信息科学系
低年级讨论班

（6）

裘宗燕

北京大学数学学院信息科学系

2010年3月～6月

一匙污水

《代码之美》第22章

往一桶酒里加一匙污水，你得到一桶污水

——叔本华 之墒定律

引言

- 不像其他工程师的工作结果，软件的正确性是二值的，或对或错。与桥梁、机场或微处理器等不同，没有物理参数限制软件在什么范围内的正确性；软件没有额定负载、最高速度或环境工况要求。因此软件更像数学证明而不是物理机器。证明是否优雅有主观性，是否正确则不然
- 这种状况使软件具有一种纯粹性（过去只有数学独享此特征）：如果一个软件正确，其正确就是绝对而永恒的。这也带来另一情况：一点瑕疵就会使它化为恶
- 并非任何**bug**都致命，但存在这种情况，一个**bug**揭示出的重大问题足以动摇软件系统的根基，使整个系统变得毫无价值。“一颗老鼠屎坏了一锅汤”，“一匙污水臭了一桶酒”
- 作者通过**1999**年开发**Solaris**一个关键子系统时遇到的一个问题，讨论并发程序开发中的复杂情况。其中可以看到设计上的隐秘缺陷最终变成程序里的错误。以及在为了使软件能完美地解决问题的过程中，各种细节怎样变得越来越复杂和邪恶
- **Solaris** 是 **Sun** 开发的重要操作系统**(OS)**

引言

- 这一讨论涉及到 **Solaris**的一些内核细节，接触操作系统的微妙机理
- 下面讨论的核心是旋转门（**turnstile**），这是 **Solaris** 里用于阻塞或唤醒线程的机制，是互斥锁、读写锁等同步原语的基础
  - 阻塞：当一个线程需要获得某种资源，而这一需要当时不能得到满足，线程就必须等待资源的出现。这时一般把该线程放入一个与所需资源相关的队列（**queue**）。这种等待称为阻塞
  - 唤醒：当某种资源出现时，需要让等待它的线程知道这一情况，为此而做的通知操作称为唤醒
  - 互斥锁：禁止一个以上线程获得的锁（互斥）。如果有多个线程希望获取它，至多一个可以得到，其他被阻塞
  - 读写锁：一种用于保护内存存储块的锁，允许多个线程同时读信息，但如果有线程要到这里写，它必须得到这个锁的排他使用权
  - 同步原语：为协调多个线程相互和谐工作而设计的基础机制

旋转门代码

旋转门为同步原语（互斥锁、读写锁）提供阻塞和唤醒支持，支持优先级继承。下面时几个典型使用的例子：

在函数 **foo_enter()** 里为"读访问"而在锁 **'lp'** 上阻塞：

```
ts = turnstile_lookup(lp);
[ 如果锁仍被占用，设置等待者的标志位
turnstile_block(ts, TS_READER_Q, lp, &foo_sobj_ops);
```

在函数 **foo_exit()** 里唤醒为做“写访问”而正阻塞在 **'lp'** 上的线程：

```
ts = turnstile_lookup(lp);
[ 或抛弃锁 (将占有者设为 NULL)，或直接递交 (把锁的占有者直接改为想
[ 去唤醒的某个线程。如要唤醒最后进入等待者，直接清空其标志位
turnstile_wakeup(ts, TS_WRITER_Q, nwaiters, new_owner or NULL);
```

turnstile_wakeup( ) 返回时持有 **lp** 的哈希链锁

turnstile_block() and turnstile_wakeup( ) 都丢弃旋转门锁

要中止旋转门操作，客户需要调用 **turnstile_exit( )**

转门和同步原语

- 使用了旋转门代码后，各种同步原语的实现就可以专注于自己的功能逻辑，所有阻塞、唤醒细节都由旋转门处理了
- **turnstile_block( )** 就是在同步原语里实际阻塞并发线程的函数，也是下面讨论的最诡异奸诈的东西。先看作者写的代码和注释：

```
/* Follow the blocking chain to its end, willing our priority to
 * everyone who's in our way.   */
        while (t->t_sobj_ops != NULL &&
            (owner = SOBJ_OWNER(t->t_sobj_ops, t->t_wchan)) != NULL) {
                if (owner == curthread) {
                        if (SOBJ_TYPE(sobj_ops) != SOBJ_USER_PI) {
                                panic("Deadlock: cycle in blocking chain");
                        }
                        /* If the cycle we've encountered ends in mp,
                         * then we know it isn't a 'real' cycle because
                         * we're going to drop mp before we go to sleep.
                         * Moreover, since we've come full circle we know
                         * that we must have willed priority to everyone
                         * in our way.  Therefore, we can break out now.  */
                        if (t->t_wchan == (void *)mp) break;
```

这段注释和随后代码说明差之毫厘、失之千里。当时正是 Solaris 8发布前的最后时刻
后面会回来重新研究这段代码

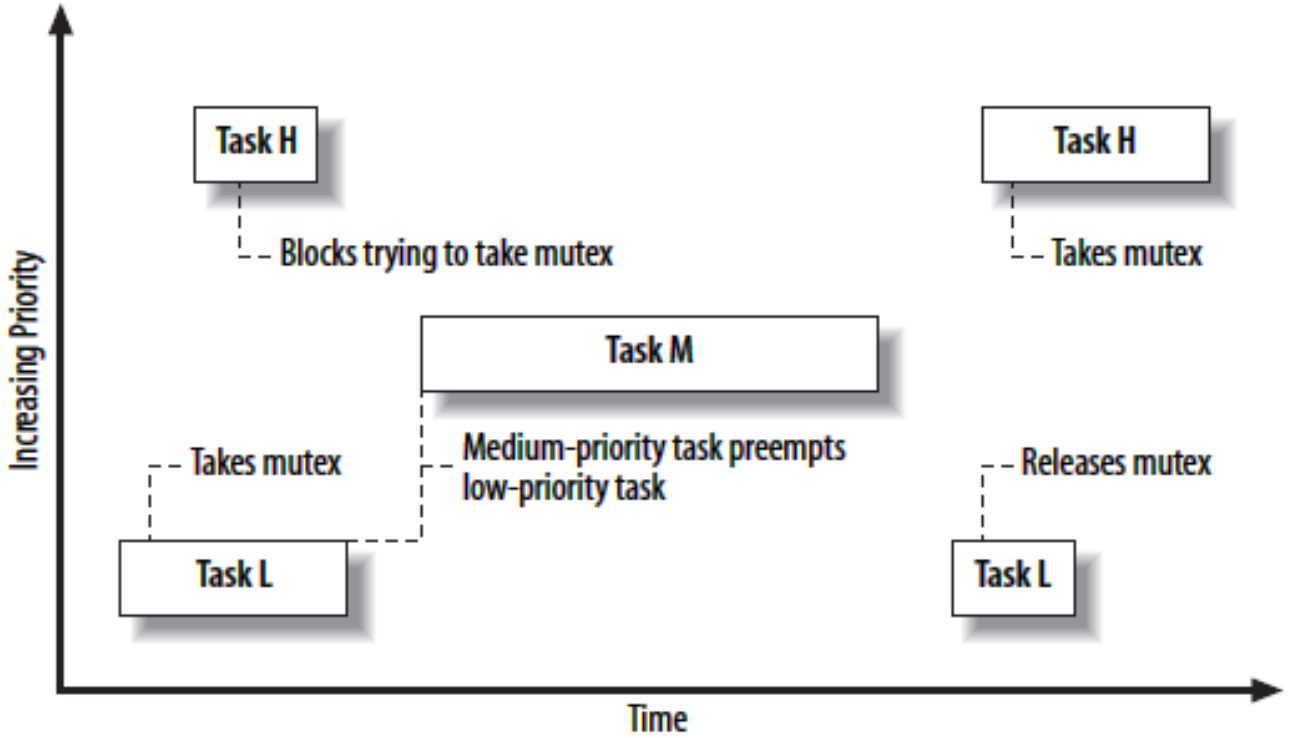
优先级倒置

- 先说明为什么需要优先级继承，以及怎样做
- 假设有三个不同优先级的线程，如果优先级最高的线程请求某个同步对象，但该对象正被优先级最低的线程持有。在简单优先级强占式的单处理器系统里，中优先级线程将先做，完成后低优先级线程做到释放该同步对象，这时高优先级线程才能开始
- 结果：

  高优先级线程不能得到优先执行

  看来不合理

  现象称为：优先级倒置



优先级继承和阻塞链

- 要避免优先级倒置，一个解决办法是采用“优先级继承”技术
  - 当高优先级线程被低优先级线程持有的资源阻塞时，高优先级线程让该低优先级线程以自己的高优先级运行完当前的临界区
  - 当低优先级线程退出这一临界区时，释放它所占资源，唤醒高优先级线程，并回到自己原来的优先级
  - 这样，在此期间，中优先级线程就不会得到运行的机会。因此没有（也不会）出现优先级倒置的现象
- **Solaris** 早在 **4.x** 版就开始在同步原语里支持优先级继承，这是旋转门子系统提供的核心服务之一。但在同步原语里正确使用它非常困难：
  - 必须知道谁拥有某锁，如该拥有者被阻塞，那么是阻塞在那个锁
  - 这样下去，可能形成一条阻塞链，其中一个线程阻塞在一个锁上，而该锁的拥有者（线程）又阻塞在下一把锁上
  - 阻塞链的细节对于正确实现优先级继承非常重要

实例

- 为看清有关情况，先看一个实际例子
- **Solaris** 有两个重要子系统，核心内存分配器和 **ZFS** 文件系统（**Zettabyte File System**），它们之间的交互就可能形成阻塞链
- 从说明问题的角度看，这两个系统的细节并不重要，不必过分关心。这里只简单介绍几句
  - 核心内存分配器是一个对象缓存式分配器，所有分配都在缓冲块里做，每个缓冲块管理一批固定大小的对象，已分配缓冲区按一个个**CPU**分别存放在称为 **magazines**（储室）的结构里
  - 当一个储室耗尽时，分配从另一称为 **depot** 的结构获得
  - 这种分层结构具有良好的 **CPU** 扩展性：大部分分配在 **CPU** 自己的储室获得（罕有冲突），分配器相对 **CPU** 的增加几乎是线性的

实例

- 这里给出一点漂亮细节，说明当 **CPU** 的储室用完后要获取 **depot** 的锁

```
/* 如不能无冲突地获得 depot 锁, 就更新冲突计数. 用 depot 冲突率
   作为依据，考虑是否为更好的扩展性增大 magazine. */
if (!mutex_tryenter(&cp->cache_depot_lock)) {
    mutex_enter(&cp->cache_depot_lock);
    cp->cache_depot_contention++;
}
```

- 上面代码试图获取锁，并记下获取失败的次数，这个计数结果可以作为全局范围中冲突情况的指示器，如果计数值太大，系统将增大 **CPU** 层缓冲块的个数，从而减小冲突率
- 这一简单结构能通过调整自身结构降低内部冲突，做的漂亮！
- 回到前面的例子

  另请记住，文件和目录在内存中对应的数据结构是**znode**

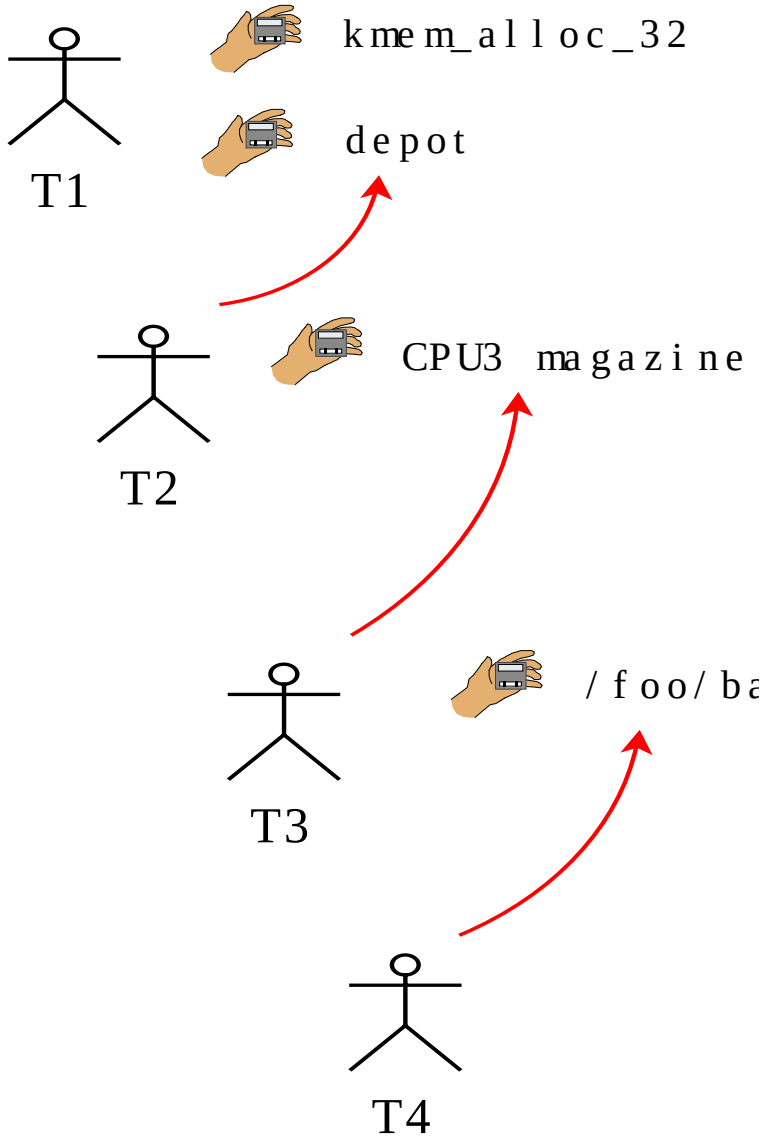
场景

考虑一个事件序列：

- 线程**T1**试图到**CPU2**的**kmem_alloc_32**缓存申请内存，因此它要先获得**kmem_alloc_32**的锁。此时发现**CPU2**的储室用尽，**T1** 为**kmem_alloc_32**获得了**depot**的锁后被抢先，此时它掌握两个锁
- 另一线程**T2**在**CPU3**运行也要从**kmem_alloc_32**做另一分配，不巧它的储室也空了，**T2**想获取**depot**锁申请**kmem_alloc_32**缓存，但看到锁被**T1**持有，因此它阻塞
- **T3**在**T2**阻塞后在**CPU3**运行企图建文件**/foo/bar/mumble**，作为操作的一部分，它必须为目录**mumble**创建一个**ZFS**目录入口锁，为此需要获取对应于**/foo/bar**的**znode**的锁，后试图分配一个**zfs_dirlock_t**。由于这个块**32**字节，要在**kmem_alloc_32**分配。**T3**想获取**CPU3**的该储室的锁，发现该锁被**T2**持有，**T3**也阻塞
- **T4**想检查目录**/foo/bar**的内容，为此需要获得对应**/foo/bar**的**znode**的锁。由于该锁被**T3**持有，因此**T4**阻塞

场景

- **T4**的阻塞是由于**T3**，**T3**阻塞于**T2**，**T2**阻塞于**T1**，这就形成了一条线程的阻塞链。由例子可以看到阻塞的混乱情况，在这种情况下弄好优先级继承不是容易的事情
  - 当一个阻塞线程把自己的优先级沿阻塞链传授时，我们必须一致地遍历整个阻塞链，即，必须看清当时在阻塞链上的所有线程的情况
  - 在这个例子里，如果**T1**释放了阻塞**T2**的锁后（这一释放传递到**T4**），我们就不想把优先级授予它（那样可能使**T1**获得不应有的优先级）



遍历阻塞链

- 怎样才能正确遍历整个阻塞链呢？
- **Solaris** 的调度状态（正在运行，在队列里等待运行，休眠）由它的线程锁保护，这是一个自旋锁。人们可能想：要正确得到阻塞链上所有线程的一致状态，只要能获得所有线程的线程锁

  由于线程锁的实现方式，此办法不行（下面是解释）
- 线程锁很特殊，它不是通常意义的自旋锁，而更像指向自旋锁的指针。它指向一个锁，该锁保护的是当时管理着这一线程的结构。当管理这一线程的结构变化时，线程锁也转去指向相应的锁。例如：
  - 当一个线程进入等待队列时，它的线程锁就指向线程的优先级对应的调度队列（**dispatch queue**，每个优先级有一个队列）
  - 线程运行时，其线程锁指向执行它的 **CPU** 的 **cpu_t** 结构里的锁
  - 线程在某个同步原语阻塞时，其线程锁又指向旋转门列表里的一个锁（事情有些纠缠，又回到了旋转门）

阻塞与唤醒

- 随着研究的深入，会看到旋转门列表结构对这个问题越来越重要
- 当务之急：无法简单地获取所有线程锁，因为可能有多个线程锁指向同一个调度锁，强行去获取就会陷入死锁

  因为想再次获得自己掌握的锁

  另外还有锁排序问题（上次讨论过）

  总之，想获取所有线程锁的想法很难实施
- 还好，不需要获取所有线程锁，就可以保证得到阻塞链的一致快照，原因是阻塞链的一个性质：它只能从非阻塞的一端开始解套

  在某同步原语上阻塞的线程，只能被拥有该同步原语的锁的线程唤醒
- 具体到前面例子：线程 **T3** 只能被 **T2** 唤醒而变成可运行的
  - 如果能保证原子地从 **T3** 到 **T2**，并原子地从 **T2** 到 **T1**，那么即使丢掉了 **T3** 的线程锁，**T3** 也不会有机会被唤醒

阻塞与唤醒

- 这意味着不必锁定整个阻塞链，而只需每个时刻锁定相继的两个线程：
  - 当 **T4** 阻塞时，可以去抓住 **T3** 的锁，而后抓住 **T2** 的锁；
  - 而后丢掉 **T3** 的锁并去抓住 **T1** 的锁；并这样做下去
- 由于只需同时抓住两个锁，它们指向同一个下层锁的情况就比较好处理了（只需检查其指向是否相同，如果相同就直接维持现状）
- 这样看问题差不多解决了，但是还有一个障碍，来自另一个设计决策
- 先解释旋转门列表，这一结构在下面讨论中很重要
  - 旋转门列表是个哈希表，其索引关键字是同步原语的虚地址，值是被同步原语阻塞的线程队列
  - 每个队列被一个旋转门锁锁定队列的头，当一个线程由于在同步原语阻塞而进入队列时，其线程锁就指向这个队列的旋转门锁
- 这里有个关键（也微妙）的设计选择：由于哈希表的性质，不同同步原语可能映射到同一表项，线程进入的队列并不唯一对应一个同步原语

旋转门列表

- 为什么采用这种设计？
  - 作为高度并行化的操作系统，**Solaris** 采用细粒度同步，其中可能存在许多（许许多多）同步原语实例，频繁操作，几乎**0**竞争
  - 因此，表示核心同步原语（**primitives-kmutex_t** 和 **krwlock_t**）的结构应尽可能小，其操作应按最常见的无竞争的情况优化
  - 把阻塞链放入同步原语结构带来的后果无法接受：或是空间（原语结构里要增加队列指针和调度锁），或是时间（由于要维护更复杂的结构，使无竞争情况的处理慢了许多）
  - 无论如何，不能把阻塞链放在同步原语里，因此需要旋转门列表
- 结论是，为提高效率需要旋转门列表，但这样：
  - 阻塞于不同同步原语的线程可能被放入同一个队列，其线程锁就会指向同一个旋转门锁
  - 由于遍历阻塞链时需要获取两个锁，会产生讨厌的锁排序问题

旋转门列表

Jeff Bonwick 的设计（写在 turnstile_interlock( ) 的注释里）：

- 实施优先级继承时，掌握着等待线程的线程锁后必须取得拥有者线程的线程锁
- 如两个线程锁都是旋转门锁，就有可能导致死锁：当拿到 **L1** 去获取 **L2** 时，某无关线程可能正在另一阻塞链上做优先级继承，它已取得 **L2** 并想得到 **L1**
- 明显解决方案是先对拥有者线程的锁做 **try_lock()**，但这不解决问题，因为它可能导致活锁（**livelock**）：两线程各拿一个锁，并同时不断试探另一个锁
- 为避免活锁，必须定义一个胜出者，即在旋转门锁上定义一个顺序。为简单，用虚拟地址定义锁顺序。这样，若 **L1 < L2**，掌握着 **L1** 要获取 **L2** 的线程自旋等待到 **L2** 可用，而掌握着 **L2** 而不能获得 **L1** 的线程应放弃 **L2** 并失败返回。为保证获胜线程有机会在失败线程之前获得 **L2**，这里让失败线程转去先设法获得 **L1**，这样它就会等待到获胜线程完成这段工作
- 还可能出现更复杂的情况：当当前线程试图获取拥有者线程的锁期间，这个锁指针可能被修改了（指向了另一个不同的锁）。如果遇到这种情况，就必须退出所有的状态，重头再来

用户层优先级继承

- 锁顺序问题是导致内核同步对象的优先级继承很难实现的重要原因，但另一方面，解决了内核优先级继承还不能完全解决优先级倒置问题
- 仅在内核同步对象上支持优先级继承显然不够，要构造多线程的实时系统，不仅需要内核层同步原语的优先级继承，也需要用户层同步原语的优先级继承。**Solaris 8** 要提供这一功能，当时指派了一个工程师去解决它（并让一些对于调度和同步有丰富经验的工程师为其提供指导）。到**1999**年**10**月这一新功能被集成到系统中
- 两个月后（**1999**年**12**月）一个同事遇到系统崩溃。作者检查了相关情况，发现明显是用户层优先级继承的实现里的问题
  - 通过进一步分析，作者认识到这不是一个表面的问题，而是一个设计缺陷。酒里出现了污水的味道
- 下面先解释一下排除 **Bug** 的方法

  软件工程师必备的一项技能就是分析一个复杂系统为什么失效，并能严格地表述所做的分析

分析系统失效

- 对足够复杂的系统，失效分析都需要取证，基于系统失效时的系统状态快照。这种快照称为**“core dump”**，是排除系统错误的基础
- 分析 **core dump** 是一种事后工作，不同于现场调试，而且只能基于系统失效时的状态剖面。这种分析很难完全，也不完备。但现场调试也有问题，有时很难重现出错的现场（尤其是并发系统，没别的办法）
- 事后分析可利用的信息较少，因此需要更缜密的思考，不断提出假设然后验证这些想法。这种锻炼也可以有效提高现场调试的能力
- 由于所用的系统状态是静态的，因此可以多个人同时分别分析，取得各自的结论。即使一个人的分析最后未被认可，相应的工作也有价值，因为它保留出一些逻辑思维的漏洞
- 总而言之：事后分析是软件工作者技术能力中很关键的一个部分，每个严肃的软件工程师都需要自我培养这种能力
- 下面看看作者的分析报告

Thread A (300039c8580)
(executing on CPU 10)

.

对锁**0xff350000**调
lwp_upimutex_lock()

.

lwp_upimutex_lock() 获得锁
upibp->upib_lock

.

lwp_upimutex_lock() 看到锁被持，调用 **turnstile_block()**

.

turnstile_block():
－获得 **A** 的线程锁
－将 **A** 的状态转为 **TS_SLEEP**
－释放 **A** 的线程锁
－释放锁 **upibp->upib_lock**
－调用 **swtch( )**

Thread B (30003c492a0)
(executing on CPU 4)

.
.

0xff350000

.
.
.
.
.
.
.
.
.
.
.
.
.
.

0xff350000的持有者释放该锁，明确将它递送给线程**A**（而且将 **300039c8580** 的拥有者设置为**upi_owner**）

从 turnstile_block()返回
.
.
.
.
.
.
.
调用 lwp_upimutex_lock() 去检查锁
的递交

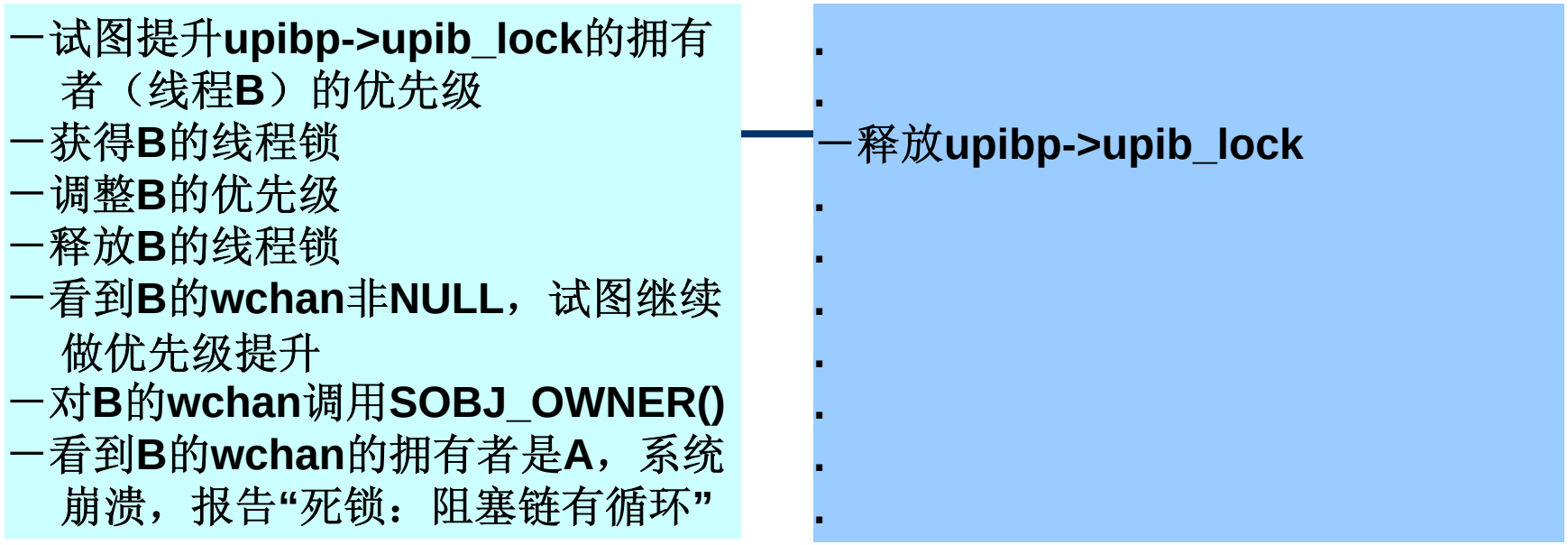
lwp_upimutex_lock() 试图获取锁
upibp->upib_lock

upibp->upib_lock被线程B持有；
通过mutex_vector_enter()调用进入
turnstile_block()：
.
turnstile_block()：
.
－获得A的线程锁

对锁0xff350000调用
lwp_upimutex_lock()
.
lwp_upimutex_lock() 获得锁
upibp->upib_lock
.
看到锁被持（被线程A所持），调用
turnstile_block()
.
turnstile_block()：
－获取B的线程锁
－将B状态改为TS_SLEEP，设B的
wchan 为对应于0xff350000的
upimutex
－试图提升0xff350000的拥有者（线程
A）的优先级
－获得A的线程锁
－调整A的优先级
－释放A的线程锁

－释放B的线程锁

0xff350000



解释：

- **upibp** 是一个指针，指向与用户层优先级继承锁关联的内核状态。**upib_lock** 是保护这个状态的锁
- **t_wchan** 是线程结构的一个成员，包含指向阻塞该线程（如果阻塞）的那个同步原语
- **SOBJ_TYPE** 是一个宏，以同步原语的操作矢量为参数，返回一个表示其类型的常量；**SOBJ_USER_PI** 是一个常量，指向一个用户层的优先级继承锁

上面情况说明问题出在 **turnstile_block( )**里：

```
        THREAD_SLEEP(t, &tc->tc_lock);
        t->t_wchan = sobj;
        t->t_sobj_ops = sobj_ops;
        ...
        /* Follow the blocking chain to its end, or until we run out of
         * inversions, willing our priority to everyone who's in our way.  */
        while (inverted && t->t_sobj_ops != NULL &&
            (owner = SOBJ_OWNER(t->t_sobj_ops, t->t_wchan)) != NULL) {
                ...
        }
(1)-> thread_unlock_nopreempt(t);
        /*
         * At this point, "t" may not be curthread. So, use "curthread", from
         * now on, instead of "t".
         */
        if (SOBJ_TYPE(sobj_ops) == SOBJ_USER_PI) {
(2)->           mutex_exit(mp);
                ...
```

在这里释放了被阻塞线程的线程锁
在**(2)**处释放**upibp->upib_lock**之前

从 **(1)** 到 **(2)** 违反了**SOBJ_USER_PI**锁的一个不变式：在**SOBJ_USER_PI**上休眠时不能持有任何内核锁。持有内核锁就可能导致死锁

问题的本质

- 对用户层锁，我们通常在用户层维持与之关联的状态的轨迹（如，是否有等待线程），这些信息完全由内核提供（但也有些情况，等待线程的标志位并不可靠，在这种情况下内核知道不能相信这个标志位）
- 要实现用户层锁的优先级继承，必须清楚掌握系统中的拥有关系信息。要像追踪内核同步原语的拥有关系一样去追踪。在**turnstile_interlock()**做复杂的线程锁操作时，仅通过在用户存储空间的操作不足以确定拥有关系
- 此情况带来一个讨厌问题：在追踪用户层锁的拥有关系时，必须用一个锁保护内核层的状态，这个（内核）锁本身也要实现优先级继承，以避免优先级倒置
- 这样就出现了一个原未预料的死锁情况：获取和释放用户层锁时必须先获取和释放一个内核锁。存在一些条件，其中线程拥有内核锁而要去获取用户层锁，也存在另一些条件，其中线程拥有内核锁而要获取用户层锁。由于这些情况，就会出现看起来像是循环的阻塞链，它们将导致内核报告致命错误

问题的本质

- 前面情况就是如此：线程**A**拥有用户层锁去请求内核锁（**upib_lock**），同时进程**B**拥有这个内核锁并希望得到用户层锁——死锁！
- 一旦理解了问题的根源，重现它就很容易了（作者觉得这时是软件工程师最得意的时候）
- 下面要考虑解决问题。由于问题极其严重，发布系统的时间很紧，作者和**Jeff**讨论，发现居然无法找到一个不带来新问题的可能解决方案，讨论越深入，发现这个问题越复杂。最后意识到是从开始就犯了错，既低估了问题，也弄错了解决它的方向
- 更糟的是，作者认识到一定有另一个潜伏问题：如果一个线程被内核锁阻塞，而且内核发现假性死锁（**false deadlock**）时会显性地崩溃；但如果线程被用户层锁阻塞并被发现假性死锁时，系统会怎么样？
- 很快就确定，在后一情况下企图获得该用户层锁，将（错误地）返回**EDEADLK**。即，内核认为这一“死锁”是用户层同步原语造成的，因此认为这是由应用程序带来的死锁，是应用程序的错误（此判断不对）

问题分析

- 在这种出错模式中，正确程序可能出现对**pthread_mutex_lock**的某次调用错误地失败的情况。这一错误模式比系统崩溃更糟，因为任何未检查其**pthread_mutex_lock**返回值（一般都不检查）的程序会假定已经拥有这个锁而实际上没拥有，做下去就可能破坏了自己的数据
- 如不了解上述情况，遇到这种情况根本就没办法。这个问题必须解决
- 但，怎么解决这个问题？作者等人认为这是个困难的问题，所以一直在设法避免内核锁。作者给出过用一个内核锁作为问题的自然约束条件的方案，但结论是这一方案无法接受。一旦有一个人提出某种不用锁的方案，另一个人立刻就发现这一方案在某些情况下不行
- 想尽了各种可能性后，作者和**Jeff**不得不做出了一个结论：一个内核锁是用户层优先级继承的一个约束。随后的工作重点从如何避免它转到如何检查它
- 有两个情况需要检查：系统崩溃和死锁

检查和处理

- 假性死锁很容易侦测和处理
  - 这时本线程总是位于阻塞链的末端，而且会发现线程自己掌握的并引起死锁的锁，就是作为参数送给**turnstile_block**的内核锁
  - 由于这时线程正在把自己的优先级授予整个阻塞链，因此可以在侦测到上述情况后跳出来。这正是前面作者写的**turnstile_block**注释所说的情况，以及最后两行代码做的事（送给**turnstile_block**的内核锁由**mp**指向）
- 系统崩溃的情况比较难处理。在这种情况下，线程拥有一个用户层同步对象，在请求内核锁时阻塞
  - 一个想法时采用类似处理方式，理由是：如果在本线程出现死锁，但阻塞链上最后的线程阻塞在用户层同步对象，就是假性死锁（这是希望推广上一条的处理方式）
  - 但这种做法是错的，因为它忽略了应用层程序的真实死锁的问题（这是应用程序中的错误），此时必须返回 **EDEADLK**

检查和处理

- 可以看到，如果阻塞链从在内核同步对象上阻塞的线程开始，直至一个在用户层同步对象上阻塞的线程，那么就一定是这种情况

  由于 **Solaris** 系统不会在持有内核锁的情况下执行用户层代码，也没有内核子系统会去请求用户层的锁。所以，一旦发现线程持有内核锁去请求用户层锁，就只有这一种情况

- 由于现在知道是被另一线程阻挡，而那个线程正执行在不可强占的代码里（那个线程正执行 **turnstile_block**，相关代码明确地禁止强占）。因此解决的方法是忙等待，直至锁的状态改变，然后再接着做令人眼花缭乱的优先级继承工作

- 这样写出的代码见下页

```
if (SOBJ_TYPE(t->t_sobj_ops) != SOBJ_USER_PI &&
    owner->t_sobj_ops != NULL &&
    SOBJ_TYPE(owner->t_sobj_ops) == SOBJ_USER_PI) {
        kmutex_t *upi_lock = (kmutex_t *) t->t_wchan;

        ASSERT(IS_UPI(upi_lock));
        ASSERT(SOBJ_TYPE(t->t_sobj_ops) == SOBJ_MUTEX);

        if (t->t_lockp != owner->t_lockp)
                thread_unlock_high(owner);
        thread_unlock_high(t);
        if (loser) lock_clear(&turnstile_loser_lock);

        while (mutex_owner(upi_lock) == owner) {
                SMT_PAUSE( );
                continue;
        }

        if (loser) lock_set(&turnstile_loser_lock);
        t = curthread;
        thread_lock_high(t);
        continue;
}
```

```
/* 我们现在掌握着拥有者的线程锁，
如果是从non-SOBJ_USER_PI
 * ops 追踪到 SOBJ_USER_PI ops,
则可知被捕捉线程在 TS_SLEEP
 * 状态且它持有mp。这种情况是过
渡状态（mp的持有者将在实际休
 *  眠在SOBJ_USER_PI sobj 之前
释放mp），所以我们自旋等待到
 *  mp被释放。而后，就像在
turnstile_interlock( ) 失败的那
 *  个情况一样，我们重新开始优先
级继承的整个过程
 */
```

问题在继续

- 接下来的压力测试暴露除进一步的问题：系统没有出现崩溃异常，但却挂住了（一般是死锁）

  分析内存转储文件，发现是一线程企图获取 **turnstile_block()** 时发生死锁，因为这个函数经过 **mutex_vector_exit( )** 递归调用自己（后一函数用于在有等待者的情况下释放互斥锁

- 前面讨论过：为获取或抛弃一个用户层的优先级继承锁，我们（无奈又必须）去获取一个内核锁

  如在用户层锁上阻塞，线程必须在完成其优先级授予后释放该内核锁，这是它调用 **swtch( )** 放弃 **CPU** 之前的最后一个动作。**p390**代码 **(2)**

  但如有其他线程在该内核锁阻塞，就需在释放内核锁时唤醒等待线程。唤醒线程需要获得与同步原语关联的旋转门表里的线程锁，而后（为撤消继承来的优先级）取得该锁前一持有者（自己）的线程锁

问题在继续

- 问题：本线程要从 **turnstile_block( )** 进入 **turnstile_pi_waive( )** 去撤消继承的优先级。此时本线程看起来已阻塞：线程锁已改，不指向当前 **CPU** 的锁，而指向与实际阻塞本线程的用户层锁对应的旋转门表项

  这样，如果有关的内核锁和用户层锁恰好被哈希到旋转门表里的同一个项（如前面第一次看到的情况），在 **turnstile_lookup( )** 和 **turnstile_pi_waive( )** 要获取的就是同一个锁

  即使这些锁没哈希到同一表项，也可能没有遵循 **turnstile_block( )** 要求的锁序列，可能出现经典的 **AB/BA** 死锁，同样不可接受

- 这个问题很难处理：

  基本问题是本线程在看起来已阻塞后还要去释放内核锁

  最希望做的是找一个办法不用内核锁，但该路已证明行不通

  现在知道需要内核锁，而且知道需要在完成整个阻塞链的优先级授予后释放它

反思，绝望和想法

- 我们不得不反思前面的基本假设：能不能颠倒 **turnstile_block()** 里的顺序，做完优先级授予再修改本线程的数据结构指明它的休眠？（不行！这样就会出现优先级倒置的窗口。）能不能引入状态标志，说明本线程正在从 **turnstile_block( )** 通过 **mutex_vector_enter( )** 去调用 **turnstile_pi_waive( )** 的时刻，因此不会死锁？（不行！这一方法没有考虑多线程死锁的场景）
- 不断提出的设想都发现有问题，我们感到绝望
  - 思考越深入，各种解决方案里被发现的问题也越多
  - 要做的就是在一个已经完美的体系中引入用户层的优先级继承
- 随着思考的深入，一些想法出现了
  - 是不是在考虑问题时想的太一般了？能不能把问题更具体化？
  - 能不能把旋转门表分块？把保护用户层优先级继承状态的内核锁放在旋转门表的一部分里，把其他锁放到另一部分？

最终方案

- 最终方案就是把旋转门表分为两块
  - 这样就能保证在 **turnstile_block( )** 里调用 **swtch( )** 之前要释放的锁与阻塞线程的锁一定在不同的旋转门表项
  - 进一步约定：所有用于保护用户层优先级继承的内核锁，在旋转门表里的虚地址都小于其他所有的锁
  - 这样，就可以保证 **turnstile_block( )** 里要求的锁顺序，单线程和多线程的情况都一并解决了
- 对解决方案的考虑：
  - 在通用旋转门表里引进了一类专用的锁（用于保护内核层和用户层的优先级继承状态），这一做法有些草率
  - 另一方面，这种做法直接而有效，是一种低风险的修改。由于当时正处于**2**年发布期限前的关键时刻，也找不到更好的办法。只能想以后有好办法再改

注释

最后结果是一段代码和一段注释

- 注释由 **Jeff** 写出

  旋转门表划分为两部分：下半部分存 **upimutextab[ ]** 锁，上半部分存其他锁。这样设计的理由是 **SOBJ_USER_PI** 锁带来一个特殊问题：在调用 **turnstile_block( )** 线程在自己的 **SOBJ_USER_PI** 锁阻塞并完成对整个阻塞链的优先级授予前，不能释放送给该函数的 **upimutextab[ ]** 锁。在这一点，调用者的 **t_lockp** 将是一个旋转门锁。如果 **mutex_exit( )** 发现该 **upimutextab[]** 有等待者，那就必须唤醒它们，这迫使我们保证锁顺序：在 **mutex_vector_exit( )** 里要为该 **upimutextab[ ]** 锁获取旋转门锁，这将最终调入 **turnstile_pi_waive( )**，而后要求获取调用者的线程锁，此时也就是该**SOBJ_USER_PI** 锁的旋转门锁。一般而言，要同时获得两个旋转门锁，必须保证锁顺序。为防止在 **turnstile_pi_waive( )** 里死锁，我们必须保证 **upimutextab[ ]** 中哈希到比其他锁更低的地址

  如果你觉得这种解决办法太没意思，请赐教。

代码

代码就是一段宏定义（**UPI** 指其他，加**TURNSTILE_HASH_SIZE**）

```
#define TURNSTILE_HASH_SIZE     128             /* must be power of 2 */
#define TURNSTILE_HASH_MASK   (TURNSTILE_HASH_SIZE - 1)
#define TURNSTILE_SOBJ_HASH(sobj)       \
        ((((ulong_t)sobj >> 2) + ((ulong_t)sobj >> 9)) & \
        TURNSTILE_HASH_MASK)
#define TURNSTILE_SOBJ_BUCKET(sobj)             \
        ((IS_UPI(sobj) ? 0 : TURNSTILE_HASH_SIZE) + \
          TURNSTILE_SOBJ_HASH(sobj))
#define TURNSTILE_CHAIN(sobj)  \
        turnstile_table[TURNSTILE_SOBJ_BUCKET(sobj)]

typedef struct turnstile_chain {
        turnstile_t      *tc_first;      /* first turnstile on hash chain */
        disp_lock_t      tc_lock;        /* lock for this hash chain */
} turnstile_chain_t;
```

- 系统按时发布。**7**年过去，没人提出更好的方案。

回顾

- 下面几条工程经验值得注意
- 要及早做实现

  设计和实现阶段的认真思考都未能预见有关的问题，只有做了实现后在 **bug** 里看到，才逐渐认清了事情后面的真正本质

- 要千锤百炼

  如果原来的实现者早做过压力测试，问题可能早就发现了

- 关注边界情况

  为什么年轻工程师要努力学会调试复杂的系统？因为这方面技术将使人终身受益。分析一个问题的解，关注它在什么情况下可能出问题而不是怎样情况下能用，就特别需要关注边界情况

  在构想新软件时，软件工程师不要去说服自己所用的设计能行，而要去排除使之不行的原因。这不是要大家推迟写代码，而是提醒大家：为某个项目写下的第一部分代码，其中的**bug** 就可能推翻整个项目的设计想法

总结

- 遵循这些原则，在构造系统时，应该先实现系统总最困难的部分，然后把它们放入基础设施，确认基础设施能（或继续能）工作。这样也不可能排除污水，但会保证危险的东西能及早被发现，以便有时间修改系统的设计，从而挽救整桶美酒

讨论时间