**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：** 贾舜宇

**学 号：** 19030100383

1. **实验内容**

解决由lock造成的优先级反转问题。

**实验背景：**

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。

1. **分析与设计**

**优先级捐赠：**

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。

实验说明中对优先级捐赠提供了三种思路：简单捐赠、递归捐赠以及多重捐赠。

线程H、M、L三者的优先级依次降低，线程L在执行并对互斥资源A持有线程锁，而H也需要访问这个资源，所以它只能在资源上等待不能进入就绪队列。当根据优先级选择线程调度时，只能从就绪队列中的M、L中进行选择，所以造成了最后执行顺序为M、H、L，造成优先级的反转问题。

​ 实现简单捐赠，在一个线程A需要获得lock时，如果拥有此lock的线程B的优先级比A的优先级低，就将A的优先级捐赠给B提高其优先级使其先执行，待B释放掉lock后，令其优先级还原。

​ 测试**priority-donte-multiple**需要考虑的有多个锁的情况下的逻辑情况，释放一个锁的时候，将该锁的拥有者改为该线程被捐赠的第二优先级，如果没有其余捐赠者，则回复原始优先级。这时线程结构中需要有一个新的对象来记录对此线程有捐赠行为的线程。

​ 测试**priority-donate-sema**中引入了信号量，sema\_down和sema\_up函数顾名思义就是使信号量减1（或加1）的操作，其中的信号量与lock是共同存在的，实际上是根据信号量的变化，实现锁的获取和释放。但在上次的实验中已经实现了将信号量的等待队列和condition的waiters队列改为优先级队列。

​ 而对测试**priority-donate-chain**的分析：（来自**[Pintos-斯坦福大学操作系统Project详解-Project1](https://www.cnblogs.com/laiy/p/pintos_project1_thread.html)**）

1. **void**
2. test\_priority\_donate\_chain (**void**)
3. {
4. **int** i;
5. **struct** lock locks[NESTING\_DEPTH - 1];
6. **struct** lock\_pair lock\_pairs[NESTING\_DEPTH];
8. /\* This test does not work with the MLFQS. \*/
9. ASSERT (!thread\_mlfqs);
11. thread\_set\_priority (PRI\_MIN);
13. **for** (i = 0; i < NESTING\_DEPTH - 1; i++)
14. lock\_init (&locks[i]);
16. lock\_acquire (&locks[0]);
17. msg ("%s got lock.", thread\_name ());
19. **for** (i = 1; i < NESTING\_DEPTH; i++)
20. {
21. **char** name[16];
22. **int** thread\_priority;
24. snprintf (name, **sizeof** name, "thread %d", i);
25. thread\_priority = PRI\_MIN + i \* 3;
26. lock\_pairs[i].first = i < NESTING\_DEPTH - 1 ? locks + i: NULL;
27. lock\_pairs[i].second = locks + i - 1;
29. thread\_create (name, thread\_priority, donor\_thread\_func, lock\_pairs + i);
30. msg ("%s should have priority %d.  Actual priority: %d.",
31. thread\_name (), thread\_priority, thread\_get\_priority ());
33. snprintf (name, **sizeof** name, "interloper %d", i);
34. thread\_create (name, thread\_priority - 1, interloper\_thread\_func, NULL);
35. }
37. lock\_release (&locks[0]);
38. msg ("%s finishing with priority %d.", thread\_name (),
39. thread\_get\_priority ());
40. }

​ 首先lock\_pair是包含两个lock指针的结构体， 然后将当前线程优先级设为PRI\_MIN， 然后这里有个locks数组， 容量为7, 然后lock\_pairs数组用来装lock\_pair， 容量也是7。

​ 然后当前线程获取locks[0]这个锁， 接着跳到7次循环里， 每次循环thread\_priority为PRI\_MIN+i\*3， 也就是3,6,9,12... 然后对应的lock\_pairs[i]的first记录locks[i]的指针， second记录locks[i-1]指针，

​ 然后创建线程， 优先级为thread\_priority, 执行参数传的是&lock\_pairs[i]， 注意这里由于优先级每次都底层， 所以每次循环都会抢占调用donor\_thread\_func， 然后分别获取lock\_pairs[i]里装的锁， 然后每次循环先获取first, 即locks[i], 然后获取second, 由于second是前一个， 而前一个的拥有者一定是前一次循环创建的线程， 第一次拿得的是locks[0]， 最后一次循环first为NULL， second为locks[6]， 即最后一个线程不拥有锁， 但是阻塞于前一个创建的线程， 这里还会输出信息， 即创建的线程阻塞之后会输出当前线程的优先级msg, 当然这里必然是每一次都提升了的， 所以每次都是thread\_priority。

​ 然后每次循环最后还创建了1个线程， 优先级为thread\_priority-1， 但是这里由于上一个线程创建和阻塞的过程中优先级捐献已经发生， 所以这里并不发生抢占， 只是创建出来了而已。

​ 然后original\_thread释放掉locks[0]， 释放掉这个之后thread1得到了唤醒， 输出信息， 释放掉这个锁， 然后输出当前优先级， 由于这个线程还是被后面最高优先级的线程说捐赠的， 所以每次往后优先级都是21, 然后释放掉first， 这里又触发下一个线程继续跑， 注意当后面的全部跑完的时候当前线程的优先级其实是不被捐赠的， 这里就变成了原来的优先级， 但是是所有线程都释放了之后才依次返回输出结束msg。

这个测试其实就是一个链式优先级捐赠， 本质测试的还是多层优先级捐赠逻辑的正确性。

需要注意的是一个逻辑： 释放掉一个锁之后， 如果当前线程不被捐赠即马上改为原来的优先级， 抢占式调度。

**简单总结一下实现简单优先级捐赠的逻辑：**

1. 在一个线程获取一个lock的时候， 如果拥有这个锁的线程优先级比自己低就提高它的优先级，并且如果这个锁还拥有别的lock时， 将会递归地捐赠优先级， 然后在这个线程释放掉这个锁之后恢复未捐赠逻辑下的优先级。、

2. 如果一个线程被多个线程捐赠， 维持当前优先级为捐赠优先级中的最大值（acquire和release之时）。

3. 在对一个线程进行优先级设置的时候， 如果这个线程处于被捐赠状态， 则对original\_priority进行设置， 然后如果设置的优先级大于当前优先级， 则改变当前优先级， 否则在捐赠状态取消的时候恢复original\_priority。

4. 在释放锁对一个锁优先级有改变的时候应考虑其余被捐赠优先级和当前优先级。

5. 释放锁的时候若优先级改变则可以发生抢占。

PS：

pintos中对**thread\_mlfs**的定义：

/\* If false (default), use round-robin scheduler.

If true, use multi-level feedback queue scheduler.

Controlled by kernel command-line option "-o mlfqs". \*/

bool thread\_mlfqs;

1. **详细实现**

在**synch.h**中，给**lock结构体**添加新的成员：

1. **struct** list\_elem elem;      /\* List element for priority donation. \*/
2. **int** max\_priority;          /\* Max priority among the threads acquiring the lock. \*/

​在**thread.h**中，给**thread结构体**添加新的成员：

1. **int** base\_priority;                  /\* Base priority. \*/
2. **struct** list locks;                  /\* Locks that the thread is holding. \*/
3. **struct** lock \*lock\_waiting;          /\* The lock that the thread is waiting for. \*/`

在**synch.c**中，修改lock\_acquire()函数：

1. /\* Acquires LOCK, sleeping until it becomes available if
2. necessary.  The lock must not already be held by the current
3. thread.
5. This function may sleep, so it must not be called within an
6. interrupt handler.  This function may be called with
7. interrupts disabled, but interrupts will be turned back on if
8. we need to sleep. \*/
9. **void**
10. lock\_acquire (**struct** lock \*lock)
11. {
12. **struct** **thread** \*current\_thread = thread\_current();
13. **struct** lock \*l;
14. **enum** intr\_level old\_level;
16. ASSERT (lock != NULL);
17. ASSERT (!intr\_context ());
18. ASSERT (!lock\_held\_by\_current\_thread (lock));
20. **if**(lock->holder != NULL && !thread\_mlfqs)
21. {
22. current\_thread->lock\_waiting = lock;
23. l = lock;
24. **while**(l && current\_thread->priority > l->max\_priority)
25. {
26. l->max\_priority = current\_thread->priority;
27. thread\_donate\_priority(l->holder);
28. l = l->holder->lock\_waiting;
29. }
30. }

33. sema\_down (&lock->semaphore);
35. old\_level = intr\_disable();
37. current\_thread = thread\_current();
38. **if**(!thread\_mlfqs)
39. {
40. current\_thread->lock\_waiting = NULL;
41. lock->max\_priority = current\_thread->priority;
42. thread\_hold\_the\_lock(lock);
43. }
45. lock->holder = current-**thread**;
47. intr\_set\_level(old\_level);
48. }

 在**thread.c**中添加函数thread\_hold\_the\_lock()和thread\_donate\_priority()的定义并在**thread.h**中声明

1. /\* Let thread hold a lock \*/
2. **void** thread\_hold\_the\_lock(**struct** lock\*lock)
3. {
4. **enum** intr\_level old\_level = intr\_disable();
5. list\_insert\_ordered(&thread\_current()->locks, &lock->elem, lock\_cmp\_priority, NULL);
7. **if**(lock->max\_priority > thread\_current()->priority)
8. {
9. thread\_current()->priority = lock->max\_priority;
10. thread\_yield();
11. }
13. intr\_set\_level(old\_level);
14. }
15. /\* Donate current priority to thread t. \*/
16. **void** thread\_donate\_priority(**struct** **thread** \*t)
17. {
18. **enum** intr\_level old\_level = intr\_disable();
19. thread\_update\_priority(t);
21. **if**(t->status == THREAD\_READY)
22. {
23. list\_remove(&t->elem);
24. list\_insert\_ordered(&ready\_list, &t->elem, thread\_cmp\_priority, NULL);
25. }
27. intr\_set\_level(old\_level);
28. }

实现锁队列排序函数

1. /\* lock comparation function \*/
2. ool lock\_cmp\_priority(**const** **struct** list\_elem \*a, **const** **struct** list\_elem \*b, **void** \*aux UNUSED)
3. {
4. **return** list\_entry(a, **struct** lock, elem)->max\_priority > list\_entry(b, **struct** lock, elem)->max\_priority;
5. }

修改lock\_release()函数

1. /\* Releases LOCK, which must be owned by the current thread.
3. An interrupt handler cannot acquire a lock, so it does not
4. make sense to try to release a lock within an interrupt
5. handler. \*/
6. **void**
7. lock\_release (**struct** lock \*lock)
8. {
9. **enum** intr\_level old\_level;
11. ASSERT (lock != NULL);
12. ASSERT (lock\_held\_by\_current\_thread (lock));
14. old\_level = intr\_disable ();
16. **if** (!thread\_mlfqs)
17. thread\_remove\_lock (lock);
19. lock->holder = NULL;
20. sema\_up (&lock->semaphore);
22. intr\_set\_level (old\_level);
23. }

实现thread\_remove\_lock()函数

1. /\* Remove a lock. \*/
2. **void** thread\_remove\_lock(**struct** lock \*lock)
3. {
4. **enum** intr\_level old\_level = intr\_disable();
5. list\_remove(&lock->elem);
6. thread\_update\_priority(thread\_current());
7. intr\_set\_level(old\_level);
8. }

当移除掉一个锁之后，当前线程的优先级可能改变，此时定义一个thread\_update\_priority()来处理此逻辑

1. /\* Update Priority \*/
2. **void** thread\_update\_priority(**struct** **thread** \*t)
3. {
4. **enum** intr\_level old\_level = intr\_disable();
5. **int** max\_priority = t->base\_priority;
6. **int** lock\_priority;
8. **if**(!list\_empty(&t->locks))
9. {
10. list\_sort(&t->locks,lock\_cmp\_priority,NULL);
11. lock\_priority = list\_entry (list\_front (&t->locks), **struct** lock, elem)->max\_priority;
12. **if** (lock\_priority > max\_priority)
13. max\_priority = lock\_priority;
14. }
16. t->priority = max\_priority;
17. intr\_set\_level (old\_level);
18. }

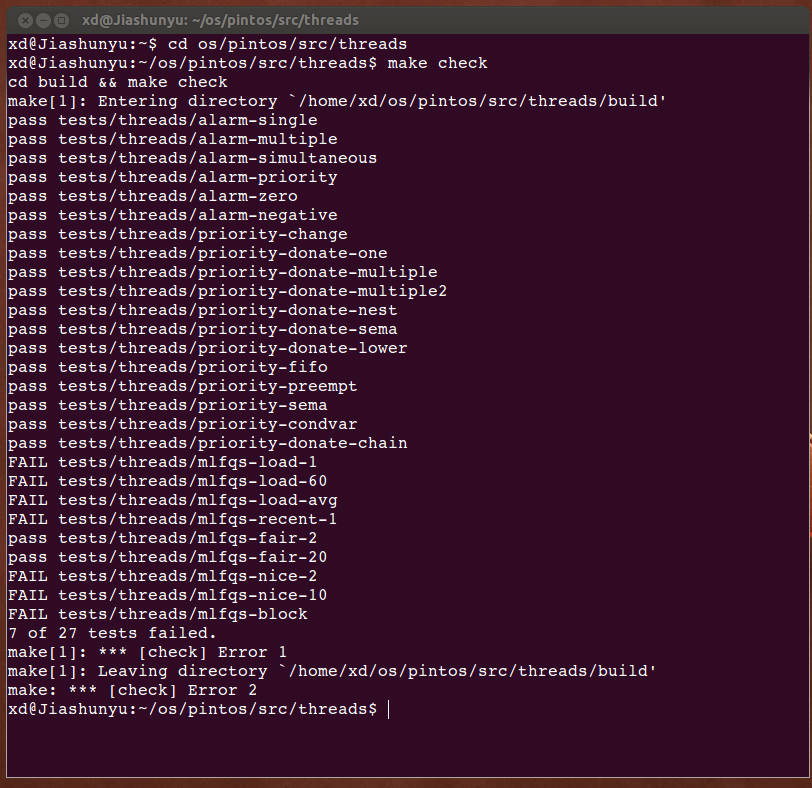
这里如果这个线程还有锁， 就先获取这个线程拥有锁的最大优先级（可能被更高级线程捐赠）， 然后如果这个优先级比base\_priority大的话更新的应该是被捐赠的优先级。

在**init\_thread**中加入初始化：

1. t->base\_priority = priority;
2. list\_init(&t->locks);
3. t->lock\_waiting = NULL;

修改**thread\_set\_priority**：

1. /\* Sets the current thread's priority to NEW\_PRIORITY. \*/
2. **void**
3. thread\_set\_priority (**int** new\_priority)
4. {
5. **if**(thread\_mlfqs)
6. **return** ;
8. **enum** intr\_level old\_level = intr\_disable ();
10. **struct** **thread** \*current\_thread = thread\_current ();
11. **int** old\_priority = current\_thread->priority;
12. current\_thread->base\_priority = new\_priority;
14. **if** (list\_empty (¤t\_thread->locks) || new\_priority > old\_priority)
15. {
16. current\_thread->priority = new\_priority;
17. thread\_yield ();
18. }
20. intr\_set\_level (old\_level);
21. }
22. **实验结果**

****

运行make check后结果如上，完成题目所要求测试，实验完成

**五、心得体会**

本次实验是操作系统课设实验的最后一次内容，难度较前两次也有不小的提升，在未看到本次关于优先级捐赠的内容之前，我也未想到基于FCFS的优先级调度问题会引出这些困难和错误，这也提醒了我在今后的类似学习或者工作中要带着怀疑和思考的眼光去看待问题，不能停留在事物的表面表现。

虽说操作系统课设的内容已经完成，但是在学习的过程中不难了解到我们的任务只是整个Pintos项目中的一小部分，但是对我来说也是受益匪浅，从源码方面可以初步了解到操作系统底层逻辑的部分情况，也能学习到操作系统中线程调度的过程，进一步加深了我对操作系统课程的学习和体会。

**参考文献**

[**Pintos-斯坦福大学操作系统Project详解-Project1**](https://www.cnblogs.com/laiy/p/pintos_project1_thread.html)