# **一、实验目的**：

在实验3的基础上，为Pintos建立优先级捐赠机制，熟悉优先级捐赠算法。

# 二、实验原理：

**1、优先级捐赠**

所谓的优先级反转，出现在高低优先级线程对锁 （Lock）的竞争之中。为了避免高优先级任务被许多低优先级任务阻塞，就需要提高占有锁的进程的优先级，将高优先级线程的优先级赋予低优先级线程，就是题目要求的优先级捐赠。

根据老师的提示，和检查 Pintos 的测试样例，可以看到优先级捐赠的典型情况有下面三种

仔细观察发生优先级捐赠的情况，可以意识到，若想完成优先级的捐赠，在两个线程之间，需要有一个用于交换优先级的媒介。自然地，锁（lock）作为两个进程之间都需要获取的目标，就可以承担起传递优先级的重任。

**2、实现**

要让锁来传递优先级，就需要对锁现有的数据结构进行改造。需要为锁引入优先级的概念。

锁引入了优先级之后，要表示线程和锁之间的关系，还需要对线程的结构进行改造。同时，由于线程在优先级捐赠过程结束之后需要恢复原始优先级，因此还需要增加一个字段，记录线程原始的优先级。



在数据结构准备完毕之后，仔细观察三种捐赠类型，寻找优先级捐赠过程中不变的特征。可以发现，不管是多么复杂的捐赠过程，其核心之处在于，在线程的并发流程中，始终保持 **线程的优先级，和其所拥有的锁的最高优先级相同。** *（锁的优先级是指，占有该锁的线程中的最高优先级）*。

意识到这点之后，我们就可以绘制出优先级捐赠的核心实现流程：



优先级捐赠的核心，就是比较线程的优先级和其占有锁的优先级。并根据比较结果，对线程自身的优先级进行实时调整。可以把这个比较过程封装成函数 thread\_check\_priority ，于是，整个优先级捐赠的关键，也是最复杂的部分，就是在于调用 thread\_check\_priority ，完成捐赠的时机。

**调用 thread\_check\_priority 的时机**

由于之前分析的，线程的优先级和其所持有锁的优先级时时相关。因此，自然地，**调用 thread\_check\_priority 的时机就在线程的优先级或其持有锁的优先级改变之时**。

什么时候锁的优先级会被改变？——当其他优先级的线程也对同一个锁进行了请求的时候。

因此，我们需要对 lock\_acquire 函数，进行改造。需要调用 thread\_check\_priority 的时机就在这里。



要注意的是，在这个函数中，**thread\_check\_priority** 函数需要调用两次：

* 在锁被获取前需要进行优先级捐赠 *提升占有这个锁的线程的优先级，使其尽快结束*
* 在锁被获取之后需要检查拿到的锁的优先级 *判断拿到的锁是否具有更高的优先级，提升自身的优先级，使自己尽快结束（递归捐赠）*

实现之前的流程之后 make check 运行测试样例发现，无法通过多重捐赠的测试，因此需要对流程进行修改，在循环中对锁进行逐层迭代遍历。修改后的 lock\_acquire 函数流程如下图所示：



如此，我们就完成了通过锁的优先级，对线程的优先级进行调整。同时，由于有了实验二线程优先级实现的保证，可以确定，只要线程的优先级发生了改变，线程的执行顺序就会按照我们的设计意图进行。至此，优先级捐赠过程就大致设计完成。

除此之外，还应该注意的是，当线程自身的优先级发生改变的时候（线程的创建、唤醒、优先级通过函数修改等）同样需要调用 thread\_check\_priority ，并维护其所持有锁的优先级，在这里就不再赘述。



**三、实验方案**

### 新增加的数据结构

* **在 struct lock 中新增**
* int priority; /\* Priority of the lock. \*/
  + priority 为锁的优先级，该优先级在锁没有被获取时候为 PRI\_MIN，被获取后等于等待获取锁的所有线程优先级的最大值。
* **在 struct thread 中新增**
* int old\_priority; /\* Old priority. \*/
* struct list locks\_holding; /\* Locks that the thread is holding. \*/
* struct lock \*lock\_waiting; /\* The lock that the thread is waiting for. \*/
  + old\_priority 记录线程自己的优先级，不受捐献的优先级影响。
  + locks\_holding 记录拥有锁的列表。
  + lock\_waiting 记录需要等待获取的锁。

### 编写 thread\_check\_priority 函数

在 thread.c 中新增，并在 thread.h 中定义

void

thread\_check\_priority (struct thread \*t)

{

int max\_priority = PRI\_MIN;

if (!list\_empty (&t->locks\_holding))

{

list\_sort (&t->locks\_holding, lock\_cmp\_by\_priority, NULL);

if (list\_entry (list\_front (&t->locks\_holding), struct lock, elem)->priority > max\_priority)

max\_priority = list\_entry (list\_front (&t->locks\_holding), struct lock, elem)->priority;

}

if (max\_priority > t->old\_priority)

t->priority = max\_priority;

else

t->priority = t->old\_priority;

list\_sort (&ready\_list, thread\_cmp\_by\_priority, NULL);

}

### lock\_acquire 修改

按照设计流程图对 lock\_acquire 进行修改，其中，迭代实现多重捐赠通过 while 循环 + 迭代器指针实现。

void

lock\_acquire (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (!lock\_held\_by\_current\_thread (lock));

if (lock->holder != NULL)

{

thread\_current ()->lock\_waiting = lock;

struct lock \*iterator\_lock = lock;

while (iterator\_lock != NULL &&

thread\_current ()->priority > iterator\_lock->priority)

{

iterator\_lock->priority = thread\_current ()->priority;

thread\_check\_priority (iterator\_lock->holder);

iterator\_lock = iterator\_lock->holder->lock\_waiting;

}

}

sema\_down (&lock->semaphore);

thread\_current ()->lock\_waiting = NULL;

list\_insert\_ordered (&thread\_current ()->locks\_holding, &lock->elem, lock\_cmp\_by\_priority, NULL);

lock->holder = thread\_current ();

thread\_check\_priority (thread\_current ());

}

### 其他

**在 lock\_release 函数新增下列语句**

list\_remove (&lock->elem);

thread\_check\_priority (thread\_current ());

lock->priority = PRI\_MIN;

lock->holder = NULL;

**修改 thread*set*priority 函数**

void

thread\_set\_priority (int new\_priority)

{

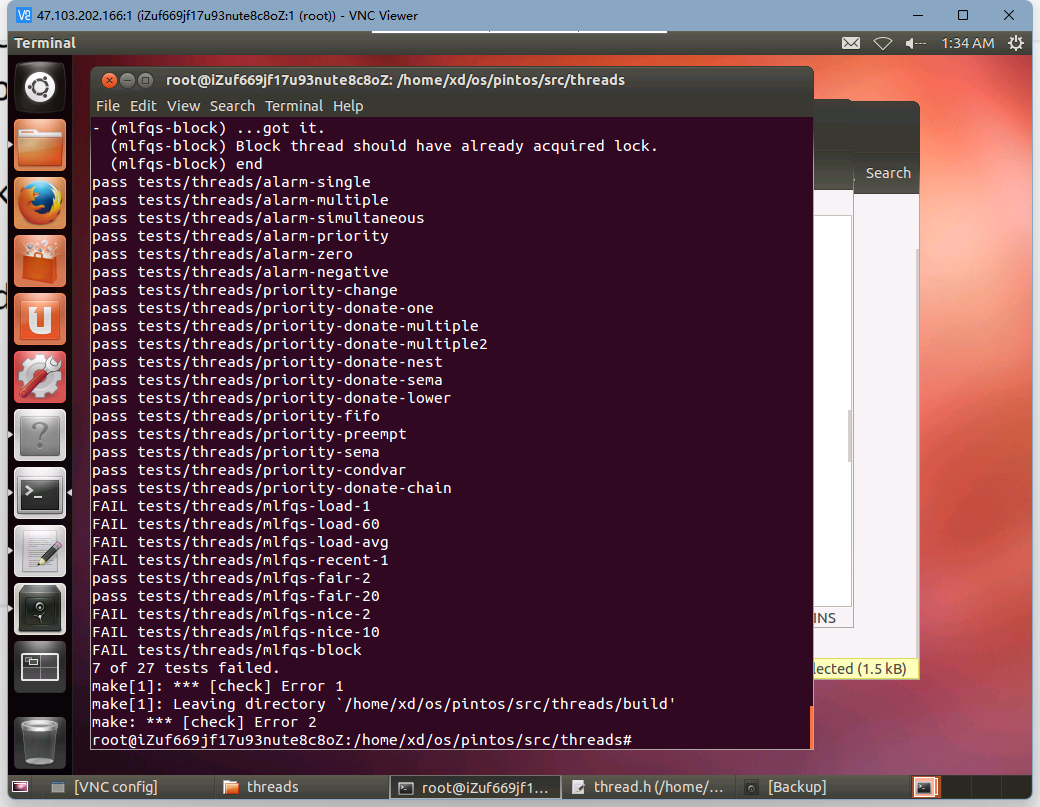
thread\_current ()->old\_priority = new\_priority;

thread\_check\_priority (thread\_current ());

thread\_yield ();

}

## 四、实验结果



cd到src/thread下执行make check 指令，编译检查后，有20个测试点通过。