

Linux htb 源代码分析

目录

前言.....	1
一、参考资料.....	1
二、印证一下理解.....	2
三、几个难点.....	3
后记.....	5

前言

关于 htb 是什么，怎么使用，Linux Qdisc 管理，tc 命令等，不是本文关注的。如果你对 htb 的代码实现感兴趣，请继续往下看。否则可以省点时间，干点别的了。

以前写文章，我会事无巨细，一步步跟踪，这一次我打算换一个写法，只列其大概，然后只点出我认为最难理解的部分逻辑。列其大概的目的，一是减少我的时间，二是读代码这东西，一定要躬行，这样才能深刻；点出难点的目的，是为了和高手进行印证，因为我对自己的理解也不是百分之百把握。代码版本 v3.14-rc3。

一、参考资料

1. 空闲资源流控算法

<http://blog.csdn.net/wwwlkk/article/details/5929404>

这篇文章表面上看与 htb 没太大关系，但是它是 htb 实现的理论基础，值得看看。

2. htb 简单使用

<http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm>

这一篇讲怎样使用 htb，如果你看完了，想了解 htb 在 Linux 内核中是怎样实现的，继续看。

3. htb 实现原理

<http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm>

这一篇讲实现原理，出自 htb 的开发者之手，相当权威和珍贵。如果对原理还有些不明白的地方，继续往下看。

4. htb 实现原理之中文版

<http://blog.chinaunix.net/uid-7220314-id-208698.html>

这篇博文分析的很清楚，不论是代码还是文字，都很清楚

5. htb 代码注释

<http://blog.chinaunix.net/uid-127037-id-2919586.html>

这篇博文的代码比较旧了，但是仍值得一看，虽然排版比较乱。

看完这些参考资料，如果你觉得理论上都明白了，那么恭喜你，可以去看代码实现了。

然后你就**仔细**研究 net/sched/sch_htb.c 这不到 2000 行代码。注意，仔细我可是用红色进行重点强调了。阅读代码如读书、体会人生是一样的，个中三昧，自己慢慢品，才能深刻体会。

二、印证一下理解

阅读代码的同时，请关注一下的几个问题，我们印证一下彼此理解是否正确：

1. class 分层的实现——哈希表 or 红黑树？
2. 三种红黑树——row, feed 和 wait_pq 各自作用
3. DRR 是怎么实现的？
4. struct htb_prio 中的 ptr 指针有什么作用？
5. htb_lookup_leaf()函数的逻辑看明白了吗？

下面公布我的答案：

1. class 分层的实现——哈希表
2. 三种红黑树——row, feed 和 wait_pq 各自作用

row 存放 green 节点，就是 token 额度还有剩余、可以进行调度的 class 节点；
feed 存放其子孙是 yellow 的节点，子孙需要向父类借额度，才可以进行调度；
wait_pq 存放 yellow 和 red 节点，在特定时间进行检查是否可以把节点恢复到 green。

3. DRR 是怎么实现的？

所谓 deficit round robin，是在 htb_dequeue_tree()函数的末尾实现的。

```
860     if (likely(skb != NULL)) {
861         bstats_update(&cl->bstats, skb);
862         cl->un.leaf.deficit[level] -= qdisc_pkt_len(skb);
863         if (cl->un.leaf.deficit[level] < 0) {
864             cl->un.leaf.deficit[level] += cl->quantum;
865             htb_next_rb_node(level ? &cl->parent->un.inner.clprio[prio].ptr :
866                             &q->hlevel[0].hprio[prio].ptr);
867         }
874     }
```

4. struct htb_prio 中的 ptr 指针有什么作用？

ptr 是 DRR 算法的一个标记，指向当前可以进行调度的节点（类）。如果当前节点的 deficit 用完了，htb_next_rb_node()会将 ptr 指针指向当前节点的下一个节点，然后再从 ptr 指向的节点进行调度。

5. htb_lookup_leaf()函数的逻辑看明白了吗？

该函数有两个作用，第一个是主要的，就是查找一个可以进行调度的叶子节点。我们知道 dequeue 包必须从叶子节点。

第二个作用，是“顺便”实现了 DRR 算法的一个关键部分，即 ptr 指针的设置。每次查找时，如果 ptr 已经设置，则继续从 ptr 指向的节点取包，直到该节点额度用完或者状态发生变化，然后 ptr 指向红黑树的下一个节点。这里的节点，既可以指叶子节点，也可以指中间节点，甚至根节点。

1. 分层

1 HFEGI 为叶子节点，都为 yellow，优先级都相同。CBD 都为中间节点，都为 green，优先级也都相同。A 是根节点，为 green。

3 EFG 组成一个红黑树, B->feed.root 指向树根 E。

4 H I 也分别是一棵树，算然只有一个节点。C->feed.root 指向 H，D->feed.root 指向 I。

2. htb_lookup_leaf()代码分析

```

750 /**
751  * htb_lookup_leaf - returns next leaf class in DRR order
752  *
753  * Find leaf where current feed pointers points to.
754  */
755 static struct htb_class *htb_lookup_leaf(struct htb_prio *hprio, const int prio)
756 {
757     int i;

```

```

758 struct {
759     struct rb_node *root;
760     struct rb_node **pptr;
761     u32 *pid;
762 } stk[TC_HTB_MAXDEPTH], *sp = stk;
763
764 BUG_ON(!hprio->row.rb_node);
765 sp->root = hprio->row.rb_node;
766 sp->pptr = &hprio->ptr;
767 sp->pid = &hprio->last_ptr_id;
768
769 for (i = 0; i < 65535; i++) {
770     if (!*sp->pptr && *sp->pid) {
771         /* ptr was invalidated but id is valid - try to recover
772          * the original or next ptr
773          */
774         *sp->pptr =
775             htb_id_find_next_upper(prio, sp->root, *sp->pid);
776     }
777     *sp->pid = 0; /* ptr is valid now so that remove this hint as it
778                  * can become out of date quickly
779                  */
780     if (!*sp->pptr) { /* we are at right end; rewind & go up */
781         *sp->pptr = sp->root;
782         while ((*sp->pptr)->rb_left)
783             *sp->pptr = (*sp->pptr)->rb_left;
784         if (sp > stk) {
785             sp--;
786             if (!*sp->pptr) {
787                 WARN_ON(1);
788                 return NULL;
789             }
790             htb_next_rb_node(sp->pptr);
791         }
792     } else {
793         struct htb_class *cl;
794         struct htb_prio *clp;
795
796         cl = rb_entry(*sp->pptr, struct htb_class, node[prio]);
797         if (!cl->level)
798             return cl;
799         clp = &cl->un.inner.clprio[prio];
800         (++sp)->root = clp->feed.rb_node;
801         sp->pptr = &clp->ptr;
802         sp->pid = &clp->last_ptr_id;
803     }
804 }
807 }

```

htb_lookup_leaf()返回优先级是prio的叶子节点。现在假设所有的ptr指针都为NULL，然后类图如图1所示，假设叶子节点HFEGI都有包。

1 level0中没有green节点，htb_dequeue()会在level1中查找，第一次调用htb_lookup_leaf()。

2 因为此时level1->row.ptr为NULL，故Line 783找到BCD树的最左下节点C，并设置level1->row.ptr = C。

3 因为C是中间节点，故Line 799继续在C的feed树中查找。

4 此时C->feed.ptr为NULL，在H树中查找最左下节点H，并设置C->feed.ptr = H。并且此时Line 784的条件满足，故设置level1->row.ptr = B。

5 此时C->feed.ptr = H，所以htb_lookup_leaf()返回H。

6 然后第二次调用htb_lookup_leaf()。因为此时level1->row.ptr为B，并且B为中间节点，故Line 799在EFG树中继续查找。

8 因为此时B->feed.ptr为NULL，故Line 793找到EFG树的最左下节点F，并设置B->feed.ptr为F。htb_lookup_leaf()返回F。

9 然后第三次调用htb_lookup_leaf()，顺着B找到F。以后每次都是返回F，直到F的deficit用完了，会在htb_dequeue_tree()中调用htb_next_rb_node()，将B->feed.ptr设置为E。

10 以后每次都是顺着B返回E，直到E的deficit用完了，会在htb_dequeue_tree()中调用htb_next_rb_node()，将B->feed.ptr设置为G。

11 当G的deficit用完后，htb_next_rb_node()会将B->feed.ptr设置为NULL。

12 下一次调用htb_lookup_leaf()时，因为level1->row.ptr为B，所以先找到B。B又是中间节点，并且因为B->feed.ptr为NULL，故Line 783找到最左下节点F，并设置B->feed.ptr = F。此时Line 784的条件满足，故设置level1->row.ptr = D。

13 本次htb_lookup_leaf()调用返回F。

14 下一次调用htb_lookup_leaf()，因为level1->row.ptr等于D，并且D为中间节点，故继续在I树中查找。然后将B->feed.ptr设置为I，并将level1->row.ptr设置为NULL（D的下一个元素）。

15 以后的查找，与前边的步骤类似。通过ptr，我们实现了中序遍历红黑树，并通过deficit实现了DRR算法。

16 如果dequeue期间，节点的颜色发生了变化，那么节点就会从相应的红黑树中删除，并丢失ptr指针。此时我们通过last_ptr_id记录被删除的节点的classid，下次调用htb_lookup_leaf()时在Line 775调用htb_id_find_next_upper()将ptr指针指向被删除节点的下一个节点，继续DRR。

如果假设每个节点的deficit都只够发送一个包，则轮讯节点次序为

C H B F B E B G B F D I D I C H B F B E B G B F D I D I 这样循环下去。既实现了绿色的中间节点BCD的DRR，也实现了中间节点的子树的DRR，运用之妙，存乎一心。

后记

读到这里，如果你还没有理解htb的实现，再多多用功吧。我认为最复杂的部分已经讲完了，当然还有其他的知识点，像tc工具与内核的交互、token计算等等，慢慢看，自然就看懂了。