**西安电子科技大学**



**操作系统课程设计**

**(2021年度)**

**实**

**验**

**报**

**告**

**实验名称：** Priority Inversion

**班 级：** 1903015

**姓 名：** 吕思勤

**学 号：** 19030130448

**一、实验内容**

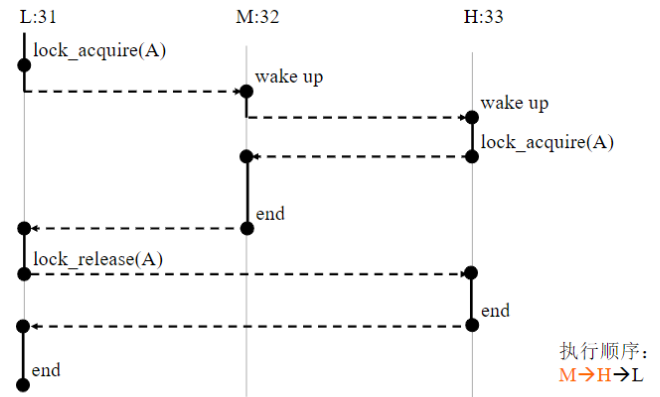
解决由lock造成的优先级反转问题。

**二、分析与设计**

优先级反转问题（Priority Inversion）：操作系统中优先级反转问题指当一个高优先级任务(线程)通过信号量机制访问共享资源时，该信号量已被一低优先级任务占有，而这个低优先级任务在访问共享资源时可能又被其它一些中等优先级任务抢先，因此造成高优先级任务被许多具有较低优先级任务阻塞，实时性难以保证。

例如：设有3个线程，其优先级由高到低为H> M> L；其中H、L中都要使用锁A访问互斥资源。

假设场景如下：H、M当前被挂起（等待相应事件的发生）；L运行，请求锁A并成功；期间，M等待的事件发生，转为就绪状态，由于M优先级高于L，因此L放弃CPU，M转为运行状态；期间，H等待的事件发生，H转为就绪状态，由于H优先级高于M，因此M放弃CPU，H转为运行状态；H也执行请求锁A的操作，但由于A已被L获取，所以H挂起，M开始运行（按优先级调度）；M运行结束后，L运行并释放锁A，系统唤醒H并运行（H优先级高于L）；H运行结束，L运行。



在这种情况下，各线程执行的顺序为M->H->L，优先级低的线程M先于优先级高的线程H运行，也即高优先级的任务被较低优先级的任务阻塞，优先级发生了反转。优先级反转会造成任务调度的不确定性，在实时系统中，还可能导致严重错误甚至系统崩溃。

从上面可以看出，优先级反转发生的基本条件为：（1）首先发生反转至少需要有三类任务，它们的优先级分别是高、中、低。

（2）低优先级和高优先级任务同时请求一个锁。

**三、详细实现**

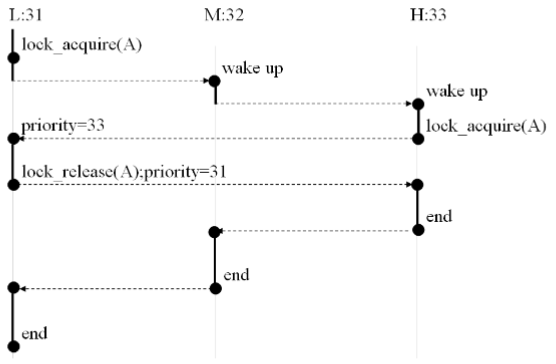
本次实验采用优先级捐租方式实现，

优先级捐赠（Priority Donation）

简单说来，优先级捐赠是指高优先级的线程将自身优先级捐给与其竞争资源的低优先级线程，以促使其尽快释放资源，并保证自身能尽快得到运行的策略。分以下几种情况：

1简单捐赠

这是最为平常和简单的一种捐赠情况，如下图：



考虑线程H、M、L，优先级H > M > L，以时间为序各线程的执行情况如下：

（1）H、M等待事件发生，处于挂起状态；L运行，持有锁A；

（2）M等待的时间发生，就绪，抢占CPU；

（3）H等待的时间发生，就绪，抢占CPU，并请求锁A；

（4）由于A已被L获得，H挂起，并将优先级捐赠给L；

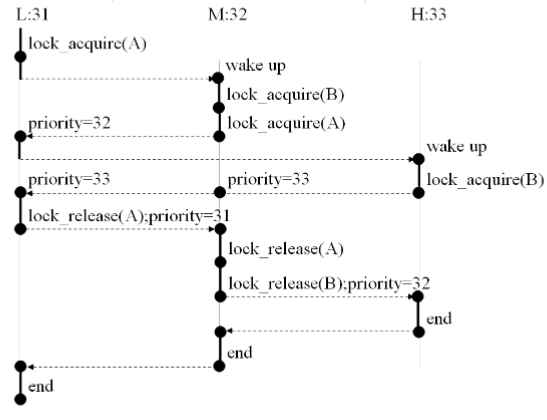
（5）L的优先级提升，开始运行，并释放锁A，同时恢复原始优先级；

（6）H被唤醒，运行；

（7）M运行；

（8）L运行。这样可以保证最终线程的执行顺序为H->M->L

2递归捐献这是一种较为复杂的情况，如下图：



设有线程H、M、L，优先级H > M > L：

（1）H、M等待事件发生，处于挂起状态；L运行，持有锁A；

（2）M等待的事件到达，就绪，抢占CPU，请求锁B成功，并继续请求锁A，由于A已被L持有，M挂起，并捐赠自己优先级32给L；

（3）L优先级获得提升，开始执行；

（4）H等待的事件到达，就绪，抢占CPU，并请求锁B，由于B被M持有，H挂起，并捐赠自己优先级33给M，由于M因L而被阻塞，故优先级33也捐赠给L；

（5）L优先级再次提升，开始运行；

（6）L释放锁A，M被唤醒，L的优先级恢复到31；

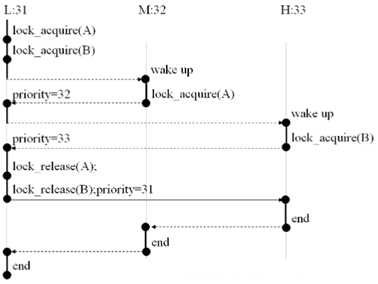
（7）M开始运行，释放锁B，H被唤醒，M的优先级恢复到32；

（8）H运行；

（9）M运行；

（10）L运行；

3多重捐献这种情况也较为复杂，如下图：



设有线程H、M、L，优先级H > M > L：

（1）H、M等待事件发生，处于挂起状态；L运行，先后持有锁A和锁B；

（2）M等待的事件发生，就绪，抢占CPU，并请求锁A；

（3）由于A被L占有，故M挂起，并将优先级32捐赠给L；

（4）L运行；

（5）H等待的事件发生，就绪，抢占CPU，并请求锁B；

（6）由于B已被L占有，H挂起，并捐献优先级33给L；

（7）L优先级提升，运行，先后释放A、B，L优先级恢复到31；

（8）H被唤醒，运行；

（9）M运行；

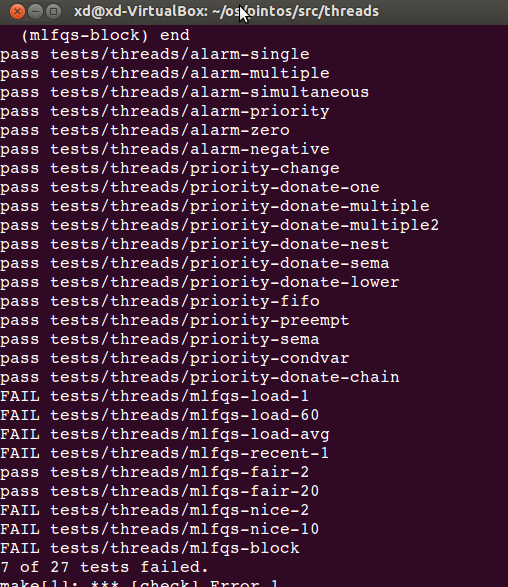
（10）L运行。

实现优先级捐赠问题时，首先需要考虑保存原优先级的问题，定义一个包含原先优先级和被捐赠时捐赠者需要什么锁两个成员的结构体 pri\_stack，并在 thread 结构体中加入类型为 struct pri\_stack 的成员，考虑到多重捐赠的情况，该成员是栈结构。之后，在 thread 结构体中加入 block\_sema、 block\_lock 两个成员，记录被哪个信号量阻塞、在等待哪个 lock。在这之后，还需修改 init\_thread()函数，使其在初始化时同时初始化这些新加入的成员。调用 thread\_set\_priority()时，为了保证线程在捐赠完毕时其优先级回到新的优先级，在该函数中加入对线程是否被捐赠过的判断。同时，还需考虑被捐赠者是否在block\_list 中，若是，且被捐赠者也被某个锁阻塞，此时调整其在信号量等待队列中的位置，并触发递归捐赠。

**四、实验结果**

成功完成了这些任务后，make check时priority-donate-one等7个检测就可以通过，如下

。



**五、心得体会**

通过这次实验，我更深入的理解了优先级反转问题，并进一步采用了优先级捐赠算法完成本次实验任务。在本次实验中我遇到的主要难题是，一开使修改代码时，我仅在thread\_set\_priority()函数的开始处加入了对是否被捐赠的判断及判断为 TRUE 之后对其栈底存储的优先级的赋值，而函数内的对当前优先级进行改变的语句均未改动，这样的做法在最终 make check 时产生了错误，具体为 priority-donate-lower 项显示 FAIL。阅读 make check 对该项的错误报告后， 针对报告的情况把 thread\_set\_priority()函数改为仅在未被捐赠过的情况下方能直接修改当前优先级、 在被捐赠过的情况下仅修改栈底，而不对当前优先级进行更改后，原本 FAIL 的项目显示 pass。这次的实验让我获益良多，日后我也会不断的增进自己，继续在操作系统领域专研、学习。