

操作系统实验 3-3

AVL 树转红黑树问题

马栩杰

2014011085

无 43 班

2016 年 12 月 4 日

1 问题描述

在 Windows 的虚拟内存管理中，将 VAD 组织成 AVL 树。VAD 树是一种平衡二叉树。

红黑树也是一种自平衡二叉查找树，在 Linux 2.6 及其以后版本的内核中，采用红黑树来维护内存块。

请尝试参考 Linux 源代码将 WRK 源代码中的 VAD 树由 AVL 树替换成红黑树。

2 Linux 中的红黑树

此时实验中，参考的 Linux 内核版本是 4.8，也就是目前最新的一个稳定版¹。

在 Linux 项目中全局查找关键字 rbtree，发现 Linux 的红黑树的实现在 lib/rbtree.c² 中，其接口定义在 include/linux/rbtree.h³ 中。另外，还在项目中惊喜地发现了关于红黑树的文档 Documentation/rbtree.txt⁴。

参考红黑树的文档以及代码，我们可以知道 Linux rbtree 实现的细节。

- 使用红黑树的每个数据结点都是一个包含 rb_node 的结构体

```
1 struct mytype {  
2     struct rb_node node;  
3     char *keystring;  
4 };
```

- 红黑树的结点 rb_node 定义包括结点的颜色和左右子结点

```
1 struct rb_node {  
2     unsigned long __rb_parent_color;  
3     struct rb_node *rb_right;
```

¹GitHub 上可以看到这个 Release 发布于 2016.10.03，迄今发布的更新的版本都是 RC 版

²<https://github.com/torvalds/linux/blob/master/lib/rbtree.c>

³<https://github.com/torvalds/linux/blob/master/include/linux/rbtree.h>

⁴<https://github.com/torvalds/linux/blob/master/Documentation/rbtree.txt>

```

4     struct rb_node *rb_left;
5 } __attribute__((aligned(sizeof(long))));
6     /* The alignment might seem pointless, but allegedly CRIS needs it */

```

- 红黑树还提供了一系列基本操作

```

1 // 查找亲结点
2 #define rb_parent(r)    ((struct rb_node *)((r)->__rb_parent_color & ~3))
3
4 // 设置根结点
5 #define RB_ROOT    (struct rb_root) { NULL, }
6
7 // 结点存在性检测
8 #define  rb_entry(ptr, type, member) container_of(ptr, type, member)
9
10 // 空树检测
11 #define RB_EMPTY_ROOT(root)    (READ_ONCE((root)->rb_node) == NULL)
12
13 /* 'empty' nodes are nodes that are known not to be inserted in an rbtree */
14 #define RB_EMPTY_NODE(node)    \
15     ((node)->__rb_parent_color == (unsigned long)(node))
16 #define RB_CLEAR_NODE(node)    \
17     ((node)->__rb_parent_color = (unsigned long)(node))
18
19
20 // 颜色平衡操作
21 extern void rb_insert_color(struct rb_node *, struct rb_root *);
22 extern void rb_erase(struct rb_node *, struct rb_root *);
23
24 /* Find logical next and previous nodes in a tree */
25 extern struct rb_node *rb_next(const struct rb_node *);
26 extern struct rb_node *rb_prev(const struct rb_node *);
27 extern struct rb_node *rb_first(const struct rb_root *);
28 extern struct rb_node *rb_last(const struct rb_root *);
29
30 /* Postorder iteration - always visit the parent after its children */
31 extern struct rb_node *rb_first_postorder(const struct rb_root *);
32 extern struct rb_node *rb_next_postorder(const struct rb_node *);
33
34 /* Fast replacement of a single node without remove/rebalance/add/rebalance */
35 extern void rb_replace_node(struct rb_node *victim, struct rb_node *new,
36                             struct rb_root *root);
37 extern void rb_replace_node_rcu(struct rb_node *victim, struct rb_node *new,

```

```

38     struct rb_root *root);
39
40 // 插入新结点
41 static inline void rb_link_node(struct rb_node *node, struct rb_node *parent,
42     struct rb_node **rb_link)
43 {
44     node->__rb_parent_color = (unsigned long)parent;
45     node->rb_left = node->rb_right = NULL;
46
47     *rb_link = node;
48 }

```

- 另外在 `include/linux/rbtree_augmented.h`⁵ 中也有一些更具体的红黑树访问接口

```

1  #define __rb_parent(pc)      ((struct rb_node *) (pc & ~3))
2  #define __rb_color(pc)      ((pc) & 1)
3  #define __rb_is_black(pc)   __rb_color(pc)
4  #define __rb_is_red(pc)     (!__rb_color(pc))
5  #define rb_color(rb)        __rb_color((rb)->__rb_parent_color)
6  #define rb_is_red(rb)       __rb_is_red((rb)->__rb_parent_color)
7  #define rb_is_black(rb)     __rb_is_black((rb)->__rb_parent_color)
8
9  static inline void rb_set_parent(struct rb_node *rb, struct rb_node *p) {
10     rb->__rb_parent_color = rb_color(rb) | (unsigned long)p;
11 }
12
13 static inline void rb_set_parent_color(struct rb_node *rb,
14     struct rb_node *p, int color) {
15     rb->__rb_parent_color = (unsigned long)p | color;
16 }
17
18 static inline void
19 __rb_change_child(struct rb_node *old, struct rb_node *new,
20     struct rb_node *parent, struct rb_root *root) {
21     if (parent) {
22         if (parent->rb_left == old)
23             WRITE_ONCE(parent->rb_left, new);
24         else
25             WRITE_ONCE(parent->rb_right, new);
26     } else
27         WRITE_ONCE(root->rb_node, new);
28 }

```

⁵https://github.com/torvalds/linux/blob/master/include/linux/rbtree_augmented.h

```

29
30 extern void __rb_erase_color(struct rb_node *parent, struct rb_root *root,
31 void (*augment_rotate)(struct rb_node *old, struct rb_node *new));

```

代码中比较奇怪的地方是结点的亲结点和颜色通过位运算存储在 `__rb_parent_color` 这一个变量中，不知道作者写的时候是怎么考虑的。

其中最关键的几个操作是向树中插入删除结点以及颜色平衡，在代码中都有明确的函数接口可供外部调用。

3 WRK 中的 AVL 树

同样在 WRK 代码中查找关键字 AVL，在 `base/ntos/inc/ps.h` 中可以看到 AVL 树与结点的结构体定义：

```

1 typedef struct _MM_AVL_TABLE { // 定义 AVL 树结构
2     MMADDRESS_NODE BalancedRoot; // 树根
3     ULONG_PTR DepthOfTree: 5; // 深度
4     ULONG_PTR Unused: 3;
5     #if defined (_WIN64)
6     ULONG_PTR NumberGenericTableElements: 56;
7     #else
8     ULONG_PTR NumberGenericTableElements: 24;
9     #endif
10    PVOID NodeHint;
11    PVOID NodeFreeHint;
12 } MM_AVL_TABLE, *PMM_AVL_TABLE;
13
14
15 typedef struct _MMADDRESS_NODE { // 定义 AVL 结点结构
16     union {
17         LONG_PTR Balance : 2; // 平衡度
18         struct _MMADDRESS_NODE *Parent; // 亲结点
19     } u1;
20     struct _MMADDRESS_NODE *LeftChild; // 左子结点
21     struct _MMADDRESS_NODE *RightChild; // 右子结点
22     ULONG_PTR StartingVpn;
23     ULONG_PTR EndingVpn;
24 } MMADDRESS_NODE, *PMMADDRESS_NODE;

```

可以看到，这里的 `MMADDRESS_NODE` 结构与 `rb_node` 的结构是类似的。基于这一点，我们可以通过将 Linux 红黑树接口中的 `rb_node` 简单地替换为 `MMADDRESS_NODE` 来实现代码移植。另外这里同样出现了一个奇怪的变量 `u1`，用一个 32 位长度的内存地址存放一个 union，union 中同时保存树的平衡度和亲结点地址，这与 `rb_node` 中的 `_rb_parent_color` 做了非常类似的事情。为什么两个

不同的操作系统内核会在这种奇怪的地方出现微妙的相似性呢？原来这里所做的事情是用地址变量的最后两位来存储结点的平衡度 (WRK) 或者结点的颜色 (Linux)，由于地址总是 4 的倍数，所以地址的最后两位永远是 0，于是将平衡度或者颜色与亲结点地址存在一起可以节省一个变量的内存空间... 当然好的一点是我们在移植代码的过程中，我们可以用 `u1` 这个变量来实现 `_rb_parent_color` 的功能，而避免再重新定义新的变量。

另外，发现 Windows 代码中有两处 AVL 树接口的定义，分别在 `base/ntos/rtl/avltable.c` 和 `base/ntos/mm/addrsup.c` 中。参考 `addrsup.c` 文件起始的注释：

```
1  /*
2      This module implements a new version of the generic table package
3      based on balanced binary trees (later named AVL), as described in
4      Knuth, "The Art of Computer Programming, Volume 3, Sorting and Searching",
5      and refers directly to algorithms as they are presented in the second
6      edition Copyrighted in 1973.
7
8      Used rtl\avltable.c as a starting point, adding the following:
9      - Use less memory for structures as these are nonpaged & heavily used.
10     - Caller allocates the pool to reduce mutex hold times.
11     - Various VAD-specific customizations/optimizations.
12     - Hints.
13 */
```

我们了解到 `addrsup.c` 是对 `avltable.c` 的改进版本，其算法基于 Knuth TAOCP 中对 AVL 树的描述编写。于是，确定了需要参考 & 修改的文件是 `addrsup.c`。

找到 `addrsup.c` 中的 AVL 树接口函数

```
1  // 删除树结点 (806 行)
2  VOID
3  FASTCALL
4  MiRemoveNode (
5      IN PMMADDRESS_NODE NodeToDelete,
6      IN PMM_AVL_TABLE Table
7  )
8
9  // 插入树结点 (1293 行)
10 VOID
11 FASTCALL
12 MiInsertNode (
13     IN PMMADDRESS_NODE NodeToInsert,
14     IN PMM_AVL_TABLE Table
15 )
```

根据平衡二叉树的 ADT 接口我们知道，只需要将这两个函数用红黑树的版本替换即可。

4 代码移植

在弄清楚 Linux 和 WRK 的代码实现之后，就可以开始修改内核啦！

4.1 文件头部的宏和短函数

首先，参照 `rbtree_augmented.h` 和 `rbtree.h` 中的实现，在 `addrsup.c` 的前端加入一些进行插入和删除操作所需要的宏和一些短函数）：

```
1 // 为了保证 rbtree 与 AVL 初始化时的一致性
2 // 我们将 black 定义为 0, red 定义为 1
3 #define RB_BLACK 0
4 #define RB_RED 1
5
6 #define __rb_parent(pc) ((PMMADDRESS_NODE)((long)(pc) & ~3))
7 #define rb_parent(rb) (SANITIZE_PARENT_NODE((rb)->u1.Parent))
8 #define rb_red_parent(rb) (rb_parent(rb))
9
10 #define __rb_color(pc) ((long)(pc) & 1)
11 #define __rb_is_black(pc) (!__rb_color(pc))
12 #define __rb_is_red(pc) __rb_color(pc)
13 #define rb_color(rb) __rb_color((rb)->u1.Parent)
14 #define rb_is_red(rb) __rb_is_red((rb)->u1.Parent)
15 #define rb_is_black(rb) __rb_is_black((rb)->u1.Parent)
16 #define rb_set_red(rb) ((rb)->u1.Balance = RB_RED)
17 #define rb_set_black(rb) ((rb)->u1.Balance = RB_BLACK)
18
19 // 这里有一个小小的问题：如果按照 Linux 原本的函数定义，这里的几个函数都应该是 inline
20 // 的，然而实际执行 nmake 编译的时候发现 inline 会报错。这应该与编译器实现有关。
21 // 于是就把它们的 inline 给去掉了
22 static void rb_set_parent(PMMADDRESS_NODE rb, PMMADDRESS_NODE p) {
23     rb->u1.Parent = (PMMADDRESS_NODE)((long)p | rb_color(rb));
24 }
25
26 static void rb_set_parent_color(PMMADDRESS_NODE rb,
27                                PMMADDRESS_NODE p, int color) {
28     // 将颜色和亲结点地址一起存进 u1
29     rb->u1.Parent = (PMMADDRESS_NODE)((long)p | (long)color);
30 }
31
32 static void
33 __rb_change_child(PMMADDRESS_NODE old, PMMADDRESS_NODE new_,
34                  PMMADDRESS_NODE parent, PMMADDRESS_NODE root) {
35     if (parent) {
```

```

36         if (parent->LeftChild == old)
37             parent->LeftChild = new_;
38         else
39             parent->RightChild = new_;
40     } else
41         root->RightChild = new_;
42 }
43
44 static void
45 __rb_rotate_set_parents(PMMADDRESS_NODE old, PMMADDRESS_NODE new_,
46                         PMMADDRESS_NODE root, int color) {
47     PMMADDRESS_NODE parent = rb_parent(old);
48     new_>u1.Parent = old->u1.Parent;
49     rb_set_parent_color(old, new_, color);
50     __rb_change_child(old, new_, parent, root);
51 }

```

4.2 删除结点

然后加入删除结点的函数：

```

1  static PMMADDRESS_NODE
2  __rb_erase_augmented(PMMADDRESS_NODE node, PMMADDRESS_NODE root)
3  {
4      // 基本上原封不动地照搬 Linux
5      PMMADDRESS_NODE child = node->RightChild;
6      PMMADDRESS_NODE tmp = node->LeftChild;
7      PMMADDRESS_NODE parent, rebalance;
8      PMMADDRESS_NODE pc;
9
10     if (!tmp) {
11         /*
12          * Case 1: node to erase has no more than 1 child (easy!)
13          *
14          * Note that if there is one child it must be red due to 5)
15          * and node must be black due to 4). We adjust colors locally
16          * so as to bypass __rb_erase_color() later on.
17          */
18         pc = node->u1.Parent;
19         parent = __rb_parent(pc);
20         __rb_change_child(node, child, parent, root);
21         if (child) {
22             child->u1.Parent = pc;

```

```

23         rebalance = NULL;
24     } else
25         rebalance = __rb_is_black(pc) ? parent : NULL;
26     tmp = parent;
27 } else if (!child) {
28     /* Still case 1, but this time the child is node->LeftChild */
29     tmp->u1.Parent = pc = node->u1.Parent;
30     parent = __rb_parent(pc);
31     __rb_change_child(node, tmp, parent, root);
32     rebalance = NULL;
33     tmp = parent;
34 } else {
35     PMMADDRESS_NODE successor = child, child2;
36
37     tmp = child->LeftChild;
38     if (!tmp) {
39         /*
40          * Case 2: node's successor is its right child
41          *
42          *      (n)          (s)
43          *      / \          / \
44          *      (x) (s)  ->  (x) (c)
45          *          \
46          *          (c)
47          */
48         parent = successor;
49         child2 = successor->RightChild;
50     } else {
51         /*
52          * Case 3: node's successor is leftmost under
53          * node's right child subtree
54          *
55          *      (n)          (s)
56          *      / \          / \
57          *      (x) (y)  ->  (x) (y)
58          *      /          /
59          *      (p)          (p)
60          *      /          /
61          *      (s)          (c)
62          *      \
63          *      (c)
64          */
65     do {

```



```

66         parent = successor;
67         successor = tmp;
68         tmp = tmp->LeftChild;
69     } while (tmp);
70     child2 = successor->RightChild;
71     parent->LeftChild = child2;
72     successor->RightChild = child;
73     rb_set_parent(child, successor);
74 }
75
76 tmp = node->LeftChild;
77 successor->LeftChild = tmp;
78 rb_set_parent(tmp, successor);
79
80 pc = node->u1.Parent;
81 tmp = __rb_parent(pc);
82 __rb_change_child(node, successor, tmp, root);
83
84 if (child2) {
85     successor->u1.Parent = pc;
86     rb_set_parent_color(child2, parent, RB_BLACK);
87     rebalance = NULL;
88 } else {
89     PMMADDRESS_NODE pc2 = successor->u1.Parent;
90     successor->u1.Parent = pc;
91     rebalance = __rb_is_black(pc2) ? parent : NULL;
92 }
93 tmp = successor;
94 }
95
96 return rebalance;
97 }
98
99 static void
100 ____rb_erase_color(PMMADDRESS_NODE parent, PMMADDRESS_NODE root)
101 {
102     PMMADDRESS_NODE node = NULL, sibling, tmp1, tmp2;
103
104     while (TRUE) { // 谜之问题：编译器无法识别 true, 只能写成 TRUE
105         /*
106          * Loop invariants:
107          * - node is black (or NULL on first iteration)
108          * - node is not the root (parent is not NULL)

```

```

109     * - All leaf paths going through parent and node have a
110     *   black node count that is 1 lower than other leaf paths.
111     */
112 sibling = parent->RightChild;
113 if (node != sibling) { /* node == parent->LeftChild */
114     if (rb_is_red(sibling)) {
115         /*
116          * Case 1 - left rotate at parent
117          *
118          *      P              S
119          *     / \            / \
120          *    N  s    -->   p   Sr
121          *           / \       / \
122          *          Sl Sr     N  Sl
123          */
124         tmp1 = sibling->LeftChild;
125         parent->RightChild = tmp1;
126         sibling->LeftChild = parent;
127         rb_set_parent_color(tmp1, parent, RB_BLACK);
128         __rb_rotate_set_parents(parent, sibling, root,
129                                 RB_RED);
130         sibling = tmp1;
131     }
132     tmp1 = sibling->RightChild;
133     if (!tmp1 || rb_is_black(tmp1)) {
134         tmp2 = sibling->LeftChild;
135         if (!tmp2 || rb_is_black(tmp2)) {
136             /*
137              * Case 2 - sibling color flip
138              * (p could be either color here)
139              *
140              *      (p)          (p)
141              *     / \          / \
142              *    N  S    -->  N   s
143              *           / \       / \
144              *          Sl Sr     Sl Sr
145              *
146              * This leaves us violating 5) which
147              * can be fixed by flipping p to black
148              * if it was red, or by recursing at p.
149              * p is red when coming from Case 1.
150              */
151             rb_set_parent_color(sibling, parent,

```

```

152         RB_RED);
153     if (rb_is_red(parent))
154         rb_set_black(parent);
155     else {
156         node = parent;
157         parent = rb_parent(node);
158         if (parent)
159             continue;
160     }
161     break;
162 }
163 /*
164  * Case 3 - right rotate at sibling
165  * (p could be either color here)
166  *
167  *      (p)          (p)
168  *     / \         / \
169  *    N  S  -->  N  Sl
170  *       / \         \
171  *      sl Sr         s
172  *                       \
173  *                       Sr
174  */
175 tmp1 = tmp2->RightChild;
176 sibling->LeftChild = tmp1;
177 tmp2->RightChild = sibling;
178 parent->RightChild = tmp2;
179 if (tmp1)
180     rb_set_parent_color(tmp1, sibling,
181                          RB_BLACK);
182 tmp1 = sibling;
183 sibling = tmp2;
184 }
185 /*
186  * Case 4 - left rotate at parent + color flips
187  * (p and sl could be either color here.
188  * After rotation, p becomes black, s acquires
189  * p's color, and sl keeps its color)
190  *
191  *      (p)          (s)
192  *     / \         / \
193  *    N  S  -->  P  Sr
194  *       / \         / \

```

```

195      *      (sl) sr      N (sl)
196      */
197      tmp2 = sibling->LeftChild;
198      parent->RightChild = tmp2;
199      sibling->LeftChild = parent;
200      rb_set_parent_color(tmp1, sibling, RB_BLACK);
201      if (tmp2)
202          rb_set_parent(tmp2, parent);
203      __rb_rotate_set_parents(parent, sibling, root,
204                              RB_BLACK);
205      break;
206  } else {
207      sibling = parent->LeftChild;
208      if (rb_is_red(sibling)) {
209          /* Case 1 - right rotate at parent */
210          tmp1 = sibling->RightChild;
211          parent->LeftChild = tmp1;
212          sibling->RightChild = parent;
213          rb_set_parent_color(tmp1, parent, RB_BLACK);
214          __rb_rotate_set_parents(parent, sibling, root,
215                                  RB_RED);
216          sibling = tmp1;
217      }
218      tmp1 = sibling->LeftChild;
219      if (!tmp1 || rb_is_black(tmp1)) {
220          tmp2 = sibling->RightChild;
221          if (!tmp2 || rb_is_black(tmp2)) {
222              /* Case 2 - sibling color flip */
223              rb_set_parent_color(sibling, parent,
224                                  RB_RED);
225              if (rb_is_red(parent))
226                  rb_set_black(parent);
227              else {
228                  node = parent;
229                  parent = rb_parent(node);
230                  if (parent)
231                      continue;
232              }
233              break;
234          }
235          /* Case 3 - right rotate at sibling */
236          tmp1 = tmp2->LeftChild;
237          sibling->RightChild = tmp1;

```

```

238         tmp2->LeftChild = sibling;
239         parent->LeftChild = tmp2;
240         if (tmp1)
241             rb_set_parent_color(tmp1, sibling,
242                                 RB_BLACK);
243         tmp1 = sibling;
244         sibling = tmp2;
245     }
246     /* Case 4 - left rotate at parent + color flips */
247     tmp2 = sibling->RightChild;
248     parent->LeftChild = tmp2;
249     sibling->RightChild = parent;
250     rb_set_parent_color(tmp1, sibling, RB_BLACK);
251     if (tmp2)
252         rb_set_parent(tmp2, parent);
253     __rb_rotate_set_parents(parent, sibling, root,
254                             RB_BLACK);
255     break;
256 }
257 }
258 }

```

4.3 修改插入结点接口

做了充分的准备工作之后，进入最关键的环节：修改 `MiInsertNode` 和 `MiRemoveNode` 两个函数：

```

1  VOID
2  FASTCALL
3  MiInsertNode (
4      IN PMMADDRESS_NODE NodeToInsert,
5      IN PMM_AVL_TABLE Table
6  )
7  {
8      PMMADDRESS_NODE NodeOrParent;
9      TABLE_SEARCH_RESULT SearchResult;
10
11      SearchResult = MiFindNodeOrParent (Table,
12                                         NodeToInsert->StartingVpn,
13                                         &NodeOrParent);
14      NodeToInsert->LeftChild = NULL;
15      NodeToInsert->RightChild = NULL;
16

```

```

17 Table->NumberGenericTableElements += 1;
18
19 if (SearchResult == TableEmptyTree) {
20
21     Table->BalancedRoot.RightChild = NodeToInsert;
22     rb_set_parent(NodeToInsert, &Table->BalancedRoot);
23     Table->DepthOfTree = 1;
24
25 }
26 else {
27
28     PMMADDRESS_NODE R = NodeToInsert;
29     PMMADDRESS_NODE S = NodeOrParent;
30     PMMADDRESS_NODE node, root, parent, gparent, tmp;
31
32     if (SearchResult == TableInsertAsLeft) {
33         NodeOrParent->LeftChild = NodeToInsert;
34     }
35     else {
36         NodeOrParent->RightChild = NodeToInsert;
37     }
38
39     rb_set_parent(NodeToInsert, NodeOrParent);
40
41     node = NodeToInsert;
42     root = &Table->BalancedRoot;
43
44     // 需要平衡的情况:
45     // 以下是从 rbtree.c __rb_insert 移植过来的部分
46     parent = rb_red_parent(node);
47
48     while (TRUE) {
49         /*
50          * Loop invariant: node is red
51          *
52          * If there is a black parent, we are done.
53          * Otherwise, take some corrective action as we don't
54          * want a red root or two consecutive red nodes.
55          */
56         if (!parent) {
57             rb_set_parent_color(node, NULL, RB_BLACK);
58             break;
59         } else if (rb_is_black(parent))

```

```

60         break;
61
62     gparent = rb_red_parent(parent);
63
64     tmp = gparent->RightChild;
65     if (parent != tmp) {      /* parent == gparent->LeftChild */
66         if (tmp && rb_is_red(tmp)) {
67             /*
68              * Case 1 - color flips
69              *
70              *      G          g
71              *    / \      / \
72              *   p  u  --> P  U
73              *   /      /
74              *  n      n
75              *
76              * However, since g's parent might be red, and
77              * 4) does not allow this, we need to recurse
78              * at g.
79              */
80             rb_set_parent_color(tmp, gparent, RB_BLACK);
81             rb_set_parent_color(parent, gparent, RB_BLACK);
82             node = gparent;
83             parent = rb_parent(node);
84             rb_set_parent_color(node, parent, RB_RED);
85             continue;
86         }
87
88         tmp = parent->RightChild;
89         if (node == tmp) {
90             /*
91              * Case 2 - left rotate at parent
92              *
93              *      G          G
94              *    / \      / \
95              *   p  U  --> n  U
96              *   \      /
97              *    n      p
98              *
99              * This still leaves us in violation of 4), the
100              * continuation into Case 3 will fix that.
101              */
102             tmp = node->LeftChild;

```

```

103     parent->RightChild = tmp;
104     node->LeftChild = parent;
105     if (tmp)
106         rb_set_parent_color(tmp, parent, RB_BLACK);
107     rb_set_parent_color(parent, node, RB_RED);
108     parent = node;
109     tmp = node->RightChild;
110 }
111
112 /*
113  * Case 3 - right rotate at gparent
114  *
115  *      G      P
116  *     / \    / \
117  *    p  U  --> n  g
118  *   /      \
119  *  n        U
120  */
121 gparent->LeftChild = tmp; /* == parent->RightChild */
122 parent->RightChild = gparent;
123 if (tmp)
124     rb_set_parent_color(tmp, gparent, RB_BLACK);
125 __rb_rotate_set_parents(gparent, parent, root, RB_RED);
126 break;
127 } else {
128     tmp = gparent->LeftChild;
129     if (tmp && rb_is_red(tmp)) {
130         /* Case 1 - color flips */
131         rb_set_parent_color(tmp, gparent, RB_BLACK);
132         rb_set_parent_color(parent, gparent, RB_BLACK);
133         node = gparent;
134         parent = rb_parent(node);
135         rb_set_parent_color(node, parent, RB_RED);
136         continue;
137     }
138
139     tmp = parent->LeftChild;
140     if (node == tmp) {
141         /* Case 2 - right rotate at parent */
142         tmp = node->RightChild;
143         parent->LeftChild = tmp;
144         node->RightChild = parent;
145         if (tmp)

```



```

146         rb_set_parent_color(tmp, parent, RB_BLACK);
147         rb_set_parent_color(parent, node, RB_RED);
148         parent = node;
149         tmp = node->LeftChild;
150     }
151
152     /* Case 3 - left rotate at gparent */
153     gparent->RightChild = tmp; /* == parent->LeftChild */
154     parent->LeftChild = gparent;
155     if (tmp)
156         rb_set_parent_color(tmp, gparent, RB_BLACK);
157     __rb_rotate_set_parents(gparent, parent, root, RB_RED);
158     break;
159 }
160 }
161 }
162
163 return;
164 }

```

4.4 修改删除结点接口

由于此前已经实现了删除结点的函数 `__rb_erase_augmented` 和 `____rb_erase_color`，这里只需要简单地调用一下就好了。

```

1  VOID
2  FASTCALL
3  MiRemoveNode (
4      IN PMMADDRESS_NODE NodeToDelete,
5      IN PMM_AVL_TABLE Table
6  )
7  {
8      PMMADDRESS_NODE root = &Table->BalancedRoot;
9      PMMADDRESS_NODE rebalance;
10     rebalance = __rb_erase_augmented(NodeToDelete, root);
11     if (rebalance) { // 需要再平衡
12         ____rb_erase_color(rebalance, root);
13     }
14
15 }

```

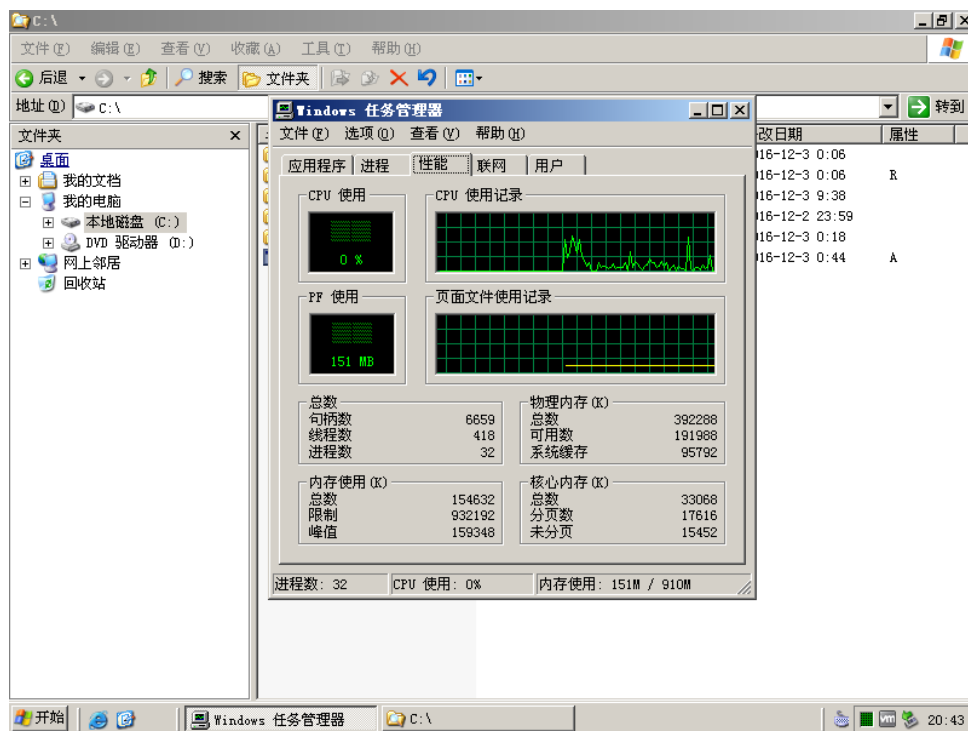
至此，代码移植就结束啦，几乎没有原创内容，仅仅是将 Linux `rb_node` 对应的接口移动到 WRK 的 `MADDRESS_NODE` 结构体上。

5 实验结果与分析

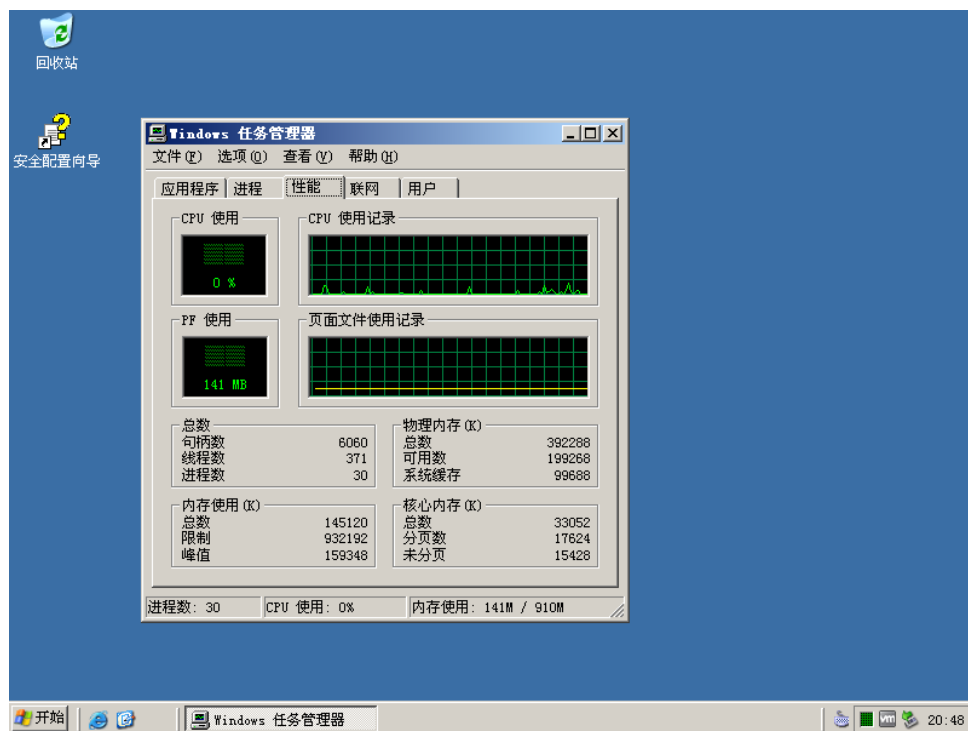
我们启动 Windows Server 2003，编译运行 WRK。

重新开机，怀着忐忑的心情选择 WRK，成功开机了！终于不用再重复一遍遍启动 WinDBG 和虚拟机调试的过程了..

用红黑树的 WRK：



用 AVL 的 WRK：



结果，移植代码后的文件系统和原本的文件系统看不出有什么显著的区别..

6 实验总结

相比前两项实验，这次的任务量明显大得多⁶..

实验中需要阅读大量操作系统的代码，这个时候一个好用的工具就非常重要。在实践中，我发现 GitHub 的全局变量查找速度非常快，所以直接在 GitHub 上阅读代码比起本地的代码阅读工具要更方便、效率更高，结果是我基本上依靠 GitHub 读完了 Linux 和 WRK 中实验涉及到内容相关的代码。

关于 Linux 和 WRK 的代码风格，总的来说，Linux 的代码命名比较简明易懂，相比之下 Windows 的代码命名更难读一些⁷。Linux 和 WRK 的代码注释都很漂亮，其中 Linux 更是在代码中贴心地插入了图形化的讲解，不了解红黑树的人甚至可以直接通过读 Linux 的源码来学习这一数据结构。从这一点来说，开源的力量是非常伟大的！

⁶关于为什么我同时也做了前两个实验：由于我以前编写过多线程程序，对前两个实验的内容比较熟悉，于是在大作业题目刚刚公布的那两周就写完了前两次实验的代码，然后直到期中之后的某一次课上才听马老师说做第三个实验的话只做一个就可以了.. 请允许我做一个悲伤的表情..TuT

⁷有一些谜之缩写，以及滥用大写字母，个人认为一看就觉得很不爽