优先级反转那点事儿

实时操作系统的一个基本要求就是基于优先级的抢占系统。保证优先级高的线程在"第一时间"抢到执行权，是实时系统的第一黄金准则。

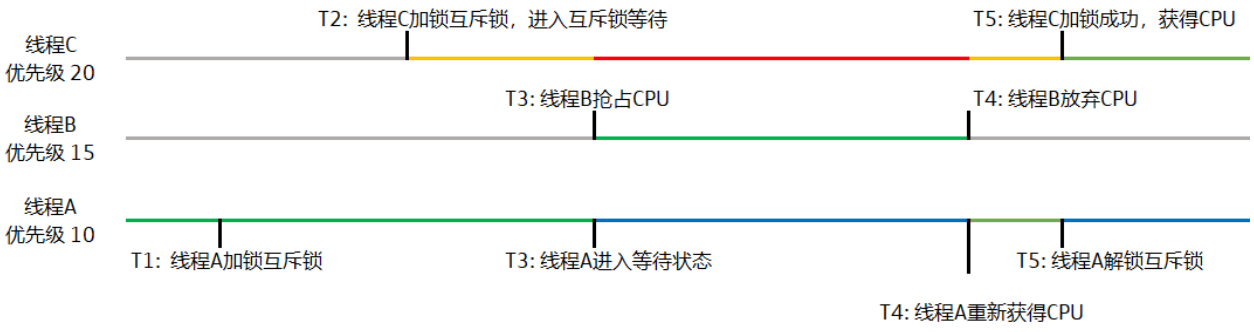
但是这种基于优先级抢占的系统，有一个著名的问题需要关注，就是"优先级反转"（Priority Inversion），简单来说，就是有低优先级的线程占据了 CPU，妨碍了高优先级线程的执行。"优先级反转"是几乎每一个实时操作系统的噩梦，系统设计上花了很多精力去关注，但依然会出错。现代最出名的例子，就是 1997 美国宇航局的火星探路车（Mars Pathfinder）了。

这个例子是如此经典，网上一搜一大把。基本情况就是火星探路车在登陆火星后的一段时间里无法工作，最后查明是因为优先级反转导致探路车的计算机不断重启。总算最后 NASA 远程打了个补丁上去，解决了这个问题。这个耗时三年，花了 2.6 亿美金的项目差点就折在小小的"优先级反转"上了。

所以今天就让我们看看优先级反转是怎样发生的，以及对于 QNX 这样的基于消息传递的操作系统有什么影响。

先看看什么是"优先级反转"

下面这个时序图就是一个经典的优先级反转



- 线程 A 在一个比较低的优先级上工作, 假设是 10 吧。然后在时间点 T1 的时候，线程 A 锁定了一把互斥锁，并开始操作互斥数据。

- 这时有个高优线级线程 C（比如优先级 20）在时间点 T2 被唤醒，它也也需要操作互斥数据。当它加锁互斥锁时，因为互斥锁在 T1 被线程 A 锁掉了，所以线程 C 放弃 CPU 进入阻塞状态，而线程 A 得以占据 CPU，继续执行。
- 事情到这一步还是正确的，虽然优先级 10 的 A 线程看上去抢了优先级 20 的 C 线程的时间，但因为程序逻辑，C 确实需要退出 CPU 等 A 完成互斥数据操作后，才能获得 CPU。
- 但是，假设我们有个线程 B 在优先级 15 上，在 T3 时间点上醒了过来，因为他比当前执行的线程 A 优先级高，所以它会立即抢占 CPU。而线程 A 被迫进入 READY 状态等待。
- 一直到时间点 T4，线程 B 放弃 CPU，这时优先级 10 的线程 A 是唯一 READY 线程，它再次占据 CPU 继续执行，最后在 T5 解锁了互斥锁。
- 在 T5，线程 A 解锁的瞬间，线程 C 立即获取互斥锁，并在优先级 20 上等待 CPU。因为它比线程 A 的优先级高，系统立刻调度线程 C 执行，而线程 A 再次进入 READY 状态。

上面这个时序里，线程 B 从 T3 到 T4 占据 CPU 运行的行为，就是事实上的优先级反转。一个优先级 15 的线程 B，通过压制优线级 10 的线程 A，而事实上导致高优先级线程 C 无法正确得到 CPU。这段时间是不可控的，因为线程 B 可以长时间占据 CPU（即使轮转时间片到时，线程 A 和 B 都处于可执行态，但是因为 B 的优先级高，它依然可以占据 CPU），其结果就是高优先级线程 C 长时间无法得到 CPU。

上面所说的美国宇航局的火星车，就是因为有高优先级的线程被压制，从而在指定时间内无法获得 CPU，导致 “看门狗”认为系统出了无法恢复的故障，直接重启了系统。重启后系统再次进入相同状态，导致不断重启，无法正常工作。

那么正确的操作应该是什么样的呢？有 A,B,C 线程的这个系统要怎么解决优先级反转这个问题的呢？

人工防止优先级反转的方法

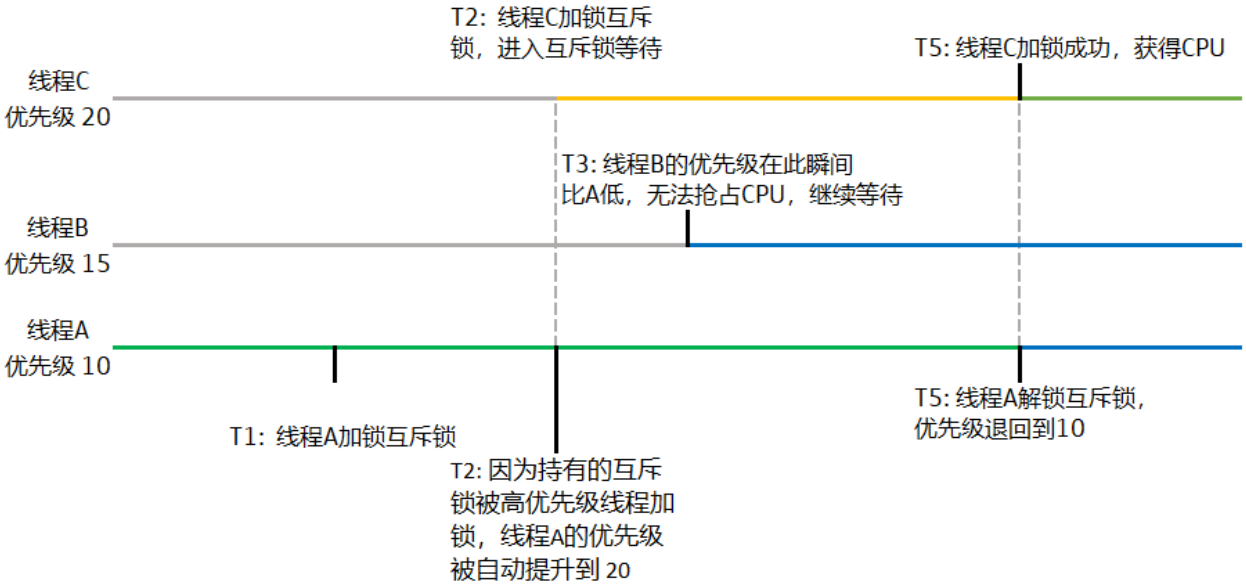
其实也不是很复杂，低优先级的 A 线程获得互斥锁前，需要先将自己的优先级临时提高，最后处理完后再退回原优先级。

```
set_priority(20);
pthread_mutex_lock();
....
pthread mutex unlock();
set_priority(10);
```

这样在 T3 的时候，线程虽然有 15 的优先级，对于仍在 20 的线程 A 无法形成压制，A 就会继续执行，直到 T5，线程 A 解锁，线程 C 立即获得互斥锁并在 20 上运行，线程 B 因为优先级低依然被压制。

当然，这里把优线级升到 20 只是特例，实际上，你需要评估所有可能上锁的线程，找到最高优先级，然后升到那里……

显然对于复杂系统这个要求过高了，事实上，在现代的实时操作系统中，这个工作是操作系统替你完成的。当高优先级线程请求互斥锁时，在我们的例子中，也就是 T2 那个瞬间，因为操作系统发现锁已经被一个低先级的线程 A 给锁了，所以它会把线程 A 的优先级临时调高（调到同 C 一样高），而在 A 解锁时，优先级再被调回原来的值。



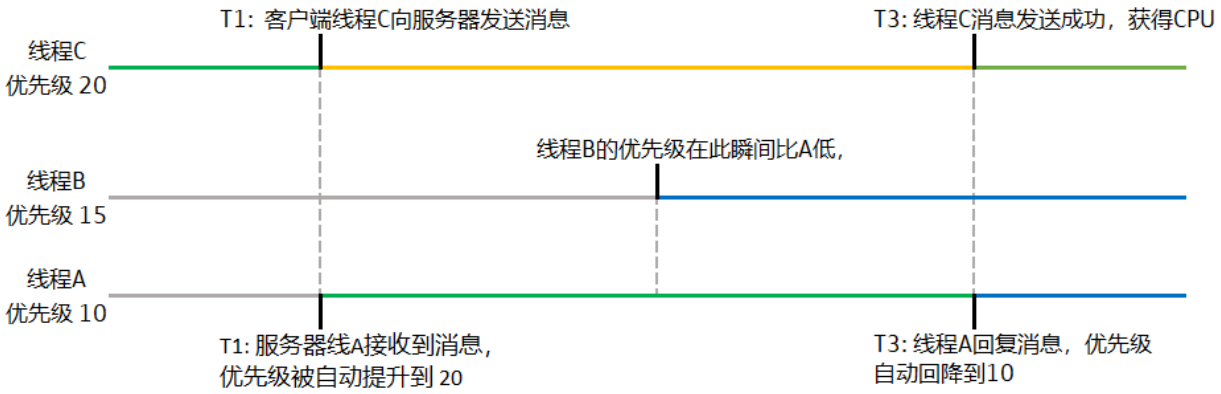
- 这里带来一个小知识点，在现代的实时操作系统上，如果需要互斥保护，应尽量使用互斥锁(mutex)。有些传统的程序员喜欢用初始值为 1 的信号灯(semaphore)。虽然在功效上这两个都能互斥，但信号灯一般系统无法做优先级继承，所以会有优先级反转的隐患。

优先级反转与 QNX

QNX 做为一个实时操作系统，自然也会对互斥锁做优先级继承，以防止线程因为用了互斥锁而发生优先级反转。但其实，QNX 有更严重的，可以导致优先级反转的机制，那就是"消息传递"。

看过《从 API 开始理解 QNX》那篇文章的人应该记得，QNX 的消息发送，接收和回复是阻塞发生的，所以，下面这个例子就很容易理解了。

假设有个高先级(20)的客户端线程 C，发了一个消息给低优先级(10)服务器线程 A；这时，C 被阻塞等待 A 的答复，A 在处理请求中忽然来了个中优先级(15)线程 B 抢占 CPU 并持续执行，A 会被阻挡，从而事实上造成了中优先级线程 B 阻挡高优线级线程 C 的现实。



消息传递是 QNX 的根本，几乎无时无刻都在发生着，当然需要保证不会发生优线级反转。方法其实也简单，当服务器线程收到信息从 MsgReceive()里退出时，线程的优先级会自动地提升到(或者下降到)发送消息的客户端线程的优先级，这样就规避了由可能由消息传递引起的优先级反转。这个也说得过去，因为服务器线程逻辑上就是因为收到了客户端请求而开始服务的，那它用客户端的优先级来进行服务是完全合理的。

举个例子，一个用户程序往在优先级 7 上，当它往文件里写东西时，write() 会向 QNX6 文件系统服务器发送消息，文件系统的服务线程就会降到优先级 7 上进行服务；最后，QNX6 文件系统（在优先级 7 上）向硬盘服务器发送消息，这时硬盘服务线程也会降到 7，然后操作硬件把数据存入硬盘。

上面只是个例子，在 QNX 上文件写入实际上是不一样的。但是，通过这个例子你可以看到，一个客户端的优先级是不断向后传递继承的。

脉冲的优先级及其继承

所以从上面的介绍可以知道，在 QNX 上，互斥锁和消息传递是都会通自动优先级继承来防止优先级反转的。不仅是这样，就连“脉冲”，也是带有优先级的；收到脉冲的线程，也会把自己调整到脉冲里的优先级。

前面提到过，客户端有时候会发一个请求给服务器端，“当这种情况发生时，请用这个事件来通知我”。比较常用的，有 InterruptAttachEvent()，可以在中断号上绑定一个事件。"事件" 是一个 struct sigevent，最常用的就是一个脉冲（SIGEV_PULSE)。也就是说，客户端的请求可以翻译成，“如果这个中断发生了，给我发这个脉冲”。

你现在知道了，脉冲里是带有优先级的。所以如果你写一个串口驱动，虽然用高优先级（比如 24）启动了驱动，但是用 InterruptAttachEvent() 错误地绑了一个优先级 10 的脉冲，就会导致当中断发生时，你的驱动就会以优先级 10 进行中断处理，这显然不是你所要的。正确的做法，是在绑事件前，把线程自己本身的优先级找出来，用这个优先级初始化脉冲事件，从而保证在正确的优先级上进行中断处理。

QNX 上性能优化与优先级继承

在 QNX 系统开发后期，很多人会面临一个系统性能优化的过程。如果你以为反正就是看看谁性能太差，把它的优先级提一下就好了，那实在是大错特错了。从上述关于优先级继承的例子可以看出，在 QNX 上优先级是"牵一发而动全身"的。有可能你改了某个客户端的优线级，这个优先级会通过消息传递一级一级地传递出去；也有可能，你改的是某个服务器线程的优先级，虽然通过 pidin 你看到它在优先级 22 上 RECEIVE，但其实一旦收到消息，它会立刻调整自己的优先级，所以修改服务器 MsgReceive()线程的优先级是没有什么意义的。

正确的做法一般是，先让各系统都按默认优先级运行。然后针对有问题的部份，用 instrument kernel 生成 kernel trace，通过 kenrel trace 来分析在相关时间段内，各个进程的优先级情况，然后对关键进程进行优先级调整。不断重复上述步骤，以达到系统最优状态。