存储管理问题
AVL 树到红黑树问题

王禹无 48 2014011241

December 1, 2016

0.1 实验描述与实验要求

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。

0.2 实验环境

本次实验的实验环境如下：

操作系统环境: WINDOWS

IDE: VMware Station

0.3 实验实现

本次实验希望能用 Linux 中管理 VAD 的红黑树替代 Windows 中管理 VAD 的 AVL 树。

0.3.1 知识背景

AVL 树

在计算机科学中，AVL 树是最先发明的自平衡二叉查找树。在 AVL 树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是 O（log n）。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。节点的平衡因子是它的左子树的高度减去它的右子树的高度（有时相反）。带有平衡因子 1、0 或 -1 的节点被认为是平衡的。带有平衡因子 -2 或 2 的节点被认为是不平衡的，并需要重新平衡这个树。平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。

——*From WikiPedia*

红黑树

红黑树是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它可以在 O(log n) 时间内做查找，插入和删除，这里的 n 是树中元素的数目。

——*From WikiPedia*

0.3.2 实验设计

首先找到 Linux 中红黑树的定义，这里我选择的是版本 4.8.11 的稳定内核。可以发现红黑树的结构和定义以及一些操作都在./include/linux/rbtree.h 与./lib/rbtree.c 中,以及./lib/rbtree.c 所依赖的./include/linux/rbtree_argument.h 中。

红黑树的节点定义如下：

```
struct rb_node {
    unsigned long  __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
```

红黑树的根定义如下：

```
struct rb_root {
    struct rb_node *rb_node;
};
```

注意一下这里红黑树根的定义和节点定义不一样，在移植代码的时候需要注意。

红黑树在 Linux 中的一些基本操作如下：

```
#define RB_RED      1
#define RB_BLACK    0

#define __rb_parent(pc)     ((PMMADDRESS_NODE)((long)pc & ~3))
#define rb_parent(rb)       (SANITIZE_PARENT_NODE((rb)->u1.Parent)
    )

#define __rb_color(pc)      ((long)(pc) & 1)
#define __rb_is_red(pc)     __rb_color((long)pc)
#define __rb_is_black(pc)   (!__rb_color((long)pc))
#define rb_color(rb)        __rb_color((rb)->u1.Parent)
#define rb_is_red(rb)       __rb_is_red((rb)->u1.Parent)
#define rb_is_black(rb)     __rb_is_black((rb)->u1.Parent)

static void rb_set_black(PMMADDRESS_NODE rb);
static void rb_set_red(PMMADDRESS_NODE rb);
static void rb_set_parent(PMMADDRESS_NODE rb, PMMADDRESS_NODE p);
static void rb_set_parent_color(PMMADDRESS_NODE rb,
    PMMADDRESS_NODE p, int color);
static void __rb_change_child(PMMADDRESS_NODE old,
    PMMADDRESS_NODE _new,
          PMMADDRESS_NODE parent, PMMADDRESS_NODE root);
static void __rb_rotate_set_parents(PMMADDRESS_NODE old,
```

```
    PMMADDRESS_NODE _new,
21              PMMADDRESS_NODE root, int color);
22 static void ______rb_erase_color(PMMADDRESS_NODE parent,
    PMMADDRESS_NODE root);
23 /* Non-inline version for rb_erase_augmented() use */
24 void ___rb_erase_color(PMMADDRESS_NODE parent,PMMADDRESS_NODE root
    );
25 static PMMADDRESS_NODE ___rb_erase_augmented(PMMADDRESS_NODE node,
     PMMADDRESS_NODE root);
```

这一部分函数可以直接移植到 WRK 中。

再来看 WRK。WRK 中 VAD 所使用的 AVL 树的节点与根定义在./base/ntos/inc/ps.h 中，根节点定义如下所示：

```
1  typedef struct _MM_AVL_TABLE {
2      MMADDRESS_NODE  BalancedRoot;
3      ULONG_PTR DepthOfTree: 5;
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;
```

如上代码所示，实际的根节点其实是 BalancedRoot->RightChild，而 BalancedRoot->LeftChild 始终为 NULL。

而 AVL 树的结点定义如下所示：

```
1  typedef struct _MMADDRESS_NODE {
2      union {
3          LONG_PTR Balance : 2;
4          struct _MMADDRESS_NODE *Parent;
5      } u1;
6      struct _MMADDRESS_NODE *LeftChild;
7      struct _MMADDRESS_NODE *RightChild;
8      ULONG_PTR StartingVpn;
9      ULONG_PTR EndingVpn;
10 } MMADDRESS_NODE, *PMMADDRESS_NODE;
```

为了使得移植能够较小的改动原代码，我们采用原始的定义，并将原来结点中的 balance 作为红黑树中的 color 使用。而由于 balance 最开始被初始化为 0，而红黑树中结点一开始被初始化为黑色结点，因此在移植过程中，应当将黑色定义为 0，红色定义为 1，这与 Linux 原码中有所不同。

AVL 树的方法定义在./base/ntos/mm/addrsup.c 中, 我们需要修改的是其中插入结点与删除结点的函数，即 MiInsertNode() 和 MiRemoveNode() 两个函数。

0.3.3 实验过程

我们可以直接将上一节中提到的提到的处理红黑树颜色的一些功能函数包括查找 parent，设置 parent，判断颜色是否为红/黑，设置颜色为红/黑等，直接复制到 addrsup.c 中，注意要将原来 Linux 中的 node,root 等结构替换成为 WRK 中对应的 node,root 等结构。具体的替换过程如下所示：

rb_node * 替换为 PMMADDRESS_NODE;

rb_root * 替换为 PMM_AVL_TABLE;

rb_left 替换为 LeftChild;

rb_right 替换为 RightChild;

root->rb_node 替换为 root->BalancedRoot.RightChild;

而对于红黑树的旋转操作，因为原始 Linux 原码中的红黑树的旋转操作较为复杂，因此在这里自己实现了红黑树的旋转操作，即向左旋转与向右旋转，在我的代码里面定义为 rb_rotate_left() 与 rb_rotate_right()。由于旋转的这部分代码是所有红黑树普遍适用的，因此不在此单独赘述。

而原始的 WRK 函数中，需要将 MiInsertNode() 和 MiRemoveNode() 两个函数做出符合红黑树的修改，其余的函数都不需要做出改变。因此我们将原始 MiInsertNode() 与 MiRemoveNode() 中的代码删除，自行完成红黑树的删除与插入的代码。具体来讲：

对于 MiInsertNode()，除去插入的节点是根节点的情况，与插入的节点父节点是黑色的情况不会破坏红黑树的性质，无需做额外的处理外，我们需要对以下状况进行修复：

1. 当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色（需区分左子与右子，处理是对称的）
2. 当前节点的父节点是红色，叔叔节点是黑色，当前节点是其父节点的右子
3. 当前节点的父节点是红色，叔叔节点是黑色，当前节点是其父节点的左子

这里即可采用常规的红黑树冲突处理的方式进行插入、旋转、调色等操作即可。即对于情况一，将当前节点的父节点和叔叔节点涂黑，祖父结点涂红，把当前结点指向祖父节点，从新的当前节点重新开始算法；对于情况二，当前节点的父节点做为新的当前节点，以新当前节点为支点左旋。；对于情况三，父节点变为黑色，祖父节点变为红色，在祖父节点为支点右旋即可。

具体的代码如下所示：

```
VOID
FASTCALL
MiInsertNode (
    IN PMMADDRESS_NODE NodeToInsert,
    IN PMM_AVL_TABLE Table
    )
{
    PMMADDRESS_NODE NodeOrParent;
    TABLE_SEARCH_RESULT SearchResult;

    SearchResult = MiFindNodeOrParent (Table,
                                       NodeToInsert->StartingVpn,
                                       &NodeOrParent);

    NodeToInsert->LeftChild = NULL;
    NodeToInsert->RightChild = NULL;

    Table->NumberGenericTableElements += 1;

    if (SearchResult == TableEmptyTree) {

        Table->BalancedRoot.RightChild = NodeToInsert;
        rb_set_parent(NodeToInsert,&Table->BalancedRoot)
    }
    else {

        PMMADDRESS_NODE R = NodeToInsert;
        PMMADDRESS_NODE S = NodeOrParent;
        PMMADDRESS_NODE gparent;

        if (SearchResult == TableInsertAsLeft) {
            NodeOrParent->LeftChild = NodeToInsert;
        }
        else {
            NodeOrParent->RightChild = NodeToInsert;
        }

        rb_set_parent(NodeToInsert,NodeOrParent)
        NodeToInsert->u1.Balance = RB_RED ;

```

```
41          //
42          // The above completes the standard binary tree insertion
    , which
43          // happens to correspond to steps A1–A5 of Knuth's "
    balanced tree
44          // search and insertion" algorithm.  Now comes the time
    to adjust
45          // balance factors and possibly do a single or double
    rotation as
46          // in steps A6–A10.
47          //
48          // Set the Balance factor in the root to a convenient
    value
49          // to simplify loop control.
50          //
51
52          while(1)
53          {
54              parent=SANITIZE_PARENT_NODE(NodeToInsert->u1.Parent);
55              if(!(parent && rb_is_red(parent))
56                  break;
57              gparent = rb_parent(parent);
58              if(parent == gparent->LeftChild)
59              {
60                  PMMADDRESS_NODE uncle = gparent->RightChild;
61                  if ( uncle && rb_is_red (uncle))
62                  {
63                      rb_set_black(uncle);
64                      rb_set_black(parent);
65                      rb_set_red(gparent);
66                      NodeToInsert = gparent;
67                  }
68                  else
69                  {
70                      if(parent->RightChild == NodeToInsert)
71                      {
72                          PMMADDRESS_NODE tmp;
73                          rb_rotate_left(parent,Table);
74                          tmp = parent;
75                          parent = NodeToInsert;
76                          NodeToInsert = tmp;
77                      }
78                      rb_set_black(parent);
79                      rb_set_red(gparent);
80                      rb_rotate_right(gparent,Table);
81                  }
82              }
83              else
```

```
84              {
85                  PMMADDRESS_NODE uncle = gparent->LeftChild;
86                  if ( uncle && rb_is_red (uncle))
87                  {
88                      rb_set_black(uncle);
89                      rb_set_black(parent);
90                      rb_set_red(gparent);
91                      NodeToInsert = gparent;
92                  }
93                  else
94                  {
95                      if(parent->LeftChild == NodeToInsert)
96                      {
97                          PMMADDRESS_NODE tmp;
98                          rb_rotate_right(parent,Table);
99                          tmp = parent;
100                         parent = NodeToInsert;
101                         NodeToInsert = tmp;
102                     }
103                     rb_set_black(parent);
104                     rb_set_red(gparent);
105                     rb_rotate_left(gparent,Table);
106                 }
107             }
108         }
109         rb_set_black(Table->BalancedRoot.RightChild);
110     }
111     return;
112 }
```

而对于 MiRemoveNode 除了使用常规的二叉搜索树的删除外，还有在删除了节点之后，为满足红黑树的性质而进行调整。为满足红黑树而进行的调整定义在 rb_erase_color() 函数中。

对于二叉搜索树的删除，按照删除结点孩子的个数可以分为三种：

1. 没有儿子，即为叶结点。直接把父结点的对应儿子指针设为 NULL，删除儿子结点就 OK 了
2. 只有一个儿子。那么把父结点的相应儿子指针指向儿子的独生子，删除儿子结点也 OK 了
3. 有两个儿子。可以选择左儿子中的最大元素或者右儿子中的最小元素放到待删除节点的位置，就可以保证结构的不变。

具体的代码如下所示：

```
1 VOID
```

```
FASTCALL
MiRemoveNode (
    IN PMMADDRESS_NODE NodeToDelete,
    IN PMM_AVL_TABLE Table
    )
{
    PMMADDRESS_NODE child,parent;
    int color;

    if(!NodeToDelete->LeftChild)
        child=NodeToDelete->RightChild;
    else if(!NodeToDelete->RightChild)
        child=NodeToDelete->LeftChild;
    else
    {
        PMMADDRESS_NODE old,left;
        old = NodeToDelete;
        NodeToDelete = NodeToDelete->RightChild;
        while((left=NodeToDelete->LeftChild)!=NULL)
            NodeToDelete = left;
        if(rb_parent(old))
        {
            if(rb_parent(old)->LeftChild == old)
                rb_parent(old)->LeftChild = NodeToDelete;
            else
                rb_parent(old)->RightChild = NodeToDelete;
        }
        else
        {
            (Table->BalancedRoot).RightChild = NodeToDelete;
        }
        child = NodeToDelete->RightChild;
        parent = rb_parent(NodeToDelete);
        color = rb_color(NodeToDelete);

        if (parent == old) {
            parent = NodeToDelete;
        } else {
            if (child)
                rb_set_parent(child, parent);
            parent->LeftChild = child;

            NodeToDelete->RightChild = old->RightChild;
            rb_set_parent(old->RightChild, NodeToDelete);
        }
        NodeToDelete->u1.Parent = old->u1.Parent;
        NodeToDelete->LeftChild = old->LeftChild;
        rb_set_parent(old->LeftChild, NodeToDelete);
```

```
50
51          if (color == rb_black)
52          rb_erase_color(child, parent, Table);
53
54          Table->NumberGenericTableElements-=1;
55
56          return;
57      }
58
59      parent = rb_parent(NodeToDelete);
60      color = rb_color(NodeToDelete);
61
62      if (child)
63          rb_set_parent(child, parent);
64      if (parent)
65      {
66          if (parent->LeftChild == NodeToDelete)
67              parent->LeftChild = child;
68          else
69              parent->RightChild = child;
70      }
71      else
72          Table->BalancedRoot.RightChild = child;
73
74      if (color == rb_black)
75          rb_erase_color(child, parent, Table);
76
77      Table->NumberGenericTableElements-=1;
78
79      return;
80  }
```

对于为满足红黑树的性质,需要对原红黑树进行调整,定义在 rb_erase_color() 函数中，具体的需要注意的情况如下所述:

1. 兄弟结点 S 是红色的
2. 兄弟结点 S 是黑色的，但是 S 的两个儿子都是黑色的。但 N 的父结点 P 是黑色
3. 兄弟结点 S 是黑色的，S 的儿子也都是黑色的，但是 N 的父亲 P 是红色
4. 兄弟结点 S 为黑色，S 的左孩子为红色，S 的右孩子是黑色，而 N 是它父结点的左儿子
5. 兄弟结点 S 为黑色，S 的左孩子为红色，S 的右孩子是黑色，而 N 是它父结点的左儿子

具体修复算法的描述在此不赘述，· 代码如下：

```
void rb_erase_color(PMMADDRESS_NODE node, PMMADDRESS_NODE parent,
     PMM_AVL_TABLE root)
{
    PMMADDRESS_NODE sibling;
    while (1)
    {
        if(!((!node || rb_is_black(node)) && node != root->
    BalancedRoot.RightChild))
            break;

        if (parent->LeftChild == node)
        {
            sibling = parent->RightChild;
            if (rb_is_red(sibling))
            {
                rb_set_black(sibling);
                rb_set_red(parent);
                rb_rotate_left(parent, root);
                sibling = parent->RightChild;
            }
            if ((!sibling->LeftChild || rb_is_black(sibling->
    LeftChild)) &&
                (!sibling->RightChild || rb_is_black(sibling->
    RightChild)))
            {
                rb_set_red(sibling);
                node = parent;
                parent = rb_parent(node);
            }
            else
            {
                if (!sibling->RightChild || rb_is_black(sibling->
    RightChild))
                {
                    rb_set_red(sibling);
                    rb_set_black(sibling->LeftChild);
                    rb_rotate_right(sibling, root);
                    sibling = parent->RightChild;
                }
                sibling->u1.Balance = rb_color(parent);
                rb_set_black(parent);
                rb_set_black(sibling->RightChild);
                rb_rotate_left(parent, root);
                node = root->BalancedRoot.RightChild;
            }
        }
```

```
42          else
43          {
44              sibling = parent->LeftChild;
45              if (rb_is_red(sibling))
46              {
47                  rb_set_black(sibling);
48                  rb_set_red(parent);
49                  rb_rotate_right(parent, root);
50                  sibling = parent->LeftChild;
51              }
52              if ((!sibling->LeftChild || rb_is_black(sibling->
    LeftChild)) &&
53                  (!sibling->RightChild || rb_is_black(sibling->
    RightChild)))
54              {
55                  rb_set_red(sibling);
56                  node = parent;
57                  parent = rb_parent(node);
58              }
59              else
60              {
61                  if (!sibling->LeftChild || rb_is_black(sibling->
    LeftChild))
62                  {
63                      rb_set_black(sibling->RightChild);
64                      rb_set_red(sibling);
65                      rb_rotate_left(sibling, root);
66                      sibling = parent->LeftChild;
67                  }
68                  sibling->u1.Balance=parent->u1.Balance;
69                  rb_set_black(parent);
70                  rb_set_black(sibling->LeftChild);
71                  rb_rotate_right(parent, root);
72                  node = root->BalancedRoot.RightChild;
73              }
74          }
75      }
76      if (node)
77          rb_set_black(node);
78
79      return;
80  }
```

0.4 实验结果

经过测试，修改为红黑树后，整体系统运行稳定，开机未出现蓝屏。系统没有产生崩溃，内存占用率等没有显著变化。总体来讲，红黑树的移植是成功的!

0.5 实验感想

操作系统的本次试验，让我首次接触到了内核调试的概念，也是我第一次在内核原码的层次上进行编程。编程过程中遇到了很多困难，包括习惯了 C++ 之后，对 C 语言的一些特性不是很熟悉；需要讲 linux 中红黑树的定义以及一些操作函数中结点的定义都替换成为在 Windows 内核中被定义好的内嵌节点；对红黑树与 AVL 不熟悉等。经过本次试验后，我对操作系统有了更深的理解。