_cpu_for_all () 更新所有线程的 recent_cpu
- 若过了 4 个 ticks, 则调用 update_mlfqs_priority_for_all () 更新所有线程的优先级
- timer_interrupt 的流程图如下:



- 下面为这些自定义函数的实现:

`recent_cpu_increment()`:

```

void recent_cpu_increment (void)
{
    if (thread_current () == idle_thread)
        return;

    thread_current ()->recent_cpu = fp_add (
        thread_current ()->recent_cpu, int_to_fp (1));
}

```

- 函数作用：每个 tick running 线程 recent_cpu 增 1
- 若为 idle_thread，则不做处理
- 否则当前线程的 recent_cpu 增 1

update_recent_cpu_for_all ():

```

void renew_load_avg (void)
{
    int len = list_size (&ready_list);
    if (thread_current () != idle_thread)
        len++;

    int temp1 = fp_div (int_to_fp (59), int_to_fp(60));
    int temp2 = fp_div (int_to_fp (1), int_to_fp (60));
    load_avg = fp_add (fp_mul (temp1, load_avg), fp_mul_int (temp2, len));
}

```

- 函数功能：每 100 个 tick 更新系统 load_avg
- len 为就绪队列和运行线程中非 idle 状态的线程总数
- 然后按照更新公式计算即可；注意 59/60、1/60 都要先转化为定点数再进行除法运算

update_recent_cpu_for_all ():

```

void update_recent_cpu_for_all (void)
{
    thread_foreach (renew_recent_cpu, NULL);
}

```

- 函数功能：每 100 个 ticks 更新所有线程的 recent_cpu
- 调用 thread_foreach，遍历所有的线程，使用自定义的 renew_recent_cpu 更新单个线程的 recent_cpu

renew_recent_cpu ():

```

void renew_recent_cpu (struct thread *t)
{
    if (t == idle_thread)
        return;

    int temp1 = fp_mul_int (load_avg, 2);
    int temp2 = fp_add_int (temp1, 1);
    int temp3 = fp_div (temp1, temp2);
    int temp4 = fp_mul (temp3, t->recent_cpu);
    t->recent_cpu = fp_add_int (temp4, t->nice);
}

```

- 函数功能：按照更新公式更新单个线程的 recent_cpu
- 按照公式进行计算即可

update_mlfqs_priority_for_all ():

```
void update_mlfqs_priority_for_all (void)
{
    thread_FOREACH (renew_priority, NULL);
    list_sort (&ready_list, cmp_thread, NULL);
}
```

- 函数功能：每 4 个 ticks 更新所以线程的 priority
- 调用 thread_FOREACH，遍历所有的线程，使用自定的 renew_priority 更新单个线程的 priority
- 更新完毕后，就绪队列的顺序会被打乱，所以需要 sort 一次

renew_priority ():

```
void renew_priority (struct thread *t)
{
    /* idle_thread 不作处理 */
    if (t == idle_thread)
        return;

    int temp1 = fp_div (t->recent_cpu, int_to_fp (4));
    int temp2 = fp_mul (int_to_fp (t->nice), int_to_fp (2));
    int temp3 = fp_sub (int_to_fp (PRI_MAX), temp1);
    temp3 = fp_to_int_round_zero (fp_sub (temp3, temp2));

    if (temp3 < PRI_MIN)
        t->priority = PRI_MIN;
    else
    {
        if (temp3 > PRI_MAX)
            t->priority = PRI_MAX;
        else
            t->priority = temp3;
    }
}
```

- 函数作用：按照更新公式更新单个线程的优先级
- 如果当前运行的是 idle_thread, 不需要更新
- 否则按照更新公式进行计算即可
- 另外还需要考虑计算结果越界的问题，如果越界，则直接取边界值即可

(4) Pintos 为完成函数实现：

另外 pintos 中还有几个已经定义好的函数，需要我们来实现，这几个函数在 mlfqs 的几个 test 中会用到

thread_set_nice ():

```
void
thread_set_nice (int nice UNUSED)
{
| thread_current ()->nice = nice;
}
```

- 设置完线程的 nice 值后，有两种选择：一是立马更新线程的优先级；二是对优先级不做处理。我都实现了一下，两种方式都可以通过全部测试

thread_get_nice ():

```
int
thread_get_nice (void)
{
| return thread_current ()->nice;
}
```

thread_get_load_avg ():

```
int
thread_get_load_avg (void)
{
| return fp_to_int_round_nearest (fp_mul_int (load_avg, 100));
}
```

- 按照要求，返回 load_avg*100 四舍五入取整数

thread_get_recent_cpu ():

```
int
thread_get_recent_cpu (void)
{
| return fp_to_int_round_nearest (fp_mul_int (thread_current ()->recent_cpu, 100));
}
```

- 按照要求，返回 recent_cpu*100 四舍五入取整数

(5) thread_mlfqs 判断

多级反馈队列调度和优先级+轮转法是两种冲突的调度机制，所以在使用多级反馈时，我们不需要优先级捐赠：

lock_acquire ():

```
if (!thread_mlfqs)
{
    thread_current ()->blocked = lock;
    while (iter_lock != NULL && iter_lock->priority < curr_thread->priority)
    {
        iter_lock->priority = curr_thread->priority;
    }
}
```

- 在进行递归嵌套捐赠优先级前，需要先判断是否为多级反馈调度机制

```

    sema_down (&lock->semaphore);

    old_level = intr_disable ();

    lock->holder = thread_current ();
    if (!thread_mlfqs)
    {
        if (!list_empty (&(lock->semaphore.waiters)))

```

- 同理，在P操作完成后，要将申请到的锁插入锁队列前，先判断thread_mlfqs释放为假；因为锁队列是为优先级捐赠而设计的

`lock_release ():`

```

if (!thread_mlfqs)
{
    list_remove (&lock->elem);
    update_thread_priority (thread_current ());

```

- 在释放锁时，在将锁从锁队列中移除之前，也需要判断thread_mlfqs是否为假

以上便是本次实验所需要修改的全部地方

3. 实验结果



```

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.
shuzhan@16339038shuzhan:~/PINTOS/pintos_final/src/threads/build$ 

```

如上图，所以测试均通过

4. 回答问题

问题 1:信号量、锁、条件变量的异同点？

答：在之前的实验中已经分析过，信号量可以理解为多值锁，这里不再详细分析，下面分析一下条件变量与前两者的异同。条件变量与信号量一样，可以阻塞一个或者多个线程；被阻塞的线程都需要满足某种条件后，被其它线程唤醒。但区别在于，等待信号量的线程，被唤醒的条件只有一个：锁被释放，临界区可以进入了；而等待条件变量的线程，被唤醒的条件可以是多种的，如临界区可以进入、某一个线程先执行完毕等等……也就是说，条件变量中，线程等待的“条件”，是比信号量中的“信

号”更宽泛的一个概念，这便是条件变量与前两者最大的不同。另外还需要明确的一点是，条件变量总是和互斥锁搭配使用的。

问题 2:模拟浮点数中如果小数部分位数为 15 或者 14 会有什么区别？

答：改为 15，可以表示的范围为 $-(2^{16}-1) \sim 2^{16}-1$ ；改为 14，可以表示的范围为 $-(2^{17}-1) \sim 2^{17}-1$ ；从 16 改为 15 或者改为 14，都可以通过测试。原因有两点：

(1) 改为 15 或者 14 后整数部分位数增加，表示范围比 16 时的大，所以不用担心溢出问题
(2) 小数部分减少，能够表示的定点数的精度减少，但是由 test 可以知道，所需要的定点数精度仅为小数点后两位，所以也不用担心精度问题。所以从 16 改为 15 或者 14 后，测试仍能通过。

问题 3:多级反馈调度为什么能避免饥饿现象？

答：因为根据多级反馈调度这个机制可以知道，优先级高的线程若一直抢占优先级低的线程的 cpu 资源，它的优先级会逐渐降低，使得它最终被其它线程抢占；而优先级低的线程，若一直没有机会拿到 cpu 资源，那么它的优先级也会逐渐升高，使得它最终有机会拿到 cpu 资源。所以正是这种根据线程的状态来调整其优先级的反馈机制，使得每个线程都在一定时间内有机会拿到 cpu，从而避免了饥饿现象。

5. 实验感想

27 个 test 终于全部都过了，还是挺开心的。整个 pintos 打下来说实话还是比较轻松的，几乎没有遇到什么大问题，要比计组实验的单、多周期设计容易多了~不过也要感谢 TA 上课讲得好哇，思路清晰，把要点几乎都告诉了我们，让我们少走了很多弯路；如果没有任何提示，让我们完全自学完成 pintos，估计还是比较吃力的。

感想也就不多说了，前面 test 分析时一不小心废话写得太多了，所以我尽快结束这篇报告了~说实话写报告还是没有打代码有意思.....