Exhibit 1

IN THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF TEXAS
TYLER DIVISION

Bedrock Computer Technologies, LLC,

Plaintiff,

v.

SoftLayer Technologies, Inc., et al.,

Defendants.

Civil Action No. 6:09-CV-269-LED

DECLARATION OF ALEXEY KUZNETSOV

I, Alexey Kuznetsov, declare and state that I am over the age of eighteen and am competent to give this declaration. Unless indicated otherwise, I have personal knowledge of the facts contained herein.

1. Beginning in 1995 or earlier, I was a contributor to the Linux kernel. While working as a contributor to the Linux kernel, I wrote software related to the Linux routing cache.

2. In 1995, I wrote software that added a routing cache to the Linux kernel. This software was included in a patch that was made publicly available on the Internet on or around November 17, 1995. This patch included changes to the source code file route.c. One of these changes was the addition of a new function, rt_cache_add(). The patch was documented in a change log, which was also made publicly available on the Internet on or around November 17, 1995. Attached hereto as Exhibit A is a true and correct copy of the change log for this patch as it existed on or about November 17, 1995 (identified by Bates labels RHT-BR00029911 – RHT-BR00029945).

3. The software associated with the patch identified in Exhibit A was included in Linux kernel version 1.3.42. Linux kernel version 1.3.42 was publicly available on the Internet

on or around November 16, 1995. Attached hereto as Exhibit B is a true and correct copy of route.c in Linux kernel version 1.3.42 (identified by Bates labels KS-DEF-0000686 - KS-DEF-0000714).

4. The function rt_cache_add() in route.c was included in each version of the Linux kernel between 1.3.42 and 1.3.100; as well as in each version of the Linux kernel between 2.0.1 and 2.0.14.

5. Linux kernel version 1.3.51 was publicly available on the Internet on or around December 27, 1995. Attached hereto as Exhibit C is a true and correct copy of route.c in Linux kernel version 1.3.51 (identified by Bates labels DEF00007865 – DEF00007899).

6. Linux kernel version 1.3.52 was publicly available on the Internet on or around December 29, 1995. Attached hereto as Exhibit D is a true and correct copy of route.c in Linux kernel version 1.3.52 (identified by Bates labels DEF00001013 – DEF00001043).

7. Linux kernel version 2.0.1 was publicly available on the Internet on or around July 3, 1996. Attached hereto as Exhibit E is a true and correct copy of route.c in Linux kernel version 2.0.1 (identified by Bates labels DEF00008567 – DEF00008601). Attached hereto as Exhibit F is a true and correct copy of route.h in Linux kernel version 2.0.1 (identified by Bates labels DEF00008602 – DEF00008605).

I declare under penalty of perjury that the foregoing is true and correct.

Executed on this the 15 day of December, 2010.

Alexey Kuznetsov

```
From a06606bdd748dfeba6cdc1100360d3035663e2d5 Mon Sep 17 00:00:00 2001
From: davem <davem>
Date: Fri, 17 Nov 1995 01:02:00 +0000
Subject: [PATCH 003/103] Merge to 1.3.42

---
 net/ipv4/route.c | 1727 ++++++++++++++++++++++++++++++++++++++---------------
--
 1 files changed, 1380 insertions(+), 347 deletions(-)

diff --git a/net/ipv4/route.c b/net/ipv4/route.c
index 6483db0..d14fead 100644
--- a/net/ipv4/route.c
+++ b/net/ipv4/route.c
@@ -35,6 +35,8 @@
  *         Alan Cox    :     Aligned routing errors more closely with BSD
  *                           our system is still very different.
  *         Alan Cox    :     Faster /proc handling
+ *   Alexey Kuznetsov  :     Massive rework to support tree based routing,
+ *                           routing caches and better behaviour.
  *
  *         This program is free software; you can redistribute it and/or
  *         modify it under the terms of the GNU General Public License
@@ -42,8 +44,10 @@
  *         2 of the License, or (at your option) any later version.
  */

+#include <linux/config.h>
 #include <asm/segment.h>
 #include <asm/system.h>
+#include <asm/bitops.h>
 #include <linux/types.h>
 #include <linux/kernel.h>
 #include <linux/sched.h>
@@ -65,102 +69,246 @@
 #include <net/netlink.h>

 /*
- *   The routing table list
+ * Forwarding Information Base definitions.
  */

-static struct rtable *rt_base = NULL;
-unsigned long rt_stamp = 1;        /* Routing table version stamp for caches
( 0 is 'unset' ) */
+struct fib_node
+{
+     struct fib_node         *fib_next;
+     __u32              fib_dst;
+     unsigned long           fib_use;
+     struct fib_info         *fib_info;
+     short              fib_metric;
+     unsigned char           fib_tos;
+};

 /*
- *   Pointer to the loopback route
```

```
+ * This structure contains data shared by many of routes.
+ */
+
+struct fib_info
+{
+     struct fib_info         *fib_next;
+     struct fib_info         *fib_prev;
+     __u32              fib_gateway;
+     struct device           *fib_dev;
+     int                fib_refcnt;
+     unsigned long           fib_window;
+     unsigned short          fib_flags;
+     unsigned short          fib_mtu;
+     unsigned short          fib_irtt;
+};
+
+struct fib_zone
+{
+     struct fib_zone    *fz_next;
+     struct fib_node    **fz_hash_table;
+     struct fib_node    *fz_list;
+     int          fz_nent;
+     int          fz_logmask;
+     __u32        fz_mask;
+};
+
+static struct fib_zone       *fib_zones[33];
+static struct fib_zone       *fib_zone_list;
+static struct fib_node       *fib_loopback = NULL;
+static struct fib_info       *fib_info_list;
+
+/*
+ * Backlogging.
  */
-
-static struct rtable *rt_loopback = NULL;
+
+#define RT_BH_REDIRECT       0
+#define RT_BH_GARBAGE_COLLECT      1
+#define RT_BH_FREE           2
+
+struct rt_req
+{
+     struct rt_req * rtr_next;
+     struct device *dev;
+     __u32 dst;
+     __u32 gw;
+     unsigned char tos;
+};
+
+int               ip_rt_lock;
+unsigned          ip_rt_bh_mask;
+static struct rt_req   *rt_backlog;

 /*
- *   Remove a routing table entry.
+ * Route cache.
```

```
  */

-static int rt_del(__u32 dst, __u32 mask,
-           char *devname, __u32 gtw, short rt_flags, short metric)
+struct rtable          *ip_rt_hash_table[RT_HASH_DIVISOR];
+static int       rt_cache_size;
+static struct rtable   *rt_free_queue;
+struct wait_queue      *rt_wait;
+
+static void rt_kick_backlog(void);
+static void rt_cache_add(unsigned hash, struct rtable * rth);
+static void rt_cache_flush(void);
+static void rt_garbage_collect_1(void);
+
+/*
+ * Evaluate mask length.
+ */
+
+static __inline__ int rt_logmask(__u32 mask)
 {
-     struct rtable *r, **rp;
-     unsigned long flags;
-     int found=0;
+     if (!(mask = ntohl(mask)))
+           return 32;
+     return ffz(~mask);
+}

-     rp = &rt_base;
-
-     /*
-      *    This must be done with interrupts off because we could take
-      *    an ICMP_REDIRECT.
-      */
-
-     save_flags(flags);
-     cli();
-     while((r = *rp) != NULL)
+/*
+ * Create mask from length.
+ */
+
+static __inline__ __u32 rt_mask(int logmask)
+{
+     if (logmask >= 32)
+           return 0;
+     return htonl(~((1<<logmask)-1));
+}
+
+static __inline__ unsigned fz_hash_code(__u32 dst, int logmask)
+{
+     return ip_rt_hash_code(ntohl(dst)>>logmask);
+}
+
+/*
+ * Free FIB node.
+ */
```

```
+
+static void fib_free_node(struct fib_node * f)
+{
+     struct fib_info * fi = f->fib_info;
+     if (!--fi->fib_refcnt)
      {
-           /*
-            *    Make sure the destination and netmask match.
-            *    metric, gateway and device are also checked
-            *    if they were specified.
-            */
-           if (r->rt_dst != dst ||
-               (mask && r->rt_mask != mask) ||
-               (gtw && r->rt_gateway != gtw) ||
-               (metric >= 0 && r->rt_metric != metric) ||
-               (devname && strcmp((r->rt_dev)->name,devname) != 0) )
+#if RT_CACHE_DEBUG >= 2
+           printk("fib_free_node: fi %08x/%s is free\n", fi->fib_gateway,
fi->fib_dev->name);
+#endif
+           if (fi->fib_next)
+                 fi->fib_next->fib_prev = fi->fib_prev;
+           if (fi->fib_prev)
+                 fi->fib_prev->fib_next = fi->fib_next;
+           if (fi == fib_info_list)
+                 fib_info_list = fi->fib_next;
+     }
+     kfree_s(f, sizeof(struct fib_node));
+}
+
+/*
+ * Find gateway route by address.
+ */
+
+static struct fib_node * fib_lookup_gateway(__u32 dst)
+{
+     struct fib_zone * fz;
+     struct fib_node * f;
+
+     for (fz = fib_zone_list; fz; fz = fz->fz_next)
+     {
+           if (fz->fz_hash_table)
+                 f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
+           else
+                 f = fz->fz_list;
+
+           for ( ; f; f = f->fib_next)
            {
-                 rp = &r->rt_next;
-                 continue;
+                 if ((dst ^ f->fib_dst) & fz->fz_mask)
+                       continue;
+                 if (f->fib_info->fib_flags & RTF_GATEWAY)
+                       return NULL;
+                 return f;
            }
-           *rp = r->rt_next;
```

```
-
-           /*
-            *    If we delete the loopback route update its pointer.
-            */
-
-           if (rt_loopback == r)
-                 rt_loopback = NULL;
-           ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, rt_flags, metric,
r->rt_dev->name);
-           kfree_s(r, sizeof(struct rtable));
-           found=1;
-     }
-     rt_stamp++;       /* New table revision */
-
-     restore_flags(flags);
-
-     if(found)
-           return 0;
-     return -ESRCH;
+     }
+     return NULL;
 }

+/*
+ * Find local route by address.
+ * FIXME: I use "longest match" principle. If destination
+ *     has some non-local route, I'll not search shorter matches.
+ *     It's possible, I'm wrong, but I wanted to prevent following
+ *     situation:
+ *   route add 193.233.7.128 netmask 255.255.255.192 gw xxxxxx
+ *   route add 193.233.7.0   netmask 255.255.255.0 eth1
+ *     (Two ethernets connected by serial line, one is small and other is
large)
+ *     Host 193.233.7.129 is locally unreachable,
+ *     but old (<=1.3.37) code will send packets destined for it to eth1.
+ *
+ */
+
+static struct fib_node * fib_lookup_local(__u32 dst)
+{
+     struct fib_zone * fz;
+     struct fib_node * f;
+
+     for (fz = fib_zone_list; fz; fz = fz->fz_next)
+     {
+           int longest_match_found = 0;
+
+           if (fz->fz_hash_table)
+                 f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
+           else
+                 f = fz->fz_list;
+
+           for ( ; f; f = f->fib_next)
+           {
+                 if ((dst ^ f->fib_dst) & fz->fz_mask)
+                       continue;
+                 if (!(f->fib_info->fib_flags & RTF_GATEWAY))
```

```
+                       return f;
+                 longest_match_found = 1;
+           }
+           if (longest_match_found)
+                 return NULL;
+     }
+     return NULL;
+}

 /*
- *   Remove all routing table entries for a device. This is called when
- *   a device is downed.
+ * Main lookup routine.
+ *   IMPORTANT NOTE: this algorithm has small difference from <=1.3.37
visible
+ *   by user. It doesn't route non-CIDR broadcasts by default.
+ *
+ *   F.e.
+ *         ifconfig eth0 193.233.7.65 netmask 255.255.255.192 broadcast
193.233.7.255
+ *   is valid, but if you really are not able (not allowed, do not want) to
+ *   use CIDR compliant broadcast 193.233.7.127, you should add host route:
+ *         route add -host 193.233.7.255 eth0
  */
-
-void ip_rt_flush(struct device *dev)
+
+static struct fib_node * fib_lookup(__u32 dst)
 {
-     struct rtable *r;
-     struct rtable **rp;
-     unsigned long flags;
+     struct fib_zone * fz;
+     struct fib_node * f;

-     rp = &rt_base;
-     save_flags(flags);
-     cli();
-     while ((r = *rp) != NULL) {
-           if (r->rt_dev != dev) {
-                 rp = &r->rt_next;
-                 continue;
+     for (fz = fib_zone_list; fz; fz = fz->fz_next)
+     {
+           if (fz->fz_hash_table)
+                 f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
+           else
+                 f = fz->fz_list;
+
+           for ( ; f; f = f->fib_next)
+           {
+                 if ((dst ^ f->fib_dst) & fz->fz_mask)
+                       continue;
+                 return f;
            }
-           *rp = r->rt_next;
-           if (rt_loopback == r)
```

```
-                 rt_loopback = NULL;
-           kfree_s(r, sizeof(struct rtable));
-     }
-     rt_stamp++;       /* New table revision */
-     restore_flags(flags);
+     }
+     return NULL;
+}
+
+static __inline__ struct device * get_gw_dev(__u32 gw)
+{
+     struct fib_node * f;
+     f = fib_lookup_gateway(gw);
+     if (f)
+           return f->fib_info->fib_dev;
+     return NULL;
 }

 /*
@@ -200,60 +348,181 @@ static __u32 guess_mask(__u32 dst, struct device *
dev)


 /*
- *   Find the route entry through which our gateway will be reached
+ *   Check if a mask is acceptable.
  */

-static inline struct device * get_gw_dev(__u32 gw)
+static inline int bad_mask(__u32 mask, __u32 addr)
 {
-     struct rtable * rt;
+     if (addr & (mask = ~mask))
+           return 1;
+     mask = ntohl(mask);
+     if (mask & (mask+1))
+           return 1;
+     return 0;
+}
+
+
+static int fib_del_list(struct fib_node **fp, __u32 dst,
+           struct device * dev, __u32 gtw, short flags, short metric, __u32
mask)
+{
+     struct fib_node *f;
+     int found=0;

-     for (rt = rt_base ; ; rt = rt->rt_next)
+     while((f = *fp) != NULL)
      {
-           if (!rt)
-                 return NULL;
-           if ((gw ^ rt->rt_dst) & rt->rt_mask)
+           struct fib_info * fi = f->fib_info;
+
+           /*
```

```
+             *    Make sure the destination and netmask match.
+             *    metric, gateway and device are also checked
+             *    if they were specified.
+             */
+            if (f->fib_dst != dst ||
+                (gtw && fi->fib_gateway != gtw) ||
+                (metric >= 0 && f->fib_metric != metric) ||
+                (dev && fi->fib_dev != dev) )
+            {
+                  fp = &f->fib_next;
                   continue;
-            /*
-             *    Gateways behind gateways are a no-no
+            }
+            cli();
+            *fp = f->fib_next;
+            if (fib_loopback == f)
+                  fib_loopback = NULL;
+            sti();
+            ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric, fi-
>fib_dev->name);
+            fib_free_node(f);
+            found++;
+      }
+      return found;
+}
+
+static __inline__ int fib_del_1(__u32 dst, __u32 mask,
+            struct device * dev, __u32 gtw, short flags, short metric)
+{
+      struct fib_node **fp;
+      struct fib_zone *fz;
+      int found=0;
+
+      if (!mask)
+      {
+            for (fz=fib_zone_list; fz; fz = fz->fz_next)
+            {
+                  int tmp;
+                  if (fz->fz_hash_table)
+                        fp = &fz->fz_hash_table[fz_hash_code(dst, fz-
>fz_logmask)];
+                  else
+                        fp = &fz->fz_list;
+
+                  tmp = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
+                  fz->fz_nent -= tmp;
+                  found += tmp;
+            }
+      }
+      else
+      {
+            if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
+            {
+                  if (fz->fz_hash_table)
+                        fp = &fz->fz_hash_table[fz_hash_code(dst, fz-
>fz_logmask)];
```

```
+                 else
+                       fp = &fz->fz_list;
+
+                 found = fib_del_list(fp, dst, dev, gtw, flags, metric,
mask);
+                 fz->fz_nent -= found;
+           }
+     }
+
+     if (found)
+     {
+           rt_cache_flush();
+           return 0;
+     }
+     return -ESRCH;
+}
+
+
+static struct fib_info * fib_create_info(__u32 gw, struct device * dev,
+                              unsigned short flags, unsigned short mss,
+                              unsigned long window, unsigned short irtt)
+{
+     struct fib_info * fi;
+
+     if (!(flags & RTF_MSS))
+     {
+           mss = dev->mtu;
+#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
+           /*
+            *    If MTU was not specified, use default.
+            *    If you want to increase MTU for some net (local subnet)
+            *    use "route add .... mss xxx".
+            *
+            *    The MTU isnt currently always used and computed as it
+            *    should be as far as I can tell. [Still verifying this is
right]
             */
-
-           if (rt->rt_flags & RTF_GATEWAY)
-                 return NULL;
-           return rt->rt_dev;
+           if ((flags & RTF_GATEWAY) && mss > 576)
+                 mss = 576;
+#endif
      }
+     if (!(flags & RTF_WINDOW))
+           window = 0;
+     if (!(flags & RTF_IRTT))
+           irtt = 0;
+
+     for (fi=fib_info_list; fi; fi = fi->fib_next)
+     {
+           if (fi->fib_gateway != gw ||
+               fi->fib_dev != dev  ||
+               fi->fib_flags != flags ||
+               fi->fib_mtu != mss ||
+               fi->fib_window != window ||
```

```
+               fi->fib_irtt != irtt)
+                 continue;
+           fi->fib_refcnt++;
+#if RT_CACHE_DEBUG >= 2
+           printk("fib_create_info: fi %08x/%s is duplicate\n", fi-
>fib_gateway, fi->fib_dev->name);
+#endif
+           return fi;
+     }
+     fi = (struct fib_info*)kmalloc(sizeof(struct fib_info), GFP_KERNEL);
+     if (!fi)
+           return NULL;
+     memset(fi, 0, sizeof(struct fib_info));
+     fi->fib_flags = flags;
+     fi->fib_dev = dev;
+     fi->fib_gateway = gw;
+     fi->fib_mtu = mss;
+     fi->fib_window = window;
+     fi->fib_refcnt++;
+     fi->fib_next = fib_info_list;
+     fi->fib_prev = NULL;
+     if (fib_info_list)
+           fib_info_list->fib_prev = fi;
+     fib_info_list = fi;
+#if RT_CACHE_DEBUG >= 2
+     printk("fib_create_info: fi %08x/%s is created\n", fi->fib_gateway, fi-
>fib_dev->name);
+#endif
+     return fi;
 }

-/*
- *   Rewrote rt_add(), as the old one was weird - Linus
- *
- *   This routine is used to update the IP routing table, either
- *   from the kernel (ICMP_REDIRECT) or via an ioctl call issued
- *   by the superuser.
- */
-
-void ip_rt_add(short flags, __u32 dst, __u32 mask,
-     __u32 gw, struct device *dev, unsigned short mtu,
+
+static __inline__ void fib_add_1(short flags, __u32 dst, __u32 mask,
+     __u32 gw, struct device *dev, unsigned short mss,
      unsigned long window, unsigned short irtt, short metric)
 {
-     struct rtable *r, *rt;
-     struct rtable **rp;
-     unsigned long cpuflags;
-     int duplicate = 0;
+     struct fib_node *f, *f1;
+     struct fib_node **fp;
+     struct fib_node **dup_fp = NULL;
+     struct fib_zone * fz;
+     struct fib_info * fi;
+     int logmask;
```

```
-     /*
-      *    A host is a unique machine and has no network bits.
-      */
-
      if (flags & RTF_HOST)
-     {
            mask = 0xffffffff;
-     }
-
      /*
-      *    Calculate the network mask
+      * If mask is not specified, try to guess it.
       */
-
-     else if (!mask)
+     else if (!mask)
      {
            if (!((dst ^ dev->pa_addr) & dev->pa_mask))
            {
@@ -261,7 +530,7 @@ void ip_rt_add(short flags, __u32 dst, __u32 mask,
                  flags &= ~RTF_GATEWAY;
                  if (flags & RTF_DYNAMIC)
                  {
-                       /*printk("Dynamic route to my own net rejected\n");*/
+                       printk("Dynamic route to my own net rejected\n");
                        return;
                  }
            }
@@ -295,132 +564,1027 @@ void ip_rt_add(short flags, __u32 dst, __u32 mask,
       *    Allocate an entry and fill it in.
       */

-     rt = (struct rtable *) kmalloc(sizeof(struct rtable), GFP_ATOMIC);
-     if (rt == NULL)
+     f = (struct fib_node *) kmalloc(sizeof(struct fib_node), GFP_KERNEL);
+     if (f == NULL)
            return;
+
+     memset(f, 0, sizeof(struct fib_node));
+     f->fib_dst = dst;
+     f->fib_metric = metric;
+     f->fib_tos    = 0;
+
+     if  ((fi = fib_create_info(gw, dev, flags, mss, window, irtt)) == NULL)
      {
+           kfree_s(f, sizeof(struct fib_node));
            return;
      }
-     memset(rt, 0, sizeof(struct rtable));
-     rt->rt_flags = flags | RTF_UP;
-     rt->rt_dst = dst;
-     rt->rt_dev = dev;
-     rt->rt_gateway = gw;
-     rt->rt_mask = mask;
-     rt->rt_mss = dev->mtu - HEADER_SIZE;
-     rt->rt_metric = metric;
-     rt->rt_window = 0;      /* Default is no clamping */
```

```
+     f->fib_info = fi;

-     /* Are the MSS/Window valid ? */
+     logmask = rt_logmask(mask);
+     fz = fib_zones[logmask];

-     if(rt->rt_flags & RTF_MSS)
-           rt->rt_mss = mtu;
-
-     if(rt->rt_flags & RTF_WINDOW)
-           rt->rt_window = window;
-     if(rt->rt_flags & RTF_IRTT)
-           rt->rt_irtt = irtt;
+
+     if (!fz)
+     {
+           int i;
+           fz = kmalloc(sizeof(struct fib_zone), GFP_KERNEL);
+           if (!fz)
+           {
+                 fib_free_node(f);
+                 return;
+           }
+           memset(fz, 0, sizeof(struct fib_zone));
+           fz->fz_logmask = logmask;
+           fz->fz_mask = mask;
+           for (i=logmask-1; i>=0; i--)
+                 if (fib_zones[i])
+                       break;
+           cli();
+           if (i<0)
+           {
+                 fz->fz_next = fib_zone_list;
+                 fib_zone_list = fz;
+           }
+           else
+           {
+                 fz->fz_next = fib_zones[i]->fz_next;
+                 fib_zones[i]->fz_next = fz;
+           }
+           fib_zones[logmask] = fz;
+           sti();
+     }

      /*
-      *    What we have to do is loop though this until we have
-      *    found the first address which has a higher generality than
-      *    the one in rt.  Then we can put rt in right before it.
-      *    The interrupts must be off for this process.
+      * If zone overgrows RTZ_HASHING_LIMIT, create hash table.
       */

-     save_flags(cpuflags);
-     cli();
+     if (fz->fz_nent >= RTZ_HASHING_LIMIT && !fz->fz_hash_table &&
logmask<32)
+     {
```

```
+           struct fib_node ** ht;
+#if RT_CACHE_DEBUG
+           printk("fib_add_1: hashing for zone %d started\n", logmask);
+#endif
+           ht = kmalloc(RTZ_HASH_DIVISOR*sizeof(struct rtable*),
GFP_KERNEL);
+
+           if (ht)
+           {
+                 memset(ht, 0, RTZ_HASH_DIVISOR*sizeof(struct fib_node*));
+                 cli();
+                 f1 = fz->fz_list;
+                 while (f1)
+                 {
+                       struct fib_node * next;
+                       unsigned hash = fz_hash_code(f1->fib_dst, logmask);
+                       next = f1->fib_next;
+                       f1->fib_next = ht[hash];
+                       ht[hash] = f1;
+                       f1 = next;
+                 }
+                 fz->fz_list = NULL;
+                 fz->fz_hash_table = ht;
+                 sti();
+           }
+     }
+
+     if (fz->fz_hash_table)
+           fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
+     else
+           fp = &fz->fz_list;

      /*
-      *    Remove old route if we are getting a duplicate.
+      * Scan list to find the first route with the same destination
       */
-
-     rp = &rt_base;
-     while ((r = *rp) != NULL)
+     while ((f1 = *fp) != NULL)
      {
-           if (r->rt_dst != dst ||
-               r->rt_mask != mask)
-           {
-                 rp = &r->rt_next;
-                 continue;
-           }
-           if (r->rt_metric != metric && r->rt_gateway != gw)
-           {
-                 duplicate = 1;
-                 rp = &r->rt_next;
-                 continue;
-           }
-           *rp = r->rt_next;
-           if (rt_loopback == r)
-                 rt_loopback = NULL;
```

```
-           ip_netlink_msg(RTMSG_DELROUTE, dst,gw, mask, flags, metric, rt-
>rt_dev->name);
-           kfree_s(r, sizeof(struct rtable));
+           if (f1->fib_dst == dst)
+                 break;
+           fp = &f1->fib_next;
      }
-
+
      /*
-      *    Add the new route
+      * Find route with the same destination and less (or equal) metric.
       */
-
-     rp = &rt_base;
-     while ((r = *rp) != NULL) {
-           /*
-            * When adding a duplicate route, add it before
-            * the route with a higher metric.
-            */
-           if (duplicate &&
-               r->rt_dst == dst &&
-               r->rt_mask == mask &&
-               r->rt_metric > metric)
+     while ((f1 = *fp) != NULL && f1->fib_dst == dst)
+     {
+           if (f1->fib_metric >= metric)
                  break;
-           else
            /*
-            * Otherwise, just add it before the
-            * route with a higher generality.
+            *    Record route with the same destination and gateway,
+            *    but less metric. We'll delete it
+            *    after instantiation of new route.
             */
-                 if ((r->rt_mask & mask) != mask)
-                       break;
-           rp = &r->rt_next;
+           if (f1->fib_info->fib_gateway == gw)
+                 dup_fp = fp;
+           fp = &f1->fib_next;
+     }
+
+     /*
+      * Is it already present?
+      */
+
+     if (f1 && f1->fib_metric == metric && f1->fib_info == fi)
+     {
+           fib_free_node(f);
+           return;
      }
-     rt->rt_next = r;
-     *rp = rt;

      /*
```

```
-       *    Update the loopback route
+       * Insert new entry to the list.
        */
-
-      if ((rt->rt_dev->flags & IFF_LOOPBACK) && !rt_loopback)
-            rt_loopback = rt;

-      rt_stamp++;       /* New table revision */
-
+      cli();
+      f->fib_next = f1;
+      *fp = f;
+      if (!fib_loopback && (fi->fib_dev->flags & IFF_LOOPBACK))
+            fib_loopback = f;
+      sti();
+      fz->fz_nent++;
+      ip_netlink_msg(RTMSG_NEWROUTE, dst, gw, mask, flags, metric, fi-
>fib_dev->name);
+
       /*
-       *    Restore the interrupts and return
+       *    Delete route with the same destination and gateway.
+       *    Note that we should have at most one such route.
        */
+      if (dup_fp)
+            fp = dup_fp;
+      else
+            fp = &f->fib_next;

-      restore_flags(cpuflags);
-      ip_netlink_msg(RTMSG_NEWROUTE, dst,gw, mask, flags, metric, rt->rt_dev-
>name);
+      while ((f1 = *fp) != NULL && f1->fib_dst == dst)
+      {
+            if (f1->fib_info->fib_gateway == gw)
+            {
+                  cli();
+                  *fp = f1->fib_next;
+                  if (fib_loopback == f1)
+                        fib_loopback = NULL;
+                  sti();
+                  ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags,
metric, f1->fib_info->fib_dev->name);
+                  fib_free_node(f1);
+                  fz->fz_nent--;
+                  break;
+            }
+            fp = &f1->fib_next;
+      }
+      rt_cache_flush();
       return;
 }

+static int rt_flush_list(struct fib_node ** fp, struct device *dev)
+{
+      int found = 0;
+      struct fib_node *f;
```

```
-/*
- *   Check if a mask is acceptable.
+     while ((f = *fp) != NULL) {
+           if (f->fib_info->fib_dev != dev) {
+                 fp = &f->fib_next;
+                 continue;
+           }
+           cli();
+           *fp = f->fib_next;
+           if (fib_loopback == f)
+                 fib_loopback = NULL;
+           sti();
+           fib_free_node(f);
+           found++;
+     }
+     return found;
+}
+
+static __inline__ void fib_flush_1(struct device *dev)
+{
+     struct fib_zone *fz;
+     int found = 0;
+
+     for (fz = fib_zone_list; fz; fz = fz->fz_next)
+     {
+           if (fz->fz_hash_table)
+           {
+                 int i;
+                 int tmp = 0;
+                 for (i=0; i<RTZ_HASH_DIVISOR; i++)
+                       tmp += rt_flush_list(&fz->fz_hash_table[i], dev);
+                 fz->fz_nent -= tmp;
+                 found += tmp;
+           }
+           else
+           {
+                 int tmp;
+                 tmp = rt_flush_list(&fz->fz_list, dev);
+                 fz->fz_nent -= tmp;
+                 found += tmp;
+           }
+     }
+
+     if (found)
+           rt_cache_flush();
+}
+
+
+/*
+ *   Called from the PROCfs module. This outputs /proc/net/route.
+ *
+ *   We preserve the old format but pad the buffers out. This means that
+ *   we can spin over the other entries as we read them. Remember the
+ *   gated BGP4 code could need to read 60,000+ routes on occasion (thats
+ *   about 7Mb of data). To do that ok we will need to also cache the
+ *   last route we got to (reads will generally be following on from
```

```
+ *   one another without gaps).
  */

-static inline int bad_mask(__u32 mask, __u32 addr)
+int rt_get_info(char *buffer, char **start, off_t offset, int length, int
dummy)
 {
-     if (addr & (mask = ~mask))
-           return 1;
-     mask = ntohl(mask);
-     if (mask & (mask+1))
-           return 1;
-     return 0;
+     struct fib_zone *fz;
+     struct fib_node *f;
+     int len=0;
+     off_t pos=0;
+     char temp[129];
+     int i;
+
+     pos = 128;
+
+     if (offset<128)
+     {
+           sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\tIRTT");
+           len = 128;
+     }
+
+     while  (ip_rt_lock)
+           sleep_on(&rt_wait);
+     ip_rt_fast_lock();
+
+     for (fz=fib_zone_list; fz; fz = fz->fz_next)
+     {
+           int maxslot;
+           struct fib_node ** fp;
+
+           if (fz->fz_nent == 0)
+                 continue;
+
+           if (pos + 128*fz->fz_nent <= offset)
+           {
+                 pos += 128*fz->fz_nent;
+                 len = 0;
+                 continue;
+           }
+
+           if (fz->fz_hash_table)
+           {
+                 maxslot = RTZ_HASH_DIVISOR;
+                 fp    = fz->fz_hash_table;
+           }
+           else
+           {
+                 maxslot     = 1;
+                 fp    = &fz->fz_list;
```

```
+           }
+
+           for (i=0; i < maxslot; i++, fp++)
+           {
+
+                 for (f = *fp; f; f = f->fib_next)
+                 {
+                       struct fib_info * fi;
+                       /*
+                        *    Spin through entries until we are ready
+                        */
+                       pos += 128;
+
+                       if (pos <= offset)
+                       {
+                             len=0;
+                             continue;
+                       }
+
+                       fi = f->fib_info;
+                       sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%d\t%lu\t%d\t%08lX\t%d\t%lu\t%u",
+                             fi->fib_dev->name, (unsigned long)f->fib_dst,
(unsigned long)fi->fib_gateway,
+                             fi->fib_flags, 0, f->fib_use, f->fib_metric,
+                             (unsigned long)fz->fz_mask, (int)fi->fib_mtu,
fi->fib_window, (int)fi->fib_irtt);
+                       sprintf(buffer+len,"%-127s\n",temp);
+
+                       len += 128;
+                       if (pos >= offset+length)
+                             goto done;
+                 }
+           }
+        }
+
+done:
+     ip_rt_unlock();
+     wake_up(&rt_wait);
+
+     *start = buffer+len-(pos-offset);
+     len = pos - offset;
+     if (len>length)
+           len = length;
+     return len;
+}
+
+int rt_cache_get_info(char *buffer, char **start, off_t offset, int length,
int dummy)
+{
+     int len=0;
+     off_t pos=0;
+     char temp[129];
+     struct rtable *r;
+     int i;
+
+     pos = 128;
```

```
+
+     if (offset<128)
+     {
+           sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tSource\t\tMTU\tWindow\tIRTT\tHH\tARP\n");
+           len = 128;
+     }
+
+
+     while  (ip_rt_lock)
+           sleep_on(&rt_wait);
+     ip_rt_fast_lock();
+
+     for (i = 0; i<RT_HASH_DIVISOR; i++)
+     {
+           for (r = ip_rt_hash_table[i]; r; r = r->rt_next)
+           {
+                 /*
+                  *    Spin through entries until we are ready
+                  */
+                 pos += 128;
+
+                 if (pos <= offset)
+                 {
+                       len = 0;
+                       continue;
+                 }
+
+                 sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%ld\t%lu\t%d\t%08lX\t%d\t%lu\t%u\t%ld\t%1d",
+                       r->rt_dev->name, (unsigned long)r->rt_dst, (unsigned
long)r->rt_gateway,
+                       r->rt_flags, r->rt_refcnt, r->rt_use, 0,
+                       (unsigned long)r->rt_src, (int)r->rt_mtu, r-
>rt_window, (int)r->rt_irtt, r->rt_hh ? r->rt_hh->hh_refcnt : -1, r->rt_hh ?
r->rt_hh->hh_uptodate : 0);
+                 sprintf(buffer+len,"%-127s\n",temp);
+                 len += 128;
+                 if (pos >= offset+length)
+                       goto done;
+           }
+        }
+
+done:
+     ip_rt_unlock();
+     wake_up(&rt_wait);
+
+     *start = buffer+len-(pos-offset);
+     len = pos-offset;
+     if (len>length)
+           len = length;
+     return len;
+}
+
+
+static void rt_free(struct rtable * rt)
+{
```

```
+     unsigned long flags;
+
+     save_flags(flags);
+     cli();
+     if (!rt->rt_refcnt)
+     {
+           struct hh_cache * hh = rt->rt_hh;
+           rt->rt_hh = NULL;
+           if (hh && !--hh->hh_refcnt)
+           {
+                 restore_flags(flags);
+                 kfree_s(hh, sizeof(struct hh_cache));
+           }
+           restore_flags(flags);
+           kfree_s(rt, sizeof(struct rt_table));
+           return;
+     }
+     rt->rt_next = rt_free_queue;
+     rt->rt_flags &= ~RTF_UP;
+     rt_free_queue = rt;
+     ip_rt_bh_mask |= RT_BH_FREE;
+#if RT_CACHE_DEBUG >= 2
+     printk("rt_free: %08x\n", rt->rt_dst);
+#endif
+     restore_flags(flags);
+}
+
+/*
+ * RT "bottom half" handlers. Called with masked inetrrupts.
+ */
+
+static __inline__ void rt_kick_free_queue(void)
+{
+     struct rtable *rt, **rtp;
+
+     rtp = &rt_free_queue;
+
+     while ((rt = *rtp) != NULL)
+     {
+           if  (!rt->rt_refcnt)
+           {
+                 struct hh_cache * hh = rt->rt_hh;
+#if RT_CACHE_DEBUG >= 2
+                 __u32 daddr = rt->rt_dst;
+#endif
+                 *rtp = rt->rt_next;
+                 rt->rt_hh = NULL;
+                 if (hh && !--hh->hh_refcnt)
+                 {
+                       sti();
+                       kfree_s(hh, sizeof(struct hh_cache));
+                 }
+                 sti();
+                 kfree_s(rt, sizeof(struct rt_table));
+#if RT_CACHE_DEBUG >= 2
+                 printk("rt_kick_free_queue: %08x is free\n", daddr);
+#endif
```

```
+                 cli();
+                 continue;
+           }
+           rtp = &rt->rt_next;
+     }
+}
+
+void ip_rt_run_bh() {
+     unsigned long flags;
+     save_flags(flags);
+     cli();
+     if (ip_rt_bh_mask && !ip_rt_lock)
+     {
+           if (ip_rt_bh_mask & RT_BH_REDIRECT)
+                 rt_kick_backlog();
+
+           if (ip_rt_bh_mask & RT_BH_GARBAGE_COLLECT)
+           {
+                 ip_rt_fast_lock();
+                 ip_rt_bh_mask &= ~RT_BH_GARBAGE_COLLECT;
+                 sti();
+                 rt_garbage_collect_1();
+                 cli();
+                 ip_rt_fast_unlock();
+           }
+
+           if (ip_rt_bh_mask & RT_BH_FREE)
+                 rt_kick_free_queue();
+     }
+     restore_flags(flags);
+}
+
+
+void ip_rt_check_expire()
+{
+     ip_rt_fast_lock();
+     if (ip_rt_lock == 1)
+     {
+           int i;
+           struct rtable *rth, **rthp;
+           unsigned long flags;
+           unsigned long now = jiffies;
+
+           save_flags(flags);
+           for (i=0; i<RT_HASH_DIVISOR; i++)
+           {
+                 rthp = &ip_rt_hash_table[i];
+
+                 while ((rth = *rthp) != NULL)
+                 {
+                       struct rtable * rth_next = rth->rt_next;
+
+                       /*
+                        * Cleanup aged off entries.
+                        */
+
+                       cli();
```

```
+                        if (!rth->rt_refcnt && rth->rt_lastuse +
RT_CACHE_TIMEOUT < now)
+                        {
+                              *rthp = rth_next;
+                              sti();
+                              rt_cache_size--;
+#if RT_CACHE_DEBUG >= 2
+                              printk("rt_check_expire clean %02x@%08x\n", i,
rth->rt_dst);
+#endif
+                              rt_free(rth);
+                              continue;
+                        }
+                        sti();
+
+                        if (!rth_next)
+                              break;
+
+                        /*
+                         * LRU ordering.
+                         */
+
+                        if (rth->rt_lastuse + RT_CACHE_BUBBLE_THRESHOULD <
rth_next->rt_lastuse ||
+                            (rth->rt_lastuse < rth_next->rt_lastuse &&
+                             rth->rt_use < rth_next->rt_use))
+                        {
+#if RT_CACHE_DEBUG >= 2
+                              printk("rt_check_expire bubbled %02x@%08x<-
>%08x\n", i, rth->rt_dst, rth_next->rt_dst);
+#endif
+                              cli();
+                              *rthp = rth_next;
+                              rth->rt_next = rth_next->rt_next;
+                              rth_next->rt_next = rth;
+                              sti();
+                              rthp = &rth_next->rt_next;
+                              continue;
+                        }
+                        rthp = &rth->rt_next;
+                  }
+            }
+            restore_flags(flags);
+            rt_kick_free_queue();
+      }
+      ip_rt_unlock();
+}
+
+static void rt_redirect_1(__u32 dst, __u32 gw, struct device *dev)
+{
+      struct rtable *rt;
+      unsigned long hash = ip_rt_hash_code(dst);
+
+      if (gw == dev->pa_addr)
+            return;
+      if (dev != get_gw_dev(gw))
+            return;
```

```
+     rt = (struct rtable *) kmalloc(sizeof(struct rtable), GFP_ATOMIC);
+     if (rt == NULL)
+           return;
+     memset(rt, 0, sizeof(struct rtable));
+     rt->rt_flags = RTF_DYNAMIC | RTF_MODIFIED | RTF_HOST | RTF_GATEWAY |
RTF_UP;
+     rt->rt_dst = dst;
+     rt->rt_dev = dev;
+     rt->rt_gateway = gw;
+     rt->rt_src = dev->pa_addr;
+     rt->rt_mtu = dev->mtu;
+#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
+     if (dev->mtu > 576)
+           rt->rt_mtu = 576;
+#endif
+     rt->rt_lastuse  = jiffies;
+     rt->rt_refcnt  = 1;
+     rt_cache_add(hash, rt);
+     ip_rt_put(rt);
+     return;
+}
+
+static void rt_cache_flush(void)
+{
+     int i;
+     struct rtable * rth, * next;
+
+     for (i=0; i<RT_HASH_DIVISOR; i++)
+     {
+           int nr=0;
+
+           cli();
+           if (!(rth = ip_rt_hash_table[i]))
+           {
+                 sti();
+                 continue;
+           }
+
+           ip_rt_hash_table[i] = NULL;
+           sti();
+
+           for (; rth; rth=next)
+           {
+                 next = rth->rt_next;
+                 rt_cache_size--;
+                 nr++;
+                 rth->rt_next = NULL;
+                 rt_free(rth);
+           }
+#if RT_CACHE_DEBUG >= 2
+           if (nr > 0)
+                 printk("rt_cache_flush: %d@%02x\n", nr, i);
+#endif
+     }
+#if RT_CACHE_DEBUG >= 1
+     if (rt_cache_size)
+     {
```

```
+           printk("rt_cache_flush: bug rt_cache_size=%d\n", rt_cache_size);
+           rt_cache_size = 0;
+     }
+#endif
+}
+
+static void rt_garbage_collect_1(void)
+{
+     int i;
+     unsigned expire = RT_CACHE_TIMEOUT>>1;
+     struct rtable * rth, **rthp;
+     unsigned long now = jiffies;
+
+     for (;;)
+     {
+           for (i=0; i<RT_HASH_DIVISOR; i++)
+           {
+                 if (!ip_rt_hash_table[i])
+                       continue;
+                 for (rthp=&ip_rt_hash_table[i]; (rth=*rthp); rthp=&rth-
>rt_next)
+                 {
+                       if (rth->rt_lastuse + expire*(rth->rt_refcnt+1) >
now)
+                             continue;
+                       rt_cache_size--;
+                       cli();
+                       *rthp=rth->rt_next;
+                       rth->rt_next = NULL;
+                       sti();
+                       rt_free(rth);
+                       break;
+                 }
+           }
+           if (rt_cache_size < RT_CACHE_SIZE_MAX)
+                 return;
+           expire >>= 1;
+     }
+}
+
+static __inline__ void rt_req_enqueue(struct rt_req **q, struct rt_req *rtr)
+{
+     unsigned long flags;
+     struct rt_req * tail;
+
+     save_flags(flags);
+     cli();
+     tail = *q;
+     if (!tail)
+           rtr->rtr_next = rtr;
+     else
+     {
+           rtr->rtr_next = tail->rtr_next;
+           tail->rtr_next = rtr;
+     }
+     *q = rtr;
+     restore_flags(flags);
```

```
+     return;
+}
+
+/*
+ * Caller should mask interrupts.
+ */
+
+static __inline__ struct rt_req * rt_req_dequeue(struct rt_req **q)
+{
+     struct rt_req * rtr;
+
+     if (*q)
+     {
+           rtr = (*q)->rtr_next;
+           (*q)->rtr_next = rtr->rtr_next;
+           if (rtr->rtr_next == rtr)
+                 *q = NULL;
+           rtr->rtr_next = NULL;
+           return rtr;
+     }
+     return NULL;
+}
+
+/*
+   Called with masked interrupts
+ */
+
+static void rt_kick_backlog()
+{
+     if (!ip_rt_lock)
+     {
+           struct rt_req * rtr;
+
+           ip_rt_fast_lock();
+
+           while ((rtr = rt_req_dequeue(&rt_backlog)) != NULL)
+           {
+                 sti();
+                 rt_redirect_1(rtr->dst, rtr->gw, rtr->dev);
+                 kfree_s(rtr, sizeof(struct rt_req));
+                 cli();
+           }
+
+           ip_rt_bh_mask &= ~RT_BH_REDIRECT;
+
+           ip_rt_fast_unlock();
+     }
+}
+
+/*
+ * rt_{del|add|flush} called only from USER process. Waiting is OK.
+ */
+
+static int rt_del(__u32 dst, __u32 mask,
+           struct device * dev, __u32 gtw, short rt_flags, short metric)
+{
+     int retval;
```

```
+
+     while (ip_rt_lock)
+           sleep_on(&rt_wait);
+     ip_rt_fast_lock();
+     retval = fib_del_1(dst, mask, dev, gtw, rt_flags, metric);
+     ip_rt_unlock();
+     wake_up(&rt_wait);
+     return retval;
+}
+
+static void rt_add(short flags, __u32 dst, __u32 mask,
+     __u32 gw, struct device *dev, unsigned short mss,
+     unsigned long window, unsigned short irtt, short metric)
+{
+     while (ip_rt_lock)
+           sleep_on(&rt_wait);
+     ip_rt_fast_lock();
+     fib_add_1(flags, dst, mask, gw, dev, mss, window, irtt, metric);
+     ip_rt_unlock();
+     wake_up(&rt_wait);
+}
+
+void ip_rt_flush(struct device *dev)
+{
+     while (ip_rt_lock)
+           sleep_on(&rt_wait);
+     ip_rt_fast_lock();
+     fib_flush_1(dev);
+     ip_rt_unlock();
+     wake_up(&rt_wait);
+}
+
+/*
+   Called by ICMP module.
+ */
+
+void ip_rt_redirect(__u32 src, __u32 dst, __u32 gw, struct device *dev)
+{
+     struct rt_req * rtr;
+     struct rtable * rt;
+
+     rt = ip_rt_route(dst, 0);
+     if (!rt)
+           return;
+
+     if (rt->rt_gateway != src ||
+         rt->rt_dev != dev ||
+         ((gw^dev->pa_addr)&dev->pa_mask) ||
+         ip_chk_addr(gw))
+     {
+           ip_rt_put(rt);
+           return;
+     }
+     ip_rt_put(rt);
+
+     ip_rt_fast_lock();
+     if (ip_rt_lock == 1)
```

```
+	{
+		rt_redirect_1(dst, gw, dev);
+		ip_rt_unlock();
+		return;
+	}
+
+	rtr = kmalloc(sizeof(struct rt_req), GFP_ATOMIC);
+	if (rtr)
+	{
+		rtr->dst = dst;
+		rtr->gw = gw;
+		rtr->dev = dev;
+		rt_req_enqueue(&rt_backlog, rtr);
+		ip_rt_bh_mask |= RT_BH_REDIRECT;
+	}
+	ip_rt_unlock();
+}
+
+
+static __inline__ void rt_garbage_collect(void)
+{
+	if (ip_rt_lock == 1)
+	{
+		rt_garbage_collect_1();
+		return;
+	}
+	ip_rt_bh_mask |= RT_BH_GARBAGE_COLLECT;
+}
+
+static void rt_cache_add(unsigned hash, struct rtable * rth)
+{
+	unsigned long	flags;
+	struct rtable	**rthp;
+	__u32		daddr = rth->rt_dst;
+	unsigned long	now = jiffies;
+
+#if RT_CACHE_DEBUG >= 2
+	if (ip_rt_lock != 1)
+	{
+		printk("rt_cache_add: ip_rt_lock==%d\n", ip_rt_lock);
+		return;
+	}
+#endif
+
+	save_flags(flags);
+
+	if (rth->rt_dev->header_cache_bind)
+	{
+		struct rtable * rtg = rth;
+
+		if (rth->rt_gateway != daddr)
+		{
+			ip_rt_fast_unlock();
+			rtg = ip_rt_route(rth->rt_gateway, 0);
+			ip_rt_fast_lock();
+		}
+
```

```
+           if (rtg)
+           {
+                 if (rtg == rth)
+                       rtg->rt_dev->header_cache_bind(&rtg->rt_hh, rtg-
>rt_dev, ETH_P_IP, rtg->rt_dst);
+                 else
+                 {
+                       if (rtg->rt_hh)
+                             ATOMIC_INCR(&rtg->rt_hh->hh_refcnt);
+                       rth->rt_hh = rtg->rt_hh;
+                       ip_rt_put(rtg);
+                 }
+           }
+     }
+
+     if (rt_cache_size >= RT_CACHE_SIZE_MAX)
+           rt_garbage_collect();
+
+     cli();
+     rth->rt_next = ip_rt_hash_table[hash];
+#if RT_CACHE_DEBUG >= 2
+     if (rth->rt_next)
+     {
+           struct rtable * trth;
+           printk("rt_cache @%02x: %08x", hash, daddr);
+           for (trth=rth->rt_next; trth; trth=trth->rt_next)
+                 printk(" . %08x", trth->rt_dst);
+           printk("\n");
+     }
+#endif
+     ip_rt_hash_table[hash] = rth;
+     rthp = &rth->rt_next;
+     sti();
+     rt_cache_size++;
+
+     /*
+      * Cleanup duplicate (and aged off) entries.
+      */
+
+     while ((rth = *rthp) != NULL)
+     {
+
+           cli();
+           if ((!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT < now)
+               || rth->rt_dst == daddr)
+           {
+                 *rthp = rth->rt_next;
+                 rt_cache_size--;
+                 sti();
+#if RT_CACHE_DEBUG >= 2
+                 printk("rt_cache clean %02x@%08x\n", hash, rth->rt_dst);
+#endif
+                 rt_free(rth);
+                 continue;
+           }
+           sti();
+           rthp = &rth->rt_next;
```

```
+     }
+     restore_flags(flags);
 }

 /*
- *   Process a route add request from the user
+   RT should be already locked.
+
+   We could improve this by keeping a chain of say 32 struct rtable's
+   last freed for fast recycling.
+
+ */
+
+struct rtable * ip_rt_slow_route (__u32 daddr, int local)
+{
+     unsigned hash = ip_rt_hash_code(daddr)^local;
+     struct rtable * rth;
+     struct fib_node * f;
+     struct fib_info * fi;
+     __u32 saddr;
+
+#if RT_CACHE_DEBUG >= 2
+     printk("rt_cache miss @%08x\n", daddr);
+#endif
+
+     rth = kmalloc(sizeof(struct rtable), GFP_ATOMIC);
+     if (!rth)
+     {
+           ip_rt_unlock();
+           return NULL;
+     }
+
+     if (local)
+           f = fib_lookup_local(daddr);
+     else
+           f = fib_lookup (daddr);
+
+     if (f)
+     {
+           fi = f->fib_info;
+           f->fib_use++;
+     }
+
+     if (!f || (fi->fib_flags & RTF_REJECT))
+     {
+#if RT_CACHE_DEBUG >= 2
+           printk("rt_route failed @%08x\n", daddr);
+#endif
+           ip_rt_unlock();
+           kfree_s(rth, sizeof(struct rtable));
+           return NULL;
+     }
+
+     saddr = fi->fib_dev->pa_addr;
+
+     if (daddr == fi->fib_dev->pa_addr)
+     {
```

```
+           f->fib_use--;
+           if ((f = fib_loopback) != NULL)
+           {
+                 f->fib_use++;
+                 fi = f->fib_info;
+           }
+     }
+
+     if (!f)
+     {
+           ip_rt_unlock();
+           kfree_s(rth, sizeof(struct rtable));
+           return NULL;
+     }
+
+     rth->rt_dst = daddr;
+     rth->rt_src = saddr;
+     rth->rt_lastuse   = jiffies;
+     rth->rt_refcnt    = 1;
+     rth->rt_use = 1;
+     rth->rt_next      = NULL;
+     rth->rt_hh  = NULL;
+     rth->rt_gateway   = fi->fib_gateway;
+     rth->rt_dev = fi->fib_dev;
+     rth->rt_mtu = fi->fib_mtu;
+     rth->rt_window    = fi->fib_window;
+     rth->rt_irtt      = fi->fib_irtt;
+     rth->rt_tos = f->fib_tos;
+     rth->rt_flags   = fi->fib_flags | RTF_HOST;
+     if (local)
+           rth->rt_flags   |= RTF_LOCAL;
+
+     if (!(rth->rt_flags & RTF_GATEWAY))
+           rth->rt_gateway = rth->rt_dst;
+
+     if (ip_rt_lock == 1)
+           rt_cache_add(hash, rth);
+     else
+     {
+           rt_free(rth);
+#if RT_CACHE_DEBUG >= 1
+           printk("rt_cache: route to %08x was born dead\n", daddr);
+#endif
+     }
+
+     ip_rt_unlock();
+     return rth;
+}
+
+void ip_rt_put(struct rtable * rt)
+{
+     if (rt)
+           ATOMIC_DECR(&rt->rt_refcnt);
+}
+
+struct rtable * ip_rt_route(__u32 daddr, int local)
+{
```

```
+     struct rtable * rth;
+
+     ip_rt_fast_lock();
+
+     for (rth=ip_rt_hash_table[ip_rt_hash_code(daddr)^local]; rth; rth=rth-
>rt_next)
+     {
+           if (rth->rt_dst == daddr)
+           {
+                 rth->rt_lastuse = jiffies;
+                 ATOMIC_INCR(&rth->rt_use);
+                 ATOMIC_INCR(&rth->rt_refcnt);
+                 ip_rt_unlock();
+                 return rth;
+           }
+     }
+     return ip_rt_slow_route (daddr, local);
+}
+
+
+/*
+ *   Process a route add request from the user, or from a kernel
+ *   task.
  */

-static int rt_new(struct rtentry *r)
+int ip_rt_new(struct rtentry *r)
 {
      int err;
      char * devname;
@@ -465,7 +1629,7 @@ static int rt_new(struct rtentry *r)
       /*
        *    BSD emulation: Permits route add someroute gw one-of-my-addresses
        *    to indicate which iface. Not as clean as the nice Linux dev
technique
-      *    but people keep using it...
+      *    but people keep using it...  (and gated likes it ;))
        */

      if (!dev && (flags & RTF_GATEWAY))
@@ -522,8 +1686,8 @@ static int rt_new(struct rtentry *r)
       /*
        *    Add the route
        */
-
-     ip_rt_add(flags, daddr, mask, gw, dev, r->rt_mss, r->rt_window, r-
>rt_irtt, metric);
+
+     rt_add(flags, daddr, mask, gw, dev, r->rt_mss, r->rt_window, r-
>rt_irtt, metric);
      return 0;
 }

@@ -539,6 +1703,7 @@ static int rt_kill(struct rtentry *r)
      struct sockaddr_in *gtw;
      char *devname;
      int err;
```

```
+     struct device * dev = NULL;

      trg = (struct sockaddr_in *) &r->rt_dst;
      msk = (struct sockaddr_in *) &r->rt_genmask;
@@ -548,159 +1713,20 @@ static int rt_kill(struct rtentry *r)
            err = getname(devname, &devname);
            if (err)
                  return err;
+           dev = dev_get(devname);
+           putname(devname);
+           if (!dev)
+                 return -ENODEV;
      }
      /*
       * metric can become negative here if it wasn't filled in
       * but that's a fortunate accident; we really use that in rt_del.
       */
-     err=rt_del((__u32)trg->sin_addr.s_addr, (__u32)msk->sin_addr.s_addr,
devname,
+     err=rt_del((__u32)trg->sin_addr.s_addr, (__u32)msk->sin_addr.s_addr,
dev,
            (__u32)gtw->sin_addr.s_addr, r->rt_flags, r->rt_metric - 1);
-     if ( devname != NULL )
-           putname(devname);
      return err;
 }

-
-/*
- *   Called from the PROCfs module. This outputs /proc/net/route.
- *
- *   We preserve the old format but pad the buffers out. This means that
- *   we can spin over the other entries as we read them. Remember the
- *   gated BGP4 code could need to read 60,000+ routes on occasion (thats
- *   about 7Mb of data). To do that ok we will need to also cache the
- *   last route we got to (reads will generally be following on from
- *   one another without gaps).
- */
-
-int rt_get_info(char *buffer, char **start, off_t offset, int length, int
dummy)
-{
-     struct rtable *r;
-     int len=128;
-     off_t pos=0;
-     off_t begin=0;
-     char temp[129];
-
-     if(offset<128)
-           sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\tIRTT");
-     pos=128;
-
-     for (r = rt_base; r != NULL; r = r->rt_next)
-     {
-           /*
-            *    Spin through entries until we are ready
```

```
-             */
-            if(pos+128<offset)
-            {
-                  pos+=128;
-                  continue;
-            }
-
-            sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%d\t%lu\t%d\t%08lX\t%d\t%lu\t%u",
-                  r->rt_dev->name, (unsigned long)r->rt_dst, (unsigned
long)r->rt_gateway,
-                  r->rt_flags, r->rt_refcnt, r->rt_use, r->rt_metric,
-                  (unsigned long)r->rt_mask, (int)r->rt_mss, r->rt_window,
(int)r->rt_irtt);
-            sprintf(buffer+len,"%-127s\n",temp);
-            len+=128;
-            pos+=128;
-            if(pos<offset)
-            {
-                  len=0;
-                  begin=pos;
-            }
-            if(pos>offset+length)
-                  break;
-      }
-
-      *start=buffer+(offset-begin);
-      len-=(offset-begin);
-      if(len>length)
-            len=length;
-      return len;
-}
-
-/*
- *    This is hackish, but results in better code. Use "-S" to see why.
- */
-
-#define early_out ({ goto no_route; 1; })
-
-/*
- *    Route a packet. This needs to be fairly quick. Florian & Co.
- *    suggested a unified ARP and IP routing cache. Done right its
- *    probably a brilliant idea. I'd actually suggest a unified
- *    ARP/IP routing/Socket pointer cache. Volunteers welcome
- */
-
-struct rtable * ip_rt_route(__u32 daddr, struct options *opt, __u32
*src_addr)
-{
-      struct rtable *rt;
-
-      for (rt = rt_base; rt != NULL || early_out ; rt = rt->rt_next)
-      {
-            if (!((rt->rt_dst ^ daddr) & rt->rt_mask))
-                  break;
-            /*
-             *    broadcast addresses can be special cases..
```

```
-            */
-           if (rt->rt_flags & RTF_GATEWAY)
-                 continue;
-           if ((rt->rt_dev->flags & IFF_BROADCAST) &&
-               (rt->rt_dev->pa_brdaddr == daddr))
-                 break;
-     }
-
-     if(rt->rt_flags&RTF_REJECT)
-           return NULL;
-
-     if(src_addr!=NULL)
-           *src_addr= rt->rt_dev->pa_addr;
-
-     if (daddr == rt->rt_dev->pa_addr) {
-           if ((rt = rt_loopback) == NULL)
-                 goto no_route;
-     }
-     rt->rt_use++;
-     return rt;
-no_route:
-     return NULL;
-}
-
-struct rtable * ip_rt_local(__u32 daddr, struct options *opt, __u32
*src_addr)
-{
-     struct rtable *rt;
-
-     for (rt = rt_base; rt != NULL || early_out ; rt = rt->rt_next)
-     {
-           /*
-            *    No routed addressing.
-            */
-           if (rt->rt_flags&RTF_GATEWAY)
-                 continue;
-
-           if (!((rt->rt_dst ^ daddr) & rt->rt_mask))
-                 break;
-           /*
-            *    broadcast addresses can be special cases..
-            */
-
-           if ((rt->rt_dev->flags & IFF_BROADCAST) &&
-                rt->rt_dev->pa_brdaddr == daddr)
-                 break;
-     }
-
-     if(src_addr!=NULL)
-           *src_addr= rt->rt_dev->pa_addr;
-
-     if (daddr == rt->rt_dev->pa_addr) {
-           if ((rt = rt_loopback) == NULL)
-                 goto no_route;
-     }
-     rt->rt_use++;
-     return rt;
```

```
-no_route:
-     return NULL;
-}
-
 /*
  *   Handle IP routing ioctl calls. These are used to manipulate the routing
tables
  */
@@ -720,8 +1746,15 @@ int ip_rt_ioctl(unsigned int cmd, void *arg)
                   if (err)
                         return err;
                   memcpy_fromfs(&rt, arg, sizeof(struct rtentry));
-                  return (cmd == SIOCDELRT) ? rt_kill(&rt) : rt_new(&rt);
+                  return (cmd == SIOCDELRT) ? rt_kill(&rt) : ip_rt_new(&rt);
       }

       return -EINVAL;
 }
+
+void ip_rt_advice(struct rtable **rp, int advice)
+{
+     /* Thanks! */
+     return;
+}
+
--
1.6.5
```

```
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *
 *              ROUTE - implementation of the IP router.
 *
 * Version:     @(#)route.c     1.0.14  05/31/93
 *
 * Authors:     Ross Biro, <bir7@leland.Stanford.Edu>
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Alan Cox, <gw4pts@gw4pts.ampr.org>
 *              Linus Torvalds, <Linus.Torvalds@helsinki.fi>
 *
 * Fixes:
 *              Alan Cox        :       Verify area fixes.
 *              Alan Cox        :       cli() protects routing changes
 *              Rui Oliveira    :       ICMP routing table updates
 *              (rco@di.uminho.pt)      Routing table insertion and update
 *              Linus Torvalds  :       Rewrote bits to be sensible
 *              Alan Cox        :       Added BSD route gw semantics
 *              Alan Cox        :       Super /proc >4K
 *              Alan Cox        :       MTU in route table
 *              Alan Cox        :       MSS actually. Also added the window
 *                                      clamper.
 *              Sam Lantinga    :       Fixed route matching in rt_del()
 *              Alan Cox        :       Routing cache support.
 *              Alan Cox        :       Removed compatibility cruft.
 *              Alan Cox        :       RTF_REJECT support.
 *              Alan Cox        :       TCP irtt support.
 *              Jonathan Naylor :       Added Metric support.
 *      Miquel van Smoorenburg  :       BSD API fixes.
 *      Miquel van Smoorenburg  :       Metrics.
 *              Alan Cox        :       Use __u32 properly
 *              Alan Cox        :       Aligned routing errors more closely with BSD
 *                                      our system is still very different.
 *              Alan Cox        :       Faster /proc handling
 *      Alexey Kuznetsov        :       Massive rework to support tree based
routing,
 *                                      routing caches and better behaviour.
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or (at your option) any later version.
 */

#include <linux/config.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <asm/bitops.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/string.h>
#include <linux/socket.h>
#include <linux/sockios.h>
#include <linux/errno.h>
#include <linux/in.h>
#include <linux/inet.h>
#include <linux/netdevice.h>
#include <net/ip.h>
```

```
#include <net/protocol.h>
#include <net/route.h>
#include <net/tcp.h>
#include <linux/skbuff.h>
#include <net/sock.h>
#include <net/icmp.h>
#include <net/netlink.h>

/*
 * Forwarding Information Base definitions.
 */

struct fib_node
{
        struct fib_node         *fib_next;
        __u32                   fib_dst;
        unsigned long           fib_use;
        struct fib_info         *fib_info;
        short                   fib_metric;
        unsigned char           fib_tos;
};

/*
 * This structure contains data shared by many of routes.
 */

struct fib_info
{
        struct fib_info         *fib_next;
        struct fib_info         *fib_prev;
        __u32                   fib_gateway;
        struct device           *fib_dev;
        int                     fib_refcnt;
        unsigned long           fib_window;
        unsigned short          fib_flags;
        unsigned short          fib_mtu;
        unsigned short          fib_irtt;
};

struct fib_zone
{
        struct fib_zone *fz_next;
        struct fib_node **fz_hash_table;
        struct fib_node *fz_list;
        int             fz_nent;
        int             fz_logmask;
        __u32           fz_mask;
};

static struct fib_zone  *fib_zones[33];
static struct fib_zone  *fib_zone_list;
static struct fib_node  *fib_loopback = NULL;
static struct fib_info  *fib_info_list;

/*
 * Backlogging.
 */

#define RT_BH_REDIRECT          0
#define RT_BH_GARBAGE_COLLECT   1
#define RT_BH_FREE              2

struct rt_req
```

```
{
        struct rt_req * rtr_next;
        struct device *dev;
        __u32 dst;
        __u32 gw;
        unsigned char tos;
};

int                     ip_rt_lock;
unsigned                ip_rt_bh_mask;
static struct rt_req    *rt_backlog;

/*
 * Route cache.
 */

struct rtable           *ip_rt_hash_table[RT_HASH_DIVISOR];
static int              rt_cache_size;
static struct rtable    *rt_free_queue;
struct wait_queue       *rt_wait;

static void rt_kick_backlog(void);
static void rt_cache_add(unsigned hash, struct rtable * rth);
static void rt_cache_flush(void);
static void rt_garbage_collect_1(void);

/*
 * Evaluate mask length.
 */

static __inline__ int rt_logmask(__u32 mask)
{
        if (!(mask = ntohl(mask)))
                return 32;
        return ffz(~mask);
}

/*
 * Create mask from length.
 */

static __inline__ __u32 rt_mask(int logmask)
{
        if (logmask >= 32)
                return 0;
        return htonl(~((1<<logmask)-1));
}

static __inline__ unsigned fz_hash_code(__u32 dst, int logmask)
{
        return ip_rt_hash_code(ntohl(dst)>>logmask);
}

/*
 * Free FIB node.
 */

static void fib_free_node(struct fib_node * f)
{
        struct fib_info * fi = f->fib_info;
        if (!--fi->fib_refcnt)
        {
#if RT_CACHE_DEBUG >= 2
```

```
                printk("fib_free_node: fi %08x/%s is free\n", fi->fib_gateway,
fi->fib_dev->name);
#endif
                if (fi->fib_next)
                        fi->fib_next->fib_prev = fi->fib_prev;
                if (fi->fib_prev)
                        fi->fib_prev->fib_next = fi->fib_next;
                if (fi == fib_info_list)
                        fib_info_list = fi->fib_next;
        }
        kfree_s(f, sizeof(struct fib_node));
}

/*
 * Find gateway route by address.
 */

static struct fib_node * fib_lookup_gateway(__u32 dst)
{
        struct fib_zone * fz;
        struct fib_node * f;

        for (fz = fib_zone_list; fz; fz = fz->fz_next)
        {
                if (fz->fz_hash_table)
                        f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
                else
                        f = fz->fz_list;

                for ( ; f; f = f->fib_next)
                {
                        if ((dst ^ f->fib_dst) & fz->fz_mask)
                                continue;
                        if (f->fib_info->fib_flags & RTF_GATEWAY)
                                return NULL;
                        return f;
                }
        }
        return NULL;
}

/*
 * Find local route by address.
 * FIXME: I use "longest match" principle. If destination
 *        has some non-local route, I'll not search shorter matches.
 *        It's possible, I'm wrong, but I wanted to prevent following
 *        situation:
 *      route add 193.233.7.128 netmask 255.255.255.192 gw xxxxxx
 *      route add 193.233.7.0   netmask 255.255.255.0 eth1
 *        (Two ethernets connected by serial line, one is small and other is large)
 *        Host 193.233.7.129 is locally unreachable,
 *        but old (<=1.3.37) code will send packets destined for it to eth1.
 *
 */

static struct fib_node * fib_lookup_local(__u32 dst)
{
        struct fib_zone * fz;
        struct fib_node * f;

        for (fz = fib_zone_list; fz; fz = fz->fz_next)
        {
                int longest_match_found = 0;
```

```
                if (fz->fz_hash_table)
                        f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
                else
                        f = fz->fz_list;

                for ( ; f; f = f->fib_next)
                {
                        if ((dst ^ f->fib_dst) & fz->fz_mask)
                                continue;
                        if (!(f->fib_info->fib_flags & RTF_GATEWAY))
                                return f;
                        longest_match_found = 1;
                }
                if (longest_match_found)
                        return NULL;
        }
        return NULL;
}

/*
 * Main lookup routine.
 *      IMPORTANT NOTE: this algorithm has small difference from <=1.3.37 visible
 *      by user. It doesn't route non-CIDR broadcasts by default.
 *
 *      F.e.
 *              ifconfig eth0 193.233.7.65 netmask 255.255.255.192 broadcast
193.233.7.255
 *      is valid, but if you really are not able (not allowed, do not want) to
 *      use CIDR compliant broadcast 193.233.7.127, you should add host route:
 *              route add -host 193.233.7.255 eth0
 */

static struct fib_node * fib_lookup(__u32 dst)
{
        struct fib_zone * fz;
        struct fib_node * f;

        for (fz = fib_zone_list; fz; fz = fz->fz_next)
        {
                if (fz->fz_hash_table)
                        f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
                else
                        f = fz->fz_list;

                for ( ; f; f = f->fib_next)
                {
                        if ((dst ^ f->fib_dst) & fz->fz_mask)
                                continue;
                        return f;
                }
        }
        return NULL;
}

static __inline__ struct device * get_gw_dev(__u32 gw)
{
        struct fib_node * f;
        f = fib_lookup_gateway(gw);
        if (f)
                return f->fib_info->fib_dev;
        return NULL;
}
```

```
/*
 *      Used by 'rt_add()' when we can't get the netmask any other way..
 *
 *      If the lower byte or two are zero, we guess the mask based on the
 *      number of zero 8-bit net numbers, otherwise we use the "default"
 *      masks judging by the destination address and our device netmask.
 */

static __u32 unsigned long default_mask(__u32 dst)
{
        dst = ntohl(dst);
        if (IN_CLASSA(dst))
                return htonl(IN_CLASSA_NET);
        if (IN_CLASSB(dst))
                return htonl(IN_CLASSB_NET);
        return htonl(IN_CLASSC_NET);
}


/*
 *      If no mask is specified then generate a default entry.
 */

static __u32 guess_mask(__u32 dst, struct device * dev)
{
        __u32 mask;

        if (!dst)
                return 0;
        mask = default_mask(dst);
        if ((dst ^ dev->pa_addr) & mask)
                return mask;
        return dev->pa_mask;
}


/*
 *      Check if a mask is acceptable.
 */

static inline int bad_mask(__u32 mask, __u32 addr)
{
        if (addr & (mask = ~mask))
                return 1;
        mask = ntohl(mask);
        if (mask & (mask+1))
                return 1;
        return 0;
}


static int fib_del_list(struct fib_node **fp, __u32 dst,
                struct device * dev, __u32 gtw, short flags, short metric, __u32
mask)
{
        struct fib_node *f;
        int found=0;

        while((f = *fp) != NULL)
        {
                struct fib_info * fi = f->fib_info;
```

```
                /*
                 *      Make sure the destination and netmask match.
                 *      metric, gateway and device are also checked
                 *      if they were specified.
                 */
                if (f->fib_dst != dst ||
                    (gtw && fi->fib_gateway != gtw) ||
                    (metric >= 0 && f->fib_metric != metric) ||
                    (dev && fi->fib_dev != dev) )
                {
                        fp = &f->fib_next;
                        continue;
                }
                cli();
                *fp = f->fib_next;
                if (fib_loopback == f)
                        fib_loopback = NULL;
                sti();
                ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric,
fi->fib_dev->name);
                fib_free_node(f);
                found++;
        }
        return found;
}

static __inline__ int fib_del_1(__u32 dst, __u32 mask,
                struct device * dev, __u32 gtw, short flags, short metric)
{
        struct fib_node **fp;
        struct fib_zone *fz;
        int found=0;

        if (!mask)
        {
                for (fz=fib_zone_list; fz; fz = fz->fz_next)
                {
                        int tmp;
                        if (fz->fz_hash_table)
                                fp = &fz->fz_hash_table[fz_hash_code(dst,
fz->fz_logmask)];
                        else
                                fp = &fz->fz_list;

                        tmp = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
                        fz->fz_nent -= tmp;
                        found += tmp;
                }
        }
        else
        {
                if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
                {
                        if (fz->fz_hash_table)
                                fp = &fz->fz_hash_table[fz_hash_code(dst,
fz->fz_logmask)];
                        else
                                fp = &fz->fz_list;

                        found = fib_del_list(fp, dst, dev, gtw, flags, metric,
mask);
                        fz->fz_nent -= found;
                }
```

```
        }

        if (found)
        {
                rt_cache_flush();
                return 0;
        }
        return -ESRCH;
}


static struct fib_info * fib_create_info(__u32 gw, struct device * dev,
                                         unsigned short flags, unsigned short mss,
                                         unsigned long window, unsigned short irtt)
{
        struct fib_info * fi;

        if (!(flags & RTF_MSS))
        {
                mss = dev->mtu;
#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
                /*
                 *      If MTU was not specified, use default.
                 *      If you want to increase MTU for some net (local subnet)
                 *      use "route add .... mss xxx".
                 *
                 *      The MTU isnt currently always used and computed as it
                 *      should be as far as I can tell. [Still verifying this is
right]
                 */
                if ((flags & RTF_GATEWAY) && mss > 576)
                        mss = 576;
#endif
        }
        if (!(flags & RTF_WINDOW))
                window = 0;
        if (!(flags & RTF_IRTT))
                irtt = 0;

        for (fi=fib_info_list; fi; fi = fi->fib_next)
        {
                if (fi->fib_gateway != gw ||
                    fi->fib_dev != dev  ||
                    fi->fib_flags != flags ||
                    fi->fib_mtu != mss ||
                    fi->fib_window != window ||
                    fi->fib_irtt != irtt)
                        continue;
                fi->fib_refcnt++;
#if RT_CACHE_DEBUG >= 2
                printk("fib_create_info: fi %08x/%s is duplicate\n",
fi->fib_gateway, fi->fib_dev->name);
#endif
                return fi;
        }
        fi = (struct fib_info*)kmalloc(sizeof(struct fib_info), GFP_KERNEL);
        if (!fi)
                return NULL;
        memset(fi, 0, sizeof(struct fib_info));
        fi->fib_flags = flags;
        fi->fib_dev = dev;
        fi->fib_gateway = gw;
        fi->fib_mtu = mss;
```

```
        fi->fib_window = window;
        fi->fib_refcnt++;
        fi->fib_next = fib_info_list;
        fi->fib_prev = NULL;
        if (fib_info_list)
                fib_info_list->fib_prev = fi;
        fib_info_list = fi;
#if RT_CACHE_DEBUG >= 2
        printk("fib_create_info: fi %08x/%s is created\n", fi->fib_gateway,
fi->fib_dev->name);
#endif
        return fi;
}


static __inline__ void fib_add_1(short flags, __u32 dst, __u32 mask,
        __u32 gw, struct device *dev, unsigned short mss,
        unsigned long window, unsigned short irtt, short metric)
{
        struct fib_node *f, *f1;
        struct fib_node **fp;
        struct fib_node **dup_fp = NULL;
        struct fib_zone * fz;
        struct fib_info * fi;
        int logmask;

        if (flags & RTF_HOST)
                mask = 0xffffffff;
        /*
         * If mask is not specified, try to guess it.
         */
        else if (!mask)
        {
                if (!((dst ^ dev->pa_addr) & dev->pa_mask))
                {
                        mask = dev->pa_mask;
                        flags &= ~RTF_GATEWAY;
                        if (flags & RTF_DYNAMIC)
                        {
                                printk("Dynamic route to my own net rejected\n");
                                return;
                        }
                }
                else
                        mask = guess_mask(dst, dev);
                dst &= mask;
        }

        /*
         *      A gateway must be reachable and not a local address
         */

        if (gw == dev->pa_addr)
                flags &= ~RTF_GATEWAY;

        if (flags & RTF_GATEWAY)
        {
                /*
                 *      Don't try to add a gateway we can't reach..
                 */

                if (dev != get_gw_dev(gw))
                        return;
```

```
        flags |= RTF_GATEWAY;
}
else
        gw = 0;

/*
 *      Allocate an entry and fill it in.
 */

f = (struct fib_node *) kmalloc(sizeof(struct fib_node), GFP_KERNEL);
if (f == NULL)
        return;

memset(f, 0, sizeof(struct fib_node));
f->fib_dst = dst;
f->fib_metric = metric;
f->fib_tos    = 0;

if  ((fi = fib_create_info(gw, dev, flags, mss, window, irtt)) == NULL)
{
        kfree_s(f, sizeof(struct fib_node));
        return;
}
f->fib_info = fi;

logmask = rt_logmask(mask);
fz = fib_zones[logmask];


if (!fz)
{
        int i;
        fz = kmalloc(sizeof(struct fib_zone), GFP_KERNEL);
        if (!fz)
        {
                fib_free_node(f);
                return;
        }
        memset(fz, 0, sizeof(struct fib_zone));
        fz->fz_logmask = logmask;
        fz->fz_mask = mask;
        for (i=logmask-1; i>=0; i--)
                if (fib_zones[i])
                        break;
        cli();
        if (i<0)
        {
                fz->fz_next = fib_zone_list;
                fib_zone_list = fz;
        }
        else
        {
                fz->fz_next = fib_zones[i]->fz_next;
                fib_zones[i]->fz_next = fz;
        }
        fib_zones[logmask] = fz;
        sti();
}

/*
 * If zone overgrows RTZ_HASHING_LIMIT, create hash table.
 */
```

```
        if (fz->fz_nent >= RTZ_HASHING_LIMIT && !fz->fz_hash_table && logmask<32)
        {
                struct fib_node ** ht;
#if RT_CACHE_DEBUG
                printk("fib_add_1: hashing for zone %d started\n", logmask);
#endif
                ht = kmalloc(RTZ_HASH_DIVISOR*sizeof(struct rtable*), GFP_KERNEL);

                if (ht)
                {
                        memset(ht, 0, RTZ_HASH_DIVISOR*sizeof(struct fib_node*));
                        cli();
                        f1 = fz->fz_list;
                        while (f1)
                        {
                                struct fib_node * next;
                                unsigned hash = fz_hash_code(f1->fib_dst, logmask);
                                next = f1->fib_next;
                                f1->fib_next = ht[hash];
                                ht[hash] = f1;
                                f1 = next;
                        }
                        fz->fz_list = NULL;
                        fz->fz_hash_table = ht;
                        sti();
                }
        }

        if (fz->fz_hash_table)
                fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
        else
                fp = &fz->fz_list;

        /*
         * Scan list to find the first route with the same destination
         */
        while ((f1 = *fp) != NULL)
        {
                if (f1->fib_dst == dst)
                        break;
                fp = &f1->fib_next;
        }

        /*
         * Find route with the same destination and less (or equal) metric.
         */
        while ((f1 = *fp) != NULL && f1->fib_dst == dst)
        {
                if (f1->fib_metric >= metric)
                        break;
                /*
                 *      Record route with the same destination and gateway,
                 *      but less metric. We'll delete it
                 *      after instantiation of new route.
                 */
                if (f1->fib_info->fib_gateway == gw)
                        dup_fp = fp;
                fp = &f1->fib_next;
        }

        /*
         * Is it already present?
```

```
         */

        if (f1 && f1->fib_metric == metric && f1->fib_info == fi)
        {
                fib_free_node(f);
                return;
        }

        /*
         * Insert new entry to the list.
         */

        cli();
        f->fib_next = f1;
        *fp = f;
        if (!fib_loopback && (fi->fib_dev->flags & IFF_LOOPBACK))
                fib_loopback = f;
        sti();
        fz->fz_nent++;
        ip_netlink_msg(RTMSG_NEWROUTE, dst, gw, mask, flags, metric,
fi->fib_dev->name);

        /*
         *      Delete route with the same destination and gateway.
         *      Note that we should have at most one such route.
         */
        if (dup_fp)
                fp = dup_fp;
        else
                fp = &f->fib_next;

        while ((f1 = *fp) != NULL && f1->fib_dst == dst)
        {
                if (f1->fib_info->fib_gateway == gw)
                {
                        cli();
                        *fp = f1->fib_next;
                        if (fib_loopback == f1)
                                fib_loopback = NULL;
                        sti();
                        ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags, metric,
f1->fib_info->fib_dev->name);
                        fib_free_node(f1);
                        fz->fz_nent--;
                        break;
                }
                fp = &f1->fib_next;
        }
        rt_cache_flush();
        return;
}

static int rt_flush_list(struct fib_node ** fp, struct device *dev)
{
        int found = 0;
        struct fib_node *f;

        while ((f = *fp) != NULL) {
                if (f->fib_info->fib_dev != dev) {
                        fp = &f->fib_next;
                        continue;
                }
                cli();
```

```
                *fp = f->fib_next;
                if (fib_loopback == f)
                        fib_loopback = NULL;
                sti();
                fib_free_node(f);
                found++;
        }
        return found;
}

static __inline__ void fib_flush_1(struct device *dev)
{
        struct fib_zone *fz;
        int found = 0;

        for (fz = fib_zone_list; fz; fz = fz->fz_next)
        {
                if (fz->fz_hash_table)
                {
                        int i;
                        int tmp = 0;
                        for (i=0; i<RTZ_HASH_DIVISOR; i++)
                                tmp += rt_flush_list(&fz->fz_hash_table[i], dev);
                        fz->fz_nent -= tmp;
                        found += tmp;
                }
                else
                {
                        int tmp;
                        tmp = rt_flush_list(&fz->fz_list, dev);
                        fz->fz_nent -= tmp;
                        found += tmp;
                }
        }

        if (found)
                rt_cache_flush();
}


/*
 *      Called from the PROCfs module. This outputs /proc/net/route.
 *
 *      We preserve the old format but pad the buffers out. This means that
 *      we can spin over the other entries as we read them. Remember the
 *      gated BGP4 code could need to read 60,000+ routes on occasion (thats
 *      about 7Mb of data). To do that ok we will need to also cache the
 *      last route we got to (reads will generally be following on from
 *      one another without gaps).
 */

int rt_get_info(char *buffer, char **start, off_t offset, int length, int dummy)
{
        struct fib_zone *fz;
        struct fib_node *f;
        int len=0;
        off_t pos=0;
        char temp[129];
        int i;

        pos = 128;

        if (offset<128)
```

```
        {
                sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\tIRTT");
                len = 128;
        }

        while  (ip_rt_lock)
                sleep_on(&rt_wait);
        ip_rt_fast_lock();

        for (fz=fib_zone_list; fz; fz = fz->fz_next)
        {
                int maxslot;
                struct fib_node ** fp;

                if (fz->fz_nent == 0)
                        continue;

                if (pos + 128*fz->fz_nent <= offset)
                {
                        pos += 128*fz->fz_nent;
                        len = 0;
                        continue;
                }

                if (fz->fz_hash_table)
                {
                        maxslot = RTZ_HASH_DIVISOR;
                        fp      = fz->fz_hash_table;
                }
                else
                {
                        maxslot = 1;
                        fp      = &fz->fz_list;
                }

                for (i=0; i < maxslot; i++, fp++)
                {

                        for (f = *fp; f; f = f->fib_next)
                        {
                                struct fib_info * fi;
                                /*
                                 *      Spin through entries until we are ready
                                 */
                                pos += 128;

                                if (pos <= offset)
                                {
                                        len=0;
                                        continue;
                                }

                                fi = f->fib_info;
                                sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%d\t%lu\t%d\t%08lX\t%d\t%lu\t%u",
                                        fi->fib_dev->name, (unsigned
long)f->fib_dst, (unsigned long)fi->fib_gateway,
                                        fi->fib_flags, 0, f->fib_use, f->fib_metric,
                                        (unsigned long)fz->fz_mask,
(int)fi->fib_mtu, fi->fib_window, (int)fi->fib_irtt);
                                sprintf(buffer+len,"%-127s\n",temp);
```

```
                        len += 128;
                        if (pos >= offset+length)
                                goto done;
                }
        }
}

done:
        ip_rt_unlock();
        wake_up(&rt_wait);

        *start = buffer+len-(pos-offset);
        len = pos - offset;
        if (len>length)
                len = length;
        return len;
}

int rt_cache_get_info(char *buffer, char **start, off_t offset, int length, int
dummy)
{
        int len=0;
        off_t pos=0;
        char temp[129];
        struct rtable *r;
        int i;

        pos = 128;

        if (offset<128)
        {
                sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tSource\t\tMTU\tWindow\tIRTT\tHH\tARP\n");
                len = 128;
        }


        while  (ip_rt_lock)
                sleep_on(&rt_wait);
        ip_rt_fast_lock();

        for (i = 0; i<RT_HASH_DIVISOR; i++)
        {
                for (r = ip_rt_hash_table[i]; r; r = r->rt_next)
                {
                        /*
                         *      Spin through entries until we are ready
                         */
                        pos += 128;

                        if (pos <= offset)
                        {
                                len = 0;
                                continue;
                        }

                        sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%ld\t%lu\t%d\t%08lX\t%d\t%lu\t%u\t%ld\t%1d",
                                r->rt_dev->name, (unsigned long)r->rt_dst, (unsigned
long)r->rt_gateway,
                                r->rt_flags, r->rt_refcnt, r->rt_use, 0,
                                (unsigned long)r->rt_src, (int)r->rt_mtu,
r->rt_window, (int)r->rt_irtt, r->rt_hh ? r->rt_hh->hh_refcnt : -1, r->rt_hh ?
```

```
r->rt_hh->hh_uptodate : 0);
                        sprintf(buffer+len,"%-127s\n",temp);
                        len += 128;
                        if (pos >= offset+length)
                                goto done;
                }
        }

done:
        ip_rt_unlock();
        wake_up(&rt_wait);

        *start = buffer+len-(pos-offset);
        len = pos-offset;
        if (len>length)
                len = length;
        return len;
}


static void rt_free(struct rtable * rt)
{
        unsigned long flags;

        save_flags(flags);
        cli();
        if (!rt->rt_refcnt)
        {
                struct hh_cache * hh = rt->rt_hh;
                rt->rt_hh = NULL;
                if (hh && !--hh->hh_refcnt)
                {
                        restore_flags(flags);
                        kfree_s(hh, sizeof(struct hh_cache));
                }
                restore_flags(flags);
                kfree_s(rt, sizeof(struct rt_table));
                return;
        }
        rt->rt_next = rt_free_queue;
        rt->rt_flags &= ~RTF_UP;
        rt_free_queue = rt;
        ip_rt_bh_mask |= RT_BH_FREE;
#if RT_CACHE_DEBUG >= 2
        printk("rt_free: %08x\n", rt->rt_dst);
#endif
        restore_flags(flags);
}

/*
 * RT "bottom half" handlers. Called with masked inetrrupts.
 */

static __inline__ void rt_kick_free_queue(void)
{
        struct rtable *rt, **rtp;

        rtp = &rt_free_queue;

        while ((rt = *rtp) != NULL)
        {
                if  (!rt->rt_refcnt)
                {
```

```
                        struct hh_cache * hh = rt->rt_hh;
#if RT_CACHE_DEBUG >= 2
                        __u32 daddr = rt->rt_dst;
#endif
                        *rtp = rt->rt_next;
                        rt->rt_hh = NULL;
                        if (hh && !--hh->hh_refcnt)
                        {
                                sti();
                                kfree_s(hh, sizeof(struct hh_cache));
                        }
                        sti();
                        kfree_s(rt, sizeof(struct rt_table));
#if RT_CACHE_DEBUG >= 2
                        printk("rt_kick_free_queue: %08x is free\n", daddr);
#endif
                        cli();
                        continue;
                }
                rtp = &rt->rt_next;
        }
}

void ip_rt_run_bh() {
        unsigned long flags;
        save_flags(flags);
        cli();
        if (ip_rt_bh_mask && !ip_rt_lock)
        {
                if (ip_rt_bh_mask & RT_BH_REDIRECT)
                        rt_kick_backlog();

                if (ip_rt_bh_mask & RT_BH_GARBAGE_COLLECT)
                {
                        ip_rt_fast_lock();
                        ip_rt_bh_mask &= ~RT_BH_GARBAGE_COLLECT;
                        sti();
                        rt_garbage_collect_1();
                        cli();
                        ip_rt_fast_unlock();
                }

                if (ip_rt_bh_mask & RT_BH_FREE)
                        rt_kick_free_queue();
        }
        restore_flags(flags);
}


void ip_rt_check_expire()
{
        ip_rt_fast_lock();
        if (ip_rt_lock == 1)
        {
                int i;
                struct rtable *rth, **rthp;
                unsigned long flags;
                unsigned long now = jiffies;

                save_flags(flags);
                for (i=0; i<RT_HASH_DIVISOR; i++)
                {
                        rthp = &ip_rt_hash_table[i];
```

```
                        while ((rth = *rthp) != NULL)
                        {
                                struct rtable * rth_next = rth->rt_next;

                                /*
                                 * Cleanup aged off entries.
                                 */

                                cli();
                                if (!rth->rt_refcnt && rth->rt_lastuse +
RT_CACHE_TIMEOUT < now)
                                {
                                        *rthp = rth_next;
                                        sti();
                                        rt_cache_size--;
#if RT_CACHE_DEBUG >= 2
                                        printk("rt_check_expire clean %02x@%08x\n",
i, rth->rt_dst);
#endif
                                        rt_free(rth);
                                        continue;
                                }
                                sti();

                                if (!rth_next)
                                        break;

                                /*
                                 * LRU ordering.
                                 */

                                if (rth->rt_lastuse + RT_CACHE_BUBBLE_THRESHOULD <
rth_next->rt_lastuse ||
                                    (rth->rt_lastuse < rth_next->rt_lastuse &&
                                     rth->rt_use < rth_next->rt_use))
                                {
#if RT_CACHE_DEBUG >= 2
                                        printk("rt_check_expire bubbled
%02x@%08x<->%08x\n", i, rth->rt_dst, rth_next->rt_dst);
#endif
                                        cli();
                                        *rthp = rth_next;
                                        rth->rt_next = rth_next->rt_next;
                                        rth_next->rt_next = rth;
                                        sti();
                                        rthp = &rth_next->rt_next;
                                        continue;
                                }
                                rthp = &rth->rt_next;
                        }
                }
                restore_flags(flags);
                rt_kick_free_queue();
        }
        ip_rt_unlock();
}

static void rt_redirect_1(__u32 dst, __u32 gw, struct device *dev)
{
        struct rtable *rt;
        unsigned long hash = ip_rt_hash_code(dst);
```

```
        if (gw == dev->pa_addr)
                return;
        if (dev != get_gw_dev(gw))
                return;
        rt = (struct rtable *) kmalloc(sizeof(struct rtable), GFP_ATOMIC);
        if (rt == NULL)
                return;
        memset(rt, 0, sizeof(struct rtable));
        rt->rt_flags = RTF_DYNAMIC | RTF_MODIFIED | RTF_HOST | RTF_GATEWAY | RTF_UP;
        rt->rt_dst = dst;
        rt->rt_dev = dev;
        rt->rt_gateway = gw;
        rt->rt_src = dev->pa_addr;
        rt->rt_mtu = dev->mtu;
#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
        if (dev->mtu > 576)
                rt->rt_mtu = 576;
#endif
        rt->rt_lastuse  = jiffies;
        rt->rt_refcnt  = 1;
        rt_cache_add(hash, rt);
        ip_rt_put(rt);
        return;
}

static void rt_cache_flush(void)
{
        int i;
        struct rtable * rth, * next;

        for (i=0; i<RT_HASH_DIVISOR; i++)
        {
                int nr=0;

                cli();
                if (!(rth = ip_rt_hash_table[i]))
                {
                        sti();
                        continue;
                }

                ip_rt_hash_table[i] = NULL;
                sti();

                for (; rth; rth=next)
                {
                        next = rth->rt_next;
                        rt_cache_size--;
                        nr++;
                        rth->rt_next = NULL;
                        rt_free(rth);
                }
#if RT_CACHE_DEBUG >= 2
                if (nr > 0)
                        printk("rt_cache_flush: %d@%02x\n", nr, i);
#endif
        }
#if RT_CACHE_DEBUG >= 1
        if (rt_cache_size)
        {
                printk("rt_cache_flush: bug rt_cache_size=%d\n", rt_cache_size);
                rt_cache_size = 0;
        }
```

```
#endif
}

static void rt_garbage_collect_1(void)
{
        int i;
        unsigned expire = RT_CACHE_TIMEOUT>>1;
        struct rtable * rth, **rthp;
        unsigned long now = jiffies;

        for (;;)
        {
                for (i=0; i<RT_HASH_DIVISOR; i++)
                {
                        if (!ip_rt_hash_table[i])
                                continue;
                        for (rthp=&ip_rt_hash_table[i]; (rth=*rthp);
rthp=&rth->rt_next)
                        {
                                if (rth->rt_lastuse + expire*(rth->rt_refcnt+1) >
now)
                                        continue;
                                rt_cache_size--;
                                cli();
                                *rthp=rth->rt_next;
                                rth->rt_next = NULL;
                                sti();
                                rt_free(rth);
                                break;
                        }
                }
                if (rt_cache_size < RT_CACHE_SIZE_MAX)
                        return;
                expire >>= 1;
        }
}

static __inline__ void rt_req_enqueue(struct rt_req **q, struct rt_req *rtr)
{
        unsigned long flags;
        struct rt_req * tail;

        save_flags(flags);
        cli();
        tail = *q;
        if (!tail)
                rtr->rtr_next = rtr;
        else
        {
                rtr->rtr_next = tail->rtr_next;
                tail->rtr_next = rtr;
        }
        *q = rtr;
        restore_flags(flags);
        return;
}

/*
 * Caller should mask interrupts.
 */

static __inline__ struct rt_req * rt_req_dequeue(struct rt_req **q)
{
```

```
        struct rt_req * rtr;

        if (*q)
        {
                rtr = (*q)->rtr_next;
                (*q)->rtr_next = rtr->rtr_next;
                if (rtr->rtr_next == rtr)
                        *q = NULL;
                rtr->rtr_next = NULL;
                return rtr;
        }
        return NULL;
}

/*
   Called with masked interrupts
 */

static void rt_kick_backlog()
{
        if (!ip_rt_lock)
        {
                struct rt_req * rtr;

                ip_rt_fast_lock();

                while ((rtr = rt_req_dequeue(&rt_backlog)) != NULL)
                {
                        sti();
                        rt_redirect_1(rtr->dst, rtr->gw, rtr->dev);
                        kfree_s(rtr, sizeof(struct rt_req));
                        cli();
                }

                ip_rt_bh_mask &= ~RT_BH_REDIRECT;

                ip_rt_fast_unlock();
        }
}

/*
 * rt_{del|add|flush} called only from USER process. Waiting is OK.
 */

static int rt_del(__u32 dst, __u32 mask,
                struct device * dev, __u32 gtw, short rt_flags, short metric)
{
        int retval;

        while (ip_rt_lock)
                sleep_on(&rt_wait);
        ip_rt_fast_lock();
        retval = fib_del_1(dst, mask, dev, gtw, rt_flags, metric);
        ip_rt_unlock();
        wake_up(&rt_wait);
        return retval;
}

static void rt_add(short flags, __u32 dst, __u32 mask,
        __u32 gw, struct device *dev, unsigned short mss,
        unsigned long window, unsigned short irtt, short metric)
{
        while (ip_rt_lock)
```

```
                sleep_on(&rt_wait);
        ip_rt_fast_lock();
        fib_add_1(flags, dst, mask, gw, dev, mss, window, irtt, metric);
        ip_rt_unlock();
        wake_up(&rt_wait);
}

void ip_rt_flush(struct device *dev)
{
        while (ip_rt_lock)
                sleep_on(&rt_wait);
        ip_rt_fast_lock();
        fib_flush_1(dev);
        ip_rt_unlock();
        wake_up(&rt_wait);
}

/*
   Called by ICMP module.
 */

void ip_rt_redirect(__u32 src, __u32 dst, __u32 gw, struct device *dev)
{
        struct rt_req * rtr;
        struct rtable * rt;

        rt = ip_rt_route(dst, 0);
        if (!rt)
                return;

        if (rt->rt_gateway != src ||
            rt->rt_dev != dev ||
            ((gw^dev->pa_addr)&dev->pa_mask) ||
            ip_chk_addr(gw))
        {
                ip_rt_put(rt);
                return;
        }
        ip_rt_put(rt);

        ip_rt_fast_lock();
        if (ip_rt_lock == 1)
        {
                rt_redirect_1(dst, gw, dev);
                ip_rt_unlock();
                return;
        }

        rtr = kmalloc(sizeof(struct rt_req), GFP_ATOMIC);
        if (rtr)
        {
                rtr->dst = dst;
                rtr->gw = gw;
                rtr->dev = dev;
                rt_req_enqueue(&rt_backlog, rtr);
                ip_rt_bh_mask |= RT_BH_REDIRECT;
        }
        ip_rt_unlock();
}


static __inline__ void rt_garbage_collect(void)
{
```

```
        if (ip_rt_lock == 1)
        {
                rt_garbage_collect_1();
                return;
        }
        ip_rt_bh_mask |= RT_BH_GARBAGE_COLLECT;
}

static void rt_cache_add(unsigned hash, struct rtable * rth)
{
        unsigned long   flags;
        struct rtable   **rthp;
        __u32           daddr = rth->rt_dst;
        unsigned long   now = jiffies;

#if RT_CACHE_DEBUG >= 2
        if (ip_rt_lock != 1)
        {
                printk("rt_cache_add: ip_rt_lock==%d\n", ip_rt_lock);
                return;
        }
#endif

        save_flags(flags);

        if (rth->rt_dev->header_cache_bind)
        {
                struct rtable * rtg = rth;

                if (rth->rt_gateway != daddr)
                {
                        ip_rt_fast_unlock();
                        rtg = ip_rt_route(rth->rt_gateway, 0);
                        ip_rt_fast_lock();
                }

                if (rtg)
                {
                        if (rtg == rth)
                                rtg->rt_dev->header_cache_bind(&rtg->rt_hh,
rtg->rt_dev, ETH_P_IP, rtg->rt_dst);
                        else
                        {
                                if (rtg->rt_hh)
                                        ATOMIC_INCR(&rtg->rt_hh->hh_refcnt);
                                rth->rt_hh = rtg->rt_hh;
                                ip_rt_put(rtg);
                        }
                }
        }

        if (rt_cache_size >= RT_CACHE_SIZE_MAX)
                rt_garbage_collect();

        cli();
        rth->rt_next = ip_rt_hash_table[hash];
#if RT_CACHE_DEBUG >= 2
        if (rth->rt_next)
        {
                struct rtable * trth;
                printk("rt_cache @%02x: %08x", hash, daddr);
                for (trth=rth->rt_next; trth; trth=trth->rt_next)
                        printk(" . %08x", trth->rt_dst);
```

```
                printk("\n");
        }
#endif
        ip_rt_hash_table[hash] = rth;
        rthp = &rth->rt_next;
        sti();
        rt_cache_size++;

        /*
         * Cleanup duplicate (and aged off) entries.
         */

        while ((rth = *rthp) != NULL)
        {

                cli();
                if ((!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT < now)
                    || rth->rt_dst == daddr)
                {
                        *rthp = rth->rt_next;
                        rt_cache_size--;
                        sti();
#if RT_CACHE_DEBUG >= 2
                        printk("rt_cache clean %02x@%08x\n", hash, rth->rt_dst);
#endif
                        rt_free(rth);
                        continue;
                }
                sti();
                rthp = &rth->rt_next;
        }
        restore_flags(flags);
}

/*
   RT should be already locked.

   We could improve this by keeping a chain of say 32 struct rtable's
   last freed for fast recycling.

 */

struct rtable * ip_rt_slow_route (__u32 daddr, int local)
{
        unsigned hash = ip_rt_hash_code(daddr)^local;
        struct rtable * rth;
        struct fib_node * f;
        struct fib_info * fi;
        __u32 saddr;

#if RT_CACHE_DEBUG >= 2
        printk("rt_cache miss @%08x\n", daddr);
#endif

        rth = kmalloc(sizeof(struct rtable), GFP_ATOMIC);
        if (!rth)
        {
                ip_rt_unlock();
                return NULL;
        }

        if (local)
                f = fib_lookup_local(daddr);
```

```
        else
                f = fib_lookup (daddr);

        if (f)
        {
                fi = f->fib_info;
                f->fib_use++;
        }

        if (!f || (fi->fib_flags & RTF_REJECT))
        {
#if RT_CACHE_DEBUG >= 2
                printk("rt_route failed @%08x\n", daddr);
#endif
                ip_rt_unlock();
                kfree_s(rth, sizeof(struct rtable));
                return NULL;
        }

        saddr = fi->fib_dev->pa_addr;

        if (daddr == fi->fib_dev->pa_addr)
        {
                f->fib_use--;
                if ((f = fib_loopback) != NULL)
                {
                        f->fib_use++;
                        fi = f->fib_info;
                }
        }

        if (!f)
        {
                ip_rt_unlock();
                kfree_s(rth, sizeof(struct rtable));
                return NULL;
        }

        rth->rt_dst     = daddr;
        rth->rt_src     = saddr;
        rth->rt_lastuse = jiffies;
        rth->rt_refcnt  = 1;
        rth->rt_use     = 1;
        rth->rt_next    = NULL;
        rth->rt_hh      = NULL;
        rth->rt_gateway = fi->fib_gateway;
        rth->rt_dev     = fi->fib_dev;
        rth->rt_mtu     = fi->fib_mtu;
        rth->rt_window  = fi->fib_window;
        rth->rt_irtt    = fi->fib_irtt;
        rth->rt_tos     = f->fib_tos;
        rth->rt_flags   = fi->fib_flags | RTF_HOST;
        if (local)
                rth->rt_flags   |= RTF_LOCAL;

        if (!(rth->rt_flags & RTF_GATEWAY))
                rth->rt_gateway = rth->rt_dst;

        if (ip_rt_lock == 1)
                rt_cache_add(hash, rth);
        else
        {
                rt_free(rth);
```

```
#if RT_CACHE_DEBUG >= 1
                printk("rt_cache: route to %08x was born dead\n", daddr);
#endif
        }

        ip_rt_unlock();
        return rth;
}

void ip_rt_put(struct rtable * rt)
{
        if (rt)
                ATOMIC_DECR(&rt->rt_refcnt);
}

struct rtable * ip_rt_route(__u32 daddr, int local)
{
        struct rtable * rth;

        ip_rt_fast_lock();

        for (rth=ip_rt_hash_table[ip_rt_hash_code(daddr)^local]; rth;
rth=rth->rt_next)
        {
                if (rth->rt_dst == daddr)
                {
                        rth->rt_lastuse = jiffies;
                        ATOMIC_INCR(&rth->rt_use);
                        ATOMIC_INCR(&rth->rt_refcnt);
                        ip_rt_unlock();
                        return rth;
                }
        }
        return ip_rt_slow_route (daddr, local);
}


/*
 *      Process a route add request from the user, or from a kernel
 *      task.
 */

int ip_rt_new(struct rtentry *r)
{
        int err;
        char * devname;
        struct device * dev = NULL;
        unsigned long flags;
        __u32 daddr, mask, gw;
        short metric;

        /*
         *      If a device is specified find it.
         */

        if ((devname = r->rt_dev) != NULL)
        {
                err = getname(devname, &devname);
                if (err)
                        return err;
                dev = dev_get(devname);
                putname(devname);
                if (!dev)
```

```
                return -ENODEV;
        }

        /*
         *      If the device isn't INET, don't allow it
         */

        if (r->rt_dst.sa_family != AF_INET)
                return -EAFNOSUPPORT;

        /*
         *      Make local copies of the important bits
         *      We decrement the metric by one for BSD compatibility.
         */

        flags = r->rt_flags;
        daddr = (__u32) ((struct sockaddr_in *) &r->rt_dst)->sin_addr.s_addr;
        mask  = (__u32) ((struct sockaddr_in *) &r->rt_genmask)->sin_addr.s_addr;
        gw    = (__u32) ((struct sockaddr_in *) &r->rt_gateway)->sin_addr.s_addr;
        metric = r->rt_metric > 0 ? r->rt_metric - 1 : 0;

        /*
         *      BSD emulation: Permits route add someroute gw one-of-my-addresses
         *      to indicate which iface. Not as clean as the nice Linux dev
technique
         *      but people keep using it...  (and gated likes it ;))
         */

        if (!dev && (flags & RTF_GATEWAY))
        {
                struct device *dev2;
                for (dev2 = dev_base ; dev2 != NULL ; dev2 = dev2->next)
                {
                        if ((dev2->flags & IFF_UP) && dev2->pa_addr == gw)
                        {
                                flags &= ~RTF_GATEWAY;
                                dev = dev2;
                                break;
                        }
                }
        }

        /*
         *      Ignore faulty masks
         */

        if (bad_mask(mask, daddr))
                mask=0;

        /*
         *      Set the mask to nothing for host routes.
         */

        if (flags & RTF_HOST)
                mask = 0xffffffff;
        else if (mask && r->rt_genmask.sa_family != AF_INET)
                return -EAFNOSUPPORT;

        /*
         *      You can only gateway IP via IP..
         */

        if (flags & RTF_GATEWAY)
```

```
        {
                if (r->rt_gateway.sa_family != AF_INET)
                        return -EAFNOSUPPORT;
                if (!dev)
                        dev = get_gw_dev(gw);
        }
        else if (!dev)
                dev = ip_dev_check(daddr);

        /*
         *      Unknown device.
         */

        if (dev == NULL)
                return -ENETUNREACH;

        /*
         *      Add the route
         */

        rt_add(flags, daddr, mask, gw, dev, r->rt_mss, r->rt_window, r->rt_irtt,
metric);
        return 0;
}


/*
 *      Remove a route, as requested by the user.
 */

static int rt_kill(struct rtentry *r)
{
        struct sockaddr_in *trg;
        struct sockaddr_in *msk;
        struct sockaddr_in *gtw;
        char *devname;
        int err;
        struct device * dev = NULL;

        trg = (struct sockaddr_in *) &r->rt_dst;
        msk = (struct sockaddr_in *) &r->rt_genmask;
        gtw = (struct sockaddr_in *) &r->rt_gateway;
        if ((devname = r->rt_dev) != NULL)
        {
                err = getname(devname, &devname);
                if (err)
                        return err;
                dev = dev_get(devname);
                putname(devname);
                if (!dev)
                        return -ENODEV;
        }
        /*
         * metric can become negative here if it wasn't filled in
         * but that's a fortunate accident; we really use that in rt_del.
         */
        err=rt_del((__u32)trg->sin_addr.s_addr, (__u32)msk->sin_addr.s_addr, dev,
                (__u32)gtw->sin_addr.s_addr, r->rt_flags, r->rt_metric - 1);
        return err;
}

/*
 *      Handle IP routing ioctl calls. These are used to manipulate the routing
```

```
tables
 */

int ip_rt_ioctl(unsigned int cmd, void *arg)
{
        int err;
        struct rtentry rt;

        switch(cmd)
        {
                case SIOCADDRT:         /* Add a route */
                case SIOCDELRT:         /* Delete a route */
                        if (!suser())
                                return -EPERM;
                        err=verify_area(VERIFY_READ, arg, sizeof(struct rtentry));
                        if (err)
                                return err;
                        memcpy_fromfs(&rt, arg, sizeof(struct rtentry));
                        return (cmd == SIOCDELRT) ? rt_kill(&rt) : ip_rt_new(&rt);
        }

        return -EINVAL;
}

void ip_rt_advice(struct rtable **rp, int advice)
{
        /* Thanks! */
        return;
}
```

```
/*
 * INET                An implementation of the TCP/IP protocol suite for the LINUX
 *                     operating system.  INET is implemented using the  BSD Socket
 *                     interface as the means of communication with the user level.
 *
 *                     ROUTE - implementation of the IP router.
 *
 * Version:    @(#)route.c     1.0.14   05/31/93
 *
 * Authors:    Ross Biro, <bir7@leland.Stanford.Edu>
 *             Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *             Alan Cox, <gw4pts@gw4pts.ampr.org>
 *             Linus Torvalds, <Linus.Torvalds@helsinki.fi>
 *
 * Fixes:
 *             Alan Cox        :       Verify area fixes.
 *             Alan Cox        :       cli() protects routing changes
 *             Rui Oliveira    :       ICMP routing table updates
 *             (rco@di.uminho.pt)      Routing table insertion and update
 *             Linus Torvalds  :       Rewrote bits to be sensible
 *             Alan Cox        :       Added BSD route gw semantics
 *             Alan Cox        :       Super /proc >4K
 *             Alan Cox        :       MTU in route table
 *             Alan Cox        :       MSS actually. Also added the window
 *                                     clamper.
 *             Sam Lantinga    :       Fixed route matching in rt_del()
 *             Alan Cox        :       Routing cache support.
 *             Alan Cox        :       Removed compatibility cruft.
 *             Alan Cox        :       RTF_REJECT support.
 *             Alan Cox        :       TCP irtt support.
 *             Jonathan Naylor         :       Added Metric support.
 *      Miquel van Smoorenburg         :       BSD API fixes.
 *      Miquel van Smoorenburg         :       Metrics.
 *             Alan Cox        :       Use __u32 properly
 *             Alan Cox        :       Aligned routing errors more closely with BSD
 *                                     our system is still very different.
 *             Alan Cox        :       Faster /proc handling
 *      Alexey Kuznetsov       :       Massive rework to support tree based routing,
 *                                     routing caches and better behaviour.
 *
 *             Olaf Erb        :       irtt wasnt being copied right.
 *
 *             This program is free software; you can redistribute it and/or
 *             modify it under the terms of the GNU General Public License
 *             as published by the Free Software Foundation; either version
 *             2 of the License, or (at your option) any later version.
 */

#include <linux/config.h>
#include <asm/segment.h>
#include <asm/system.h>
```

```
52  #include <asm/bitops.h>
53  #include <linux/types.h>
54  #include <linux/kernel.h>
55  #include <linux/sched.h>
56  #include <linux/mm.h>
57  #include <linux/string.h>
58  #include <linux/socket.h>
59  #include <linux/sockios.h>
60  #include <linux/errno.h>
61  #include <linux/in.h>
62  #include <linux/inet.h>
63  #include <linux/netdevice.h>
64  #include <net/ip.h>
65  #include <net/protocol.h>
66  #include <net/route.h>
67  #include <net/tcp.h>
68  #include <linux/skbuff.h>
69  #include <net/sock.h>
70  #include <net/icmp.h>
71  #include <net/netlink.h>
72
73  /*
74   * Forwarding Information Base definitions.
75   */
76
77  struct fib_node
78  {
79          struct fib_node         *fib_next;
80          __u32                   fib_dst;
81          unsigned long           fib_use;
82          struct fib_info         *fib_info;
83          short                   fib_metric;
84          unsigned char           fib_tos;
85  };
86
87  /*
88   * This structure contains data shared by many of routes.
89   */
90
91  struct fib_info
92  {
93          struct fib_info         *fib_next;
94          struct fib_info         *fib_prev;
95          __u32                   fib_gateway;
96          struct device           *fib_dev;
97          int                     fib_refcnt;
98          unsigned long           fib_window;
99          unsigned short          fib_flags;
100         unsigned short          fib_mtu;
101         unsigned short          fib_irtt;
102 };
```

```
103
104     struct fib_zone
105     {
106             struct fib_zone  *fz_next;
107             struct fib_node  **fz_hash_table;
108             struct fib_node  *fz_list;
109             int              fz_nent;
110             int              fz_logmask;
111             __u32            fz_mask;
112     };
113
114     static struct fib_zone     *fib_zones[33];
115     static struct fib_zone     *fib_zone_list;
116     static struct fib_node     *fib_loopback = NULL;
117     static struct fib_info     *fib_info_list;
118
119     /*
120      * Backlogging.
121      */
122
123     #define RT_BH_REDIRECT          0
124     #define RT_BH_GARBAGE_COLLECT       1
125     #define RT_BH_FREE          2
126
127     struct rt_req
128     {
129             struct rt_req * rtr_next;
130             struct device *dev;
131             __u32 dst;
132             __u32 gw;
133             unsigned char tos;
134     };
135
136     int                     ip_rt_lock;
137     unsigned                ip_rt_bh_mask;
138     static struct rt_req    *rt_backlog;
139
140     /*
141      * Route cache.
142      */
143
144     struct rtable           *ip_rt_hash_table[RT_HASH_DIVISOR];
145     static int              rt_cache_size;
146     static struct rtable    *rt_free_queue;
147     struct wait_queue       *rt_wait;
148
149     static void rt_kick_backlog(void);
150     static void rt_cache_add(unsigned hash, struct rtable * rth);
151     static void rt_cache_flush(void);
152     static void rt_garbage_collect_1(void);
153
```

```
154  /*
155   * Evaluate mask length.
156   */
157
158  static __inline__ int rt_logmask(__u32 mask)
159  {
160          if (!(mask = ntohl(mask)))
161                  return 32;
162          return ffz(~mask);
163  }
164
165  /*
166   * Create mask from length.
167   */
168
169  static __inlinc__ __u32 rt_mask(int logmask)
170  {
171          if (logmask >= 32)
172                  return 0;
173          return htonl(~((1<<logmask)-1));
174  }
175
176  static __inline__ unsigned fz_hash_code(__u32 dst, int logmask)
177  {
178          return ip_rt_hash_code(ntohl(dst)>>logmask);
179  }
180
181  /*
182   * Free FIB node.
183   */
184
185  static void fib_free_node(struct fib_node * f)
186  {
187          struct fib_info * fi = f->fib_info;
188          if (!--fi->fib_refcnt)
189          {
190  #if RT_CACHE_DEBUG >= 2
191                  printk("fib_free_node: fi %08x/%s is free\n", fi->fib_gateway, fi->fib_dev->name);
192  #endif
193                  if (fi->fib_next)
194                          fi->fib_next->fib_prev = fi->fib_prev;
195                  if (fi->fib_prev)
196                          fi->fib_prev->fib_next = fi->fib_next;
197                  if (fi == fib_info_list)
198                          fib_info_list = fi->fib_next;
199          }
200          kfree_s(f, sizeof(struct fib_node));
201  }
202
203  /*
204   * Find gateway route by address.
```

```
205     */
206
207     static struct fib_node * fib_lookup_gateway(__u32 dst)
208     {
209             struct fib_zone * fz;
210             struct fib_node * f;
211
212             for (fz = fib_zone_list; fz; fz = fz->fz_next)
213             {
214                     if (fz->fz_hash_table)
215                             f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
216                     else
217                             f = fz->fz_list;
218
219                     for ( ; f; f = f->fib_next)
220                     {
221                             if ((dst ^ f->fib_dst) & fz->fz_mask)
222                                     continue;
223                             if (f->fib_info->fib_flags & RTF_GATEWAY)
224                                     return NULL;
225                             return f;
226                     }
227             }
228             return NULL;
229     }
230
231     /*
232      * Find local route by address.
233      * FIXME: I use "longest match" principle. If destination
234      *          has some non-local route, I'll not search shorter matches.
235      *          It's possible, I'm wrong, but I wanted to prevent following
236      *          situation:
237      *          route add 193.233.7.128 netmask 255.255.255.192 gw xxxxxx
238      *          route add 193.233.7.0    netmask 255.255.255.0 eth1
239      *          (Two ethernets connected by serial line, one is small and other is large)
240      *          Host 193.233.7.129 is locally unreachable,
241      *          but old (<=1.3.37) code will send packets destined for it to eth1.
242      *
243      */
244
245     static struct fib_node * fib_lookup_local(__u32 dst)
246     {
247             struct fib_zone * fz;
248             struct fib_node * f;
249
250             for (fz = fib_zone_list; fz; fz = fz->fz_next)
251             {
252                     int longest_match_found = 0;
253
254                     if (fz->fz_hash_table)
255                             f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
```

```
256                     else
257                             f = fz->fz_list;
258
259                     for ( ; f; f = f->fib_next)
260                     {
261                             if ((dst ^ f->fib_dst) & fz->fz_mask)
262                                     continue;
263                             if (!(f->fib_info->fib_flags & RTF_GATEWAY))
264                                     return f;
265                             longest_match_found = 1;
266                     }
267                     if (longest_match_found)
268                             return NULL;
269             }
270             return NULL;
271     }
272
273     /*
274      * Main lookup routine.
275      *      IMPORTANT NOTE: this algorithm has small difference from <=1.3.37 visible
276      *      by user. It doesn't route non-CIDR broadcasts by default.
277      *
278      *      F.e.
279      *              ifconfig eth0 193.233.7.65 netmask 255.255.255.192 broadcast 193.233.7.255
280      *      is valid, but if you really are not able (not allowed, do not want) to
281      *      use CIDR compliant broadcast 193.233.7.127, you should add host route:
282      *              route add -host 193.233.7.255 eth0
283      */
284
285     static struct fib_node * fib_lookup(__u32 dst)
286     {
287             struct fib_zone * fz;
288             struct fib_node * f;
289
290             for (fz = fib_zone_list; fz; fz = fz->fz_next)
291             {
292                     if (fz->fz_hash_table)
293                             f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
294                     else
295                             f = fz->fz_list;
296
297                     for ( ; f; f = f->fib_next)
298                     {
299                             if ((dst ^ f->fib_dst) & fz->fz_mask)
300                                     continue;
301                             return f;
302                     }
303             }
304             return NULL;
305     }
306
```

```
307  static __inline__ struct device * get_gw_dev(__u32 gw)
308  {
309          struct fib_node * f;
310          f = fib_lookup_gateway(gw);
311          if (f)
312                  return f->fib_info->fib_dev;
313          return NULL;
314  }
315
316  /*
317   *      Used by 'rt_add()' when we can't get the netmask any other way..
318   *
319   *      If the lower byte or two are zero, we guess the mask based on the
320   *      number of zero 8-bit net numbers, otherwise we use the "default"
321   *      masks judging by the destination address and our device netmask.
322   */
323
324  static __u32 unsigned long default_mask(__u32 dst)
325  {
326          dst = ntohl(dst);
327          if (IN_CLASSA(dst))
328                  return htonl(IN_CLASSA_NET);
329          if (IN_CLASSB(dst))
330                  return htonl(IN_CLASSB_NET);
331          return htonl(IN_CLASSC_NET);
332  }
333
334
335  /*
336   *      If no mask is specified then generate a default entry.
337   */
338
339  static __u32 guess_mask(__u32 dst, struct device * dev)
340  {
341          __u32 mask;
342
343          if (!dst)
344                  return 0;
345          mask = default_mask(dst);
346          if ((dst ^ dev->pa_addr) & mask)
347                  return mask;
348          return dev->pa_mask;
349  }
350
351
352  /*
353   *      Check if a mask is acceptable.
354   */
355
356  static inline int bad_mask(__u32 mask, __u32 addr)
357  {
```

```
358             if (addr & (mask = ~mask))
359                     return 1;
360             mask = ntohl(mask);
361             if (mask & (mask+1))
362                     return 1;
363             return 0;
364     }
365
366
367     static int fib_del_list(struct fib_node **fp, __u32 dst,
368                     struct device * dev, __u32 gtw, short flags, short metric, __u32 mask)
369     {
370             struct fib_node *f;
371             int found=0;
372
373             while((f = *fp) != NULL)
374             {
375                     struct fib_info * fi = f->fib_info;
376
377                     /*
378                      *      Make sure the destination and netmask match.
379                      *      metric, gateway and device are also checked
380                      *      if they were specified.
381                      */
382                     if (f->fib_dst != dst ||
383                        (gtw && fi->fib_gateway != gtw) ||
384                        (metric >= 0 && f->fib_metric != metric) ||
385                        (dev && fi->fib_dev != dev) )
386                     {
387                             fp = &f->fib_next;
388                             continue;
389                     }
390                     cli();
391                     *fp = f->fib_next;
392                     if (fib_loopback == f)
393                             fib_loopback = NULL;
394                     sti();
395                     ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric, fi->fib_dev->name);
396                     fib_free_node(f);
397                     found++;
398             }
399             return found;
400     }
401
402     static __inline__ int fib_del_1(__u32 dst, __u32 mask,
403                     struct device * dev, __u32 gtw, short flags, short metric)
404     {
405             struct fib_node **fp;
406             struct fib_zone *fz;
407             int found=0;
408
```

```
409             if (!mask)
410             {
411                     for (fz=fib_zone_list; fz; fz = fz->fz_next)
412                     {
413                             int tmp;
414                             if (fz->fz_hash_table)
415                                     fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
416                             else
417                                     fp = &fz->fz_list;
418
419                             tmp = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
420                             fz->fz_nent -= tmp;
421                             found += tmp;
422                     }
423             }
424             else
425             {
426                     if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
427                     {
428                             if (fz->fz_hash_table)
429                                     fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
430                             else
431                                     fp = &fz->fz_list;
432
433                             found = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
434                             fz->fz_nent -= found;
435                     }
436             }
437
438             if (found)
439             {
440                     rt_cache_flush();
441                     return 0;
442             }
443             return -ESRCH;
444     }
445
446
447     static struct fib_info * fib_create_info(__u32 gw, struct device * dev,
448                                              unsigned short flags, unsigned short mss,
449                                              unsigned long window, unsigned short irtt)
450     {
451             struct fib_info * fi;
452
453             if (!(flags & RTF_MSS))
454             {
455                     mss = dev->mtu;
456     #ifdef CONFIG_NO_PATH_MTU_DISCOVERY
457                     /*
458                      *      If MTU was not specified, use default.
459                      *      If you want to increase MTU for some net (local subnet)
```

```
460                     *       use "route add .... mss xxx".
461                     *
462                     *       The MTU isnt currently always used and computed as it
463                     *       should be as far as I can tell. [Still verifying this is right]
464                     */
465                    if ((flags & RTF_GATEWAY) && mss > 576)
466                            mss = 576;
467    #endif
468            }
469            if (!(flags & RTF_WINDOW))
470                    window = 0;
471            if (!(flags & RTF_IRTT))
472                    irtt = 0;
473
474            for (fi=fib_info_list; fi; fi = fi->fib_next)
475            {
476                    if (fi->fib_gateway != gw ||
477                       fi->fib_dev != dev  ||
478                       fi->fib_flags != flags ||
479                       fi->fib_mtu != mss ||
480                       fi->fib_window != window ||
481                       fi->fib_irtt != irtt)
482                            continue;
483                    fi->fib_refcnt++;
484    #if RT_CACHE_DEBUG >= 2
485                    printk("fib_create_info: fi %08x/%s is duplicate\n", fi->fib_gateway, fi->fib_dev->name);
486    #endif
487                    return fi;
488            }
489            fi = (struct fib_info*)kmalloc(sizeof(struct fib_info), GFP_KERNEL);
490            if (!fi)
491                    return NULL;
492            memset(fi, 0, sizeof(struct fib_info));
493            fi->fib_flags = flags;
494            fi->fib_dev = dev;
495            fi->fib_gateway = gw;
496            fi->fib_mtu = mss;
497            fi->fib_window = window;
498            fi->fib_refcnt++;
499            fi->fib_next = fib_info_list;
500            fi->fib_prev = NULL;
501            fi->fib_irtt = irtt;
502            if (fib_info_list)
503                    fib_info_list->fib_prev = fi;
504            fib_info_list = fi;
505    #if RT_CACHE_DEBUG >= 2
506            printk("fib_create_info: fi %08x/%s is created\n", fi->fib_gateway, fi->fib_dev->name);
507    #endif
508            return fi;
509    }
510
```

```
511
512 static __inline__ void fib_add_1(short flags, __u32 dst, __u32 mask,
513         __u32 gw, struct device *dev, unsigned short mss,
514         unsigned long window, unsigned short irtt, short metric)
515 {
516         struct fib_node *f, *f1;
517         struct fib_node **fp;
518         struct fib_node **dup_fp = NULL;
519         struct fib_zone * fz;
520         struct fib_info * fi;
521         int logmask;
522
523         if (flags & RTF_HOST)
524                 mask = 0xffffffff;
525         /*
526          * If mask is not specified, try to guess it.
527          */
528         else if (!mask)
529         {
530                 if (!((dst ^ dev->pa_addr) & dev->pa_mask))
531                 {
532                         mask = dev->pa_mask;
533                         flags &= ~RTF_GATEWAY;
534                         if (flags & RTF_DYNAMIC)
535                         {
536                                 printk("Dynamic route to my own net rejected\n");
537                                 return;
538                         }
539                 }
540                 else
541                         mask = guess_mask(dst, dev);
542                 dst &= mask;
543         }
544
545         /*
546          *      A gateway must be reachable and not a local address
547          */
548
549         if (gw == dev->pa_addr)
550                 flags &= ~RTF_GATEWAY;
551
552         if (flags & RTF_GATEWAY)
553         {
554                 /*
555                  *      Don't try to add a gateway we can't reach..
556                  */
557
558                 if (dev != get_gw_dev(gw))
559                         return;
560
561                 flags |= RTF_GATEWAY;
```

```
562             }
563             else
564                     gw = 0;
565
566             /*
567              *      Allocate an entry and fill it in.
568              */
569
570             f = (struct fib_node *) kmalloc(sizeof(struct fib_node), GFP_KERNEL);
571             if (f == NULL)
572                     return;
573
574             memset(f, 0, sizeof(struct fib_node));
575             f->fib_dst = dst;
576             f->fib_metric = metric;
577             f->fib_tos    = 0;
578
579             if  ((fi = fib_create_info(gw, dev, flags, mss, window, irtt)) == NULL)
580             {
581                     kfree_s(f, sizeof(struct fib_node));
582                     return;
583             }
584             f->fib_info = fi;
585
586             logmask = rt_logmask(mask);
587             fz = fib_zones[logmask];
588
589
590             if (!fz)
591             {
592                     int i;
593                     fz = kmalloc(sizeof(struct fib_zone), GFP_KERNEL);
594                     if (!fz)
595                     {
596                             fib_free_node(f);
597                             return;
598                     }
599                     memset(fz, 0, sizeof(struct fib_zone));
600                     fz->fz_logmask = logmask;
601                     fz->fz_mask = mask;
602                     for (i=logmask-1; i>=0; i--)
603                             if (fib_zones[i])
604                                     break;
605                     cli();
606                     if (i<0)
607                     {
608                             fz->fz_next = fib_zone_list;
609                             fib_zone_list = fz;
610                     }
611                     else
612                     {
```

```
613                             fz->fz_next = fib_zones[i]->fz_next;
614                             fib_zones[i]->fz_next = fz;
615                     }
616                     fib_zones[logmask] = fz;
617                     sti();
618             }
619
620             /*
621              * If zone overgrows RTZ_HASHING_LIMIT, create hash table.
622              */
623
624             if (fz->fz_nent >= RTZ_HASHING_LIMIT && !fz->fz_hash_table && logmask<32)
625             {
626                     struct fib_node ** ht;
627     #if RT_CACHE_DEBUG
628                     printk("fib_add_1: hashing for zone %d started\n", logmask);
629     #endif
630                     ht = kmalloc(RTZ_HASH_DIVISOR*sizeof(struct rtable*), GFP_KERNEL);
631
632                     if (ht)
633                     {
634                             memset(ht, 0, RTZ_HASH_DIVISOR*sizeof(struct fib_node*));
635                             cli();
636                             f1 = fz->fz_list;
637                             while (f1)
638                             {
639                                     struct fib_node * next;
640                                     unsigned hash = fz_hash_code(f1->fib_dst, logmask);
641                                     next = f1->fib_next;
642                                     f1->fib_next = ht[hash];
643                                     ht[hash] = f1;
644                                     f1 = next;
645                             }
646                             fz->fz_list = NULL;
647                             fz->fz_hash_table = ht;
648                             sti();
649                     }
650             }
651
652             if (fz->fz_hash_table)
653                     fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
654             else
655                     fp = &fz->fz_list;
656
657             /*
658              * Scan list to find the first route with the same destination
659              */
660             while ((f1 = *fp) != NULL)
661             {
662                     if (f1->fib_dst == dst)
663                             break;
```

```
664             fp = &f1->fib_next;
665     }
666
667     /*
668      * Find route with the same destination and less (or equal) metric.
669      */
670     while ((f1 = *fp) != NULL && f1->fib_dst == dst)
671     {
672             if (f1->fib_metric >= metric)
673                     break;
674             /*
675              *      Record route with the same destination and gateway,
676              *      but less metric. We'll delete it
677              *      after instantiation of new route.
678              */
679             if (f1->fib_info->fib_gateway == gw)
680                     dup_fp = fp;
681             fp = &f1->fib_next;
682     }
683
684     /*
685      * Is it already present?
686      */
687
688     if (f1 && f1->fib_metric == metric && f1->fib_info == fi)
689     {
690             fib_free_node(f);
691             return;
692     }
693
694     /*
695      * Insert new entry to the list.
696      */
697
698     cli();
699     f->fib_next = f1;
700     *fp = f;
701     if (!fib_loopback && (fi->fib_dev->flags & IFF_LOOPBACK))
702             fib_loopback = f;
703     sti();
704     fz->fz_nent++;
705     ip_netlink_msg(RTMSG_NEWROUTE, dst, gw, mask, flags, metric, fi->fib_dev->name);
706
707     /*
708      *      Delete route with the same destination and gateway.
709      *      Note that we should have at most one such route.
710      */
711     if (dup_fp)
712             fp = dup_fp;
713     else
714             fp = &f->fib_next;
```

```
715
716             while ((f1 = *fp) != NULL && f1->fib_dst == dst)
717             {
718                     if (f1->fib_info->fib_gateway == gw)
719                     {
720                             cli();
721                             *fp = f1->fib_next;
722                             if (fib_loopback == f1)
723                                     fib_loopback = NULL;
724                             sti();
725                             ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags, metric, f1->fib_info-
726     >fib_dev->name);
727                             fib_free_node(f1);
728                             fz->fz_nent--;
729                             break;
730                     }
731                     fp = &f1->fib_next;
732             }
733             rt_cache_flush();
734             return;
735     }
736
737     static int rt_flush_list(struct fib_node ** fp, struct device *dev)
738     {
739             int found = 0;
740             struct fib_node *f;
741
742             while ((f = *fp) != NULL) {
743                     if (f->fib_info->fib_dev != dev) {
744                             fp = &f->fib_next;
745                             continue;
746                     }
747                     cli();
748                     *fp = f->fib_next;
749                     if (fib_loopback == f)
750                             fib_loopback = NULL;
751                     sti();
752                     fib_free_node(f);
753                     found++;
754             }
755             return found;
756     }
757
758     static __inline__ void fib_flush_1(struct device *dev)
759     {
760             struct fib_zone *fz;
761             int found = 0;
762
763             for (fz = fib_zone_list; fz; fz = fz->fz_next)
764             {
765                     if (fz->fz_hash_table)
```

```
766                     {
767                             int i;
768                             int tmp = 0;
769                             for (i=0; i<RTZ_HASH_DIVISOR; i++)
770                                     tmp += rt_flush_list(&fz->fz_hash_table[i], dev);
771                             fz->fz_nent -= tmp;
772                             found += tmp;
773                     }
774                     else
775                     {
776                             int tmp;
777                             tmp = rt_flush_list(&fz->fz_list, dev);
778                             fz->fz_nent -= tmp;
779                             found += tmp;
780                     }
781             }
782
783             if (found)
784                     rt_cache_flush();
785     }
786
787
788     /*
789      *      Called from the PROCfs module. This outputs /proc/net/route.
790      *
791      *      We preserve the old format but pad the buffers out. This means that
792      *      we can spin over the other entries as we read them. Remember the
793      *      gated BGP4 code could need to read 60,000+ routes on occasion (thats
794      *      about 7Mb of data). To do that ok we will need to also cache the
795      *      last route we got to (reads will generally be following on from
796      *      one another without gaps).
797      */
798
799     int rt_get_info(char *buffer, char **start, off_t offset, int length, int dummy)
800     {
801             struct fib_zone *fz;
802             struct fib_node *f;
803             int len=0;
804             off_t pos=0;
805             char temp[129];
806             int i;
807
808             pos = 128;
809
810             if (offset<128)
811             {
812                     sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
813     \tFlags\tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\tIRTT");
814                     len = 128;
815             }
816
```

```
817             while  (ip_rt_lock)
818                     sleep_on(&rt_wait);
819             ip_rt_fast_lock();
820
821             for (fz=fib_zone_list; fz; fz = fz->fz_next)
822             {
823                     int maxslot;
824                     struct fib_node ** fp;
825
826                     if (fz->fz_nent == 0)
827                             continue;
828
829                     if (pos + 128*fz->fz_nent <= offset)
830                     {
831                             pos += 128*fz->fz_nent;
832                             len = 0;
833                             continue;
834                     }
835
836                     if (fz->fz_hash_table)
837                     {
838                             maxslot = RTZ_HASH_DIVISOR;
839                             fp      = fz->fz_hash_table;
840                     }
841                     else
842                     {
843                             maxslot= 1;
844                             fp      = &fz->fz_list;
845                     }
846
847                     for (i=0; i < maxslot; i++, fp++)
848                     {
849
850                             for (f = *fp; f; f = f->fib_next)
851                             {
852                                     struct fib_info * fi;
853                                     /*
854                                      *      Spin through entries until we are ready
855                                      */
856                                     pos += 128;
857
858                                     if (pos <= offset)
859                                     {
860                                             len=0;
861                                             continue;
862                                     }
863
864                                     fi = f->fib_info;
865                                     sprintf(temp,
866     "%s\t%08lX\t%08lX\t%02X\t%d\t%lu\t%d\t%08lX\t%d\t%lu\t%u",
```

```
867                                             fi->fib_dev->name, (unsigned long)f->fib_dst, (unsigned long)fi-
868     >fib_gateway,
869                                             fi->fib_flags, 0, f->fib_use, f->fib_metric,
870                                             (unsigned long)fz->fz_mask, (int)fi->fib_mtu, fi->fib_window, (int)fi-
871     >fib_irtt);
872                                     sprintf(buffer+len,"%-127s\n",temp);
873
874                                     len += 128;
875                                     if (pos >= offset+length)
876                                             goto done;
877                             }
878                     }
879             }
880
881     done:
882             ip_rt_unlock();
883             wake_up(&rt_wait);
884
885             *start = buffer+len-(pos-offset);
886             len = pos - offset;
887             if (len>length)
888                     len = length;
889             return len;
890     }
891
892     int rt_cache_get_info(char *buffer, char **start, off_t offset, int length, int dummy)
893     {
894             int len=0;
895             off_t pos=0;
896             char temp[129];
897             struct rtable *r;
898             int i;
899
900             pos = 128;
901
902             if (offset<128)
903             {
904                     sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
905     \tFlags\tRefCnt\tUse\tMetric\tSource\t\tMTU\tWindow\tIRTT\tHH\tARP\n");
906                     len = 128;
907             }
908
909
910             while  (ip_rt_lock)
911                     sleep_on(&rt_wait);
912             ip_rt_fast_lock();
913
914             for (i = 0; i<RT_HASH_DIVISOR; i++)
915             {
916                     for (r = ip_rt_hash_table[i]; r; r = r->rt_next)
917                     {
```

```
                                /*
                                 *      Spin through entries until we are ready
                                 */
                                pos += 128;

                                if (pos <= offset)
                                {
                                        len = 0;
                                        continue;
                                }

                                sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%ld\t%lu\t%d\t%08lX\t%d\t%lu\t%u\t%ld\t%1d",
                                        r->rt_dev->name, (unsigned long)r->rt_dst, (unsigned long)r->rt_gateway,
                                        r->rt_flags, r->rt_refcnt, r->rt_use, 0,
                                        (unsigned long)r->rt_src, (int)r->rt_mtu, r->rt_window, (int)r->rt_irtt, r->rt_hh
? r->rt_hh->hh_refcnt : -1, r->rt_hh ? r->rt_hh->hh_uptodate : 0);
                                sprintf(buffer+len,"%-127s\n",temp);
                                len += 128;
                                if (pos >= offset+length)
                                        goto done;
                        }
                }

done:
        ip_rt_unlock();
        wake_up(&rt_wait);

        *start = buffer+len-(pos-offset);
        len = pos-offset;
        if (len>length)
                len = length;
        return len;
}


static void rt_free(struct rtable * rt)
{
        unsigned long flags;

        save_flags(flags);
        cli();
        if (!rt->rt_refcnt)
        {
                struct hh_cache * hh = rt->rt_hh;
                rt->rt_hh = NULL;
                if (hh && !--hh->hh_refcnt)
                {
                        restore_flags(flags);
                        kfree_s(hh, sizeof(struct hh_cache));
                }
```

```
969                     restore_flags(flags);
970                     kfree_s(rt, sizeof(struct rt_table));
971                     return;
972             }
973             rt->rt_next = rt_free_queue;
974             rt->rt_flags &= ~RTF_UP;
975             rt_free_queue = rt;
976             ip_rt_bh_mask |= RT_BH_FREE;
977     #if RT_CACHE_DEBUG >= 2
978             printk("rt_free: %08x\n", rt->rt_dst);
979     #endif
980             restore_flags(flags);
981     }
982
983     /*
984      * RT "bottom half" handlers. Called with masked interrupts.
985      */
986
987     static __inline__ void rt_kick_free_queue(void)
988     {
989             struct rtable *rt, **rtp;
990
991             rtp = &rt_free_queue;
992
993             while ((rt = *rtp) != NULL)
994             {
995                     if  (!rt->rt_refcnt)
996                     {
997                             struct hh_cache * hh = rt->rt_hh;
998     #if RT_CACHE_DEBUG >= 2
999                             __u32 daddr = rt->rt_dst;
1000    #endif
1001                            *rtp = rt->rt_next;
1002                            rt->rt_hh = NULL;
1003                            if (hh && !--hh->hh_refcnt)
1004                            {
1005                                    sti();
1006                                    kfree_s(hh, sizeof(struct hh_cache));
1007                            }
1008                            sti();
1009                            kfree_s(rt, sizeof(struct rt_table));
1010    #if RT_CACHE_DEBUG >= 2
1011                            printk("rt_kick_free_queue: %08x is free\n", daddr);
1012    #endif
1013                            cli();
1014                            continue;
1015                    }
1016                    rtp = &rt->rt_next;
1017            }
1018    }
1019
```

```
1020  void ip_rt_run_bh() {
1021          unsigned long flags;
1022          save_flags(flags);
1023          cli();
1024          if (ip_rt_bh_mask && !ip_rt_lock)
1025          {
1026                  if (ip_rt_bh_mask & RT_BH_REDIRECT)
1027                          rt_kick_backlog();
1028
1029                  if (ip_rt_bh_mask & RT_BH_GARBAGE_COLLECT)
1030                  {
1031                          ip_rt_fast_lock();
1032                          ip_rt_bh_mask &= ~RT_BH_GARBAGE_COLLECT;
1033                          sti();
1034                          rt_garbage_collect_1();
1035                          cli();
1036                          ip_rt_fast_unlock();
1037                  }
1038
1039                  if (ip_rt_bh_mask & RT_BH_FREE)
1040                          rt_kick_free_queue();
1041          }
1042          restore_flags(flags);
1043  }
1044
1045
1046  void ip_rt_check_expire()
1047  {
1048          ip_rt_fast_lock();
1049          if (ip_rt_lock == 1)
1050          {
1051                  int i;
1052                  struct rtable *rth, **rthp;
1053                  unsigned long flags;
1054                  unsigned long now = jiffies;
1055
1056                  save_flags(flags);
1057                  for (i=0; i<RT_HASH_DIVISOR; i++)
1058                  {
1059                          rthp = &ip_rt_hash_table[i];
1060
1061                          while ((rth = *rthp) != NULL)
1062                          {
1063                                  struct rtable * rth_next = rth->rt_next;
1064
1065                                  /*
1066                                   * Cleanup aged off entries.
1067                                   */
1068
1069                                  cli();
1070                                  if (!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT < now)
```

```
1071                                    {
1072                                            *rthp = rth_next;
1073                                            sti();
1074                                            rt_cache_size--;
1075    #if RT_CACHE_DEBUG >= 2
1076                                            printk("rt_check_expire clean %02x@%08x\n", i, rth->rt_dst);
1077    #endif
1078                                            rt_free(rth);
1079                                            continue;
1080                                    }
1081                                    sti();
1082
1083                                    if (!rth_next)
1084                                            break;
1085
1086                                    /*
1087                                     * LRU ordering.
1088                                     */
1089
1090                                    if (rth->rt_lastuse + RT_CACHE_BUBBLE_THRESHOULD < rth_next-
1091    >rt_lastuse ||
1092                                       (rth->rt_lastuse < rth_next->rt_lastuse &&
1093                                        rth->rt_use < rth_next->rt_use))
1094                                    {
1095    #if RT_CACHE_DEBUG >= 2
1096                                            printk("rt_check_expire bubbled %02x@%08x<->%08x\n", i, rth-
1097    >rt_dst, rth_next->rt_dst);
1098    #endif
1099                                            cli();
1100                                            *rthp = rth_next;
1101                                            rth->rt_next = rth_next->rt_next;
1102                                            rth_next->rt_next = rth;
1103                                            sti();
1104                                            rthp = &rth_next->rt_next;
1105                                            continue;
1106                                    }
1107                                    rthp = &rth->rt_next;
1108                            }
1109                    }
1110                    restore_flags(flags);
1111                    rt_kick_free_queue();
1112            }
1113            ip_rt_unlock();
1114    }
1115
1116    static void rt_redirect_1(__u32 dst, __u32 gw, struct device *dev)
1117    {
1118            struct rtable *rt;
1119            unsigned long hash = ip_rt_hash_code(dst);
1120
1121            if (gw == dev->pa_addr)
```

```
1122                    return;
1123            if (dev != get_gw_dev(gw))
1124                    return;
1125            rt = (struct rtable *) kmalloc(sizeof(struct rtable), GFP_ATOMIC);
1126            if (rt == NULL)
1127                    return;
1128            memset(rt, 0, sizeof(struct rtable));
1129            rt->rt_flags = RTF_DYNAMIC | RTF_MODIFIED | RTF_HOST | RTF_GATEWAY | RTF_UP;
1130            rt->rt_dst = dst;
1131            rt->rt_dev = dev;
1132            rt->rt_gateway = gw;
1133            rt->rt_src = dev->pa_addr;
1134            rt->rt_mtu = dev->mtu;
1135    #ifdef CONFIG_NO_PATH_MTU_DISCOVERY
1136            if (dev->mtu > 576)
1137                    rt->rt_mtu = 576;
1138    #endif
1139            rt->rt_lastuse  = jiffies;
1140            rt->rt_refcnt  = 1;
1141            rt_cache_add(hash, rt);
1142            ip_rt_put(rt);
1143            return;
1144    }
1145
1146    static void rt_cache_flush(void)
1147    {
1148            int i;
1149            struct rtable * rth, * next;
1150
1151            for (i=0; i<RT_HASH_DIVISOR; i++)
1152            {
1153                    int nr=0;
1154
1155                    cli();
1156                    if (!(rth = ip_rt_hash_table[i]))
1157                    {
1158                            sti();
1159                            continue;
1160                    }
1161
1162                    ip_rt_hash_table[i] = NULL;
1163                    sti();
1164
1165                    for (; rth; rth=next)
1166                    {
1167                            next = rth->rt_next;
1168                            rt_cache_size--;
1169                            nr++;
1170                            rth->rt_next = NULL;
1171                            rt_free(rth);
1172                    }
```

```
1173    #if RT_CACHE_DEBUG >= 2
1174                    if (nr > 0)
1175                            printk("rt_cache_flush: %d@%02x\n", nr, i);
1176    #endif
1177            }
1178    #if RT_CACHE_DEBUG >= 1
1179            if (rt_cache_size)
1180            {
1181                    printk("rt_cache_flush: bug rt_cache_size=%d\n", rt_cache_size);
1182                    rt_cache_size = 0;
1183            }
1184    #endif
1185    }
1186
1187    static void rt_garbage_collect_1(void)
1188    {
1189            int i;
1190            unsigned expire = RT_CACHE_TIMEOUT>>1;
1191            struct rtable * rth, **rthp;
1192            unsigned long now = jiffies;
1193
1194            for (;;)
1195            {
1196                    for (i=0; i<RT_HASH_DIVISOR; i++)
1197                    {
1198                            if (!ip_rt_hash_table[i])
1199                                    continue;
1200                            for (rthp=&ip_rt_hash_table[i]; (rth=*rthp); rthp=&rth->rt_next)
1201                            {
1202                                    if (rth->rt_lastuse + expire*(rth->rt_refcnt+1) > now)
1203                                            continue;
1204                                    rt_cache_size--;
1205                                    cli();
1206                                    *rthp=rth->rt_next;
1207                                    rth->rt_next = NULL;
1208                                    sti();
1209                                    rt_free(rth);
1210                                    break;
1211                            }
1212                    }
1213                    if (rt_cache_size < RT_CACHE_SIZE_MAX)
1214                            return;
1215                    expire >>= 1;
1216            }
1217    }
1218
1219    static __inline__ void rt_req_enqueue(struct rt_req **q, struct rt_req *rtr)
1220    {
1221            unsigned long flags;
1222            struct rt_req * tail;
1223
```

```
1224            save_flags(flags);
1225            cli();
1226            tail = *q;
1227            if (!tail)
1228                    rtr->rtr_next = rtr;
1229            else
1230            {
1231                    rtr->rtr_next = tail->rtr_next;
1232                    tail->rtr_next = rtr;
1233            }
1234            *q = rtr;
1235            restore_flags(flags);
1236            return;
1237    }
1238
1239    /*
1240     * Caller should mask interrupts.
1241     */
1242
1243    static __inline__ struct rt_req * rt_req_dequeue(struct rt_req **q)
1244    {
1245            struct rt_req * rtr;
1246
1247            if (*q)
1248            {
1249                    rtr = (*q)->rtr_next;
1250                    (*q)->rtr_next = rtr->rtr_next;
1251                    if (rtr->rtr_next == rtr)
1252                            *q = NULL;
1253                    rtr->rtr_next = NULL;
1254                    return rtr;
1255            }
1256            return NULL;
1257    }
1258
1259    /*
1260      Called with masked interrupts
1261     */
1262
1263    static void rt_kick_backlog()
1264    {
1265            if (!ip_rt_lock)
1266            {
1267                    struct rt_req * rtr;
1268
1269                    ip_rt_fast_lock();
1270
1271                    while ((rtr = rt_req_dequeue(&rt_backlog)) != NULL)
1272                    {
1273                            sti();
1274                            rt_redirect_1(rtr->dst, rtr->gw, rtr->dev);
```

```
1275                            kfree_s(rtr, sizeof(struct rt_req));
1276                            cli();
1277                    }
1278
1279                    ip_rt_bh_mask &= ~RT_BH_REDIRECT;
1280
1281                    ip_rt_fast_unlock();
1282            }
1283    }
1284
1285    /*
1286     * rt_{del|add|flush} called only from USER process. Waiting is OK.
1287     */
1288
1289    static int rt_del(__u32 dst, __u32 mask,
1290                    struct device * dev, __u32 gtw, short rt_flags, short metric)
1291    {
1292            int retval;
1293
1294            while (ip_rt_lock)
1295                    sleep_on(&rt_wait);
1296            ip_rt_fast_lock();
1297            retval = fib_del_1(dst, mask, dev, gtw, rt_flags, metric);
1298            ip_rt_unlock();
1299            wake_up(&rt_wait);
1300            return retval;
1301    }
1302
1303    static void rt_add(short flags, __u32 dst, __u32 mask,
1304            __u32 gw, struct device *dev, unsigned short mss,
1305            unsigned long window, unsigned short irtt, short metric)
1306    {
1307            while (ip_rt_lock)
1308                    sleep_on(&rt_wait);
1309            ip_rt_fast_lock();
1310            fib_add_1(flags, dst, mask, gw, dev, mss, window, irtt, metric);
1311            ip_rt_unlock();
1312            wake_up(&rt_wait);
1313    }
1314
1315    void ip_rt_flush(struct device *dev)
1316    {
1317            while (ip_rt_lock)
1318                    sleep_on(&rt_wait);
1319            ip_rt_fast_lock();
1320            fib_flush_1(dev);
1321            ip_rt_unlock();
1322            wake_up(&rt_wait);
1323    }
1324
1325    /*
```

```
1326      Called by ICMP module.
1327     */
1328
1329    void ip_rt_redirect(__u32 src, __u32 dst, __u32 gw, struct device *dev)
1330    {
1331            struct rt_req * rtr;
1332            struct rtable * rt;
1333
1334            rt = ip_rt_route(dst, 0);
1335            if (!rt)
1336                    return;
1337
1338            if (rt->rt_gateway != src ||
1339               rt->rt_dev != dev ||
1340               ((gw^dev->pa_addr)&dev->pa_mask) ||
1341               ip_chk_addr(gw))
1342            {
1343                    ip_rt_put(rt);
1344                    return;
1345            }
1346            ip_rt_put(rt);
1347
1348            ip_rt_fast_lock();
1349            if (ip_rt_lock == 1)
1350            {
1351                    rt_redirect_1(dst, gw, dev);
1352                    ip_rt_unlock();
1353                    return;
1354            }
1355
1356            rtr = kmalloc(sizeof(struct rt_req), GFP_ATOMIC);
1357            if (rtr)
1358            {
1359                    rtr->dst = dst;
1360                    rtr->gw = gw;
1361                    rtr->dev = dev;
1362                    rt_req_enqueue(&rt_backlog, rtr);
1363                    ip_rt_bh_mask |= RT_BH_REDIRECT;
1364            }
1365            ip_rt_unlock();
1366    }
1367
1368
1369    static __inline__ void rt_garbage_collect(void)
1370    {
1371            if (ip_rt_lock == 1)
1372            {
1373                    rt_garbage_collect_1();
1374                    return;
1375            }
1376            ip_rt_bh_mask |= RT_BH_GARBAGE_COLLECT;
```

```
1377    }
1378
1379    static void rt_cache_add(unsigned hash, struct rtable * rth)
1380    {
1381            unsigned long   flags;
1382            struct rtable   **rthp;
1383            __u32           daddr = rth->rt_dst;
1384            unsigned long   now = jiffies;
1385
1386    #if RT_CACHE_DEBUG >= 2
1387            if (ip_rt_lock != 1)
1388            {
1389                    printk("rt_cache_add: ip_rt_lock==%d\n", ip_rt_lock);
1390                    return;
1391            }
1392    #endif
1393
1394            save_flags(flags);
1395
1396            if (rth->rt_dev->header_cache_bind)
1397            {
1398                    struct rtable * rtg = rth;
1399
1400                    if (rth->rt_gateway != daddr)
1401                    {
1402                            ip_rt_fast_unlock();
1403                            rtg = ip_rt_route(rth->rt_gateway, 0);
1404                            ip_rt_fast_lock();
1405                    }
1406
1407                    if (rtg)
1408                    {
1409                            if (rtg == rth)
1410                                    rtg->rt_dev->header_cache_bind(&rtg->rt_hh, rtg->rt_dev, ETH_P_IP, rtg-
1411    >rt_dst);
1412                            else
1413                            {
1414                                    if (rtg->rt_hh)
1415                                            ATOMIC_INCR(&rtg->rt_hh->hh_refcnt);
1416                                    rth->rt_hh = rtg->rt_hh;
1417                                    ip_rt_put(rtg);
1418                            }
1419                    }
1420            }
1421
1422            if (rt_cache_size >= RT_CACHE_SIZE_MAX)
1423                    rt_garbage_collect();
1424
1425            cli();
1426            rth->rt_next = ip_rt_hash_table[hash];
1427    #if RT_CACHE_DEBUG >= 2
```

```
1428            if (rth->rt_next)
1429            {
1430                    struct rtable * trth;
1431                    printk("rt_cache @%02x: %08x", hash, daddr);
1432                    for (trth=rth->rt_next; trth; trth=trth->rt_next)
1433                            printk(" . %08x", trth->rt_dst);
1434                    printk("\n");
1435            }
1436    #endif
1437            ip_rt_hash_table[hash] = rth;
1438            rthp = &rth->rt_next;
1439            sti();
1440            rt_cache_size++;
1441
1442            /*
1443             * Clcanup duplicatc (and agcd off) cntrics.
1444             */
1445
1446            while ((rth = *rthp) != NULL)
1447            {
1448
1449                    cli();
1450                    if ((!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT < now)
1451                       || rth->rt_dst == daddr)
1452                    {
1453                            *rthp = rth->rt_next;
1454                            rt_cache_size--;
1455                            sti();
1456    #if RT_CACHE_DEBUG >= 2
1457                            printk("rt_cache clean %02x@%08x\n", hash, rth->rt_dst);
1458    #endif
1459                            rt_free(rth);
1460                            continue;
1461                    }
1462                    sti();
1463                    rthp = &rth->rt_next;
1464            }
1465            restore_flags(flags);
1466    }
1467
1468    /*
1469      RT should be already locked.
1470
1471      We could improve this by keeping a chain of say 32 struct rtable's
1472      last freed for fast recycling.
1473
1474     */
1475
1476    struct rtable * ip_rt_slow_route (__u32 daddr, int local)
1477    {
1478            unsigned hash = ip_rt_hash_code(daddr)^local;
```

```
1479            struct rtable * rth;
1480            struct fib_node * f;
1481            struct fib_info * fi;
1482            __u32 saddr;
1483
1484    #if RT_CACHE_DEBUG >= 2
1485            printk("rt_cache miss @%08x\n", daddr);
1486    #endif
1487
1488            rth = kmalloc(sizeof(struct rtable), GFP_ATOMIC);
1489            if (!rth)
1490            {
1491                    ip_rt_unlock();
1492                    return NULL;
1493            }
1494
1495            if (local)
1496                    f = fib_lookup_local(daddr);
1497            else
1498                    f = fib_lookup (daddr);
1499
1500            if (f)
1501            {
1502                    fi = f->fib_info;
1503                    f->fib_use++;
1504            }
1505
1506            if (!f || (fi->fib_flags & RTF_REJECT))
1507            {
1508    #if RT_CACHE_DEBUG >= 2
1509                    printk("rt_route failed @%08x\n", daddr);
1510    #endif
1511                    ip_rt_unlock();
1512                    kfree_s(rth, sizeof(struct rtable));
1513                    return NULL;
1514            }
1515
1516            saddr = fi->fib_dev->pa_addr;
1517
1518            if (daddr == fi->fib_dev->pa_addr)
1519            {
1520                    f->fib_use--;
1521                    if ((f = fib_loopback) != NULL)
1522                    {
1523                            f->fib_use++;
1524                            fi = f->fib_info;
1525                    }
1526            }
1527
1528            if (!f)
1529            {
```

```
1530                    ip_rt_unlock();
1531                    kfree_s(rth, sizeof(struct rtable));
1532                    return NULL;
1533            }
1534
1535            rth->rt_dst     = daddr;
1536            rth->rt_src     = saddr;
1537            rth->rt_lastuse = jiffies;
1538            rth->rt_refcnt  = 1;
1539            rth->rt_use     = 1;
1540            rth->rt_next    = NULL;
1541            rth->rt_hh      = NULL;
1542            rth->rt_gateway = fi->fib_gateway;
1543            rth->rt_dev     = fi->fib_dev;
1544            rth->rt_mtu     = fi->fib_mtu;
1545            rth->rt_window  = fi->fib_window;
1546            rth->rt_irtt    = fi->fib_irtt;
1547            rth->rt_tos     = f->fib_tos;
1548            rth->rt_flags   = fi->fib_flags | RTF_HOST;
1549            if (local)
1550                    rth->rt_flags   |= RTF_LOCAL;
1551
1552            if (!(rth->rt_flags & RTF_GATEWAY))
1553                    rth->rt_gateway = rth->rt_dst;
1554
1555            if (ip_rt_lock == 1)
1556                    rt_cache_add(hash, rth);
1557            else
1558            {
1559                    rt_free(rth);
1560    #if RT_CACHE_DEBUG >= 1
1561                    printk("rt_cache: route to %08x was born dead\n", daddr);
1562    #endif
1563            }
1564
1565            ip_rt_unlock();
1566            return rth;
1567    }
1568
1569    void ip_rt_put(struct rtable * rt)
1570    {
1571            if (rt)
1572                    ATOMIC_DECR(&rt->rt_refcnt);
1573    }
1574
1575    struct rtable * ip_rt_route(__u32 daddr, int local)
1576    {
1577            struct rtable * rth;
1578
1579            ip_rt_fast_lock();
1580
```

```
1581            for (rth=ip_rt_hash_table[ip_rt_hash_code(daddr)^local]; rth; rth=rth->rt_next)
1582            {
1583                    if (rth->rt_dst == daddr)
1584                    {
1585                            rth->rt_lastuse = jiffies;
1586                            ATOMIC_INCR(&rth->rt_use);
1587                            ATOMIC_INCR(&rth->rt_refcnt);
1588                            ip_rt_unlock();
1589                            return rth;
1590                    }
1591            }
1592            return ip_rt_slow_route (daddr, local);
1593    }
1594
1595
1596    /*
1597     *      Process a route add request from the user, or from a kernel
1598     *      task.
1599     */
1600
1601    int ip_rt_new(struct rtentry *r)
1602    {
1603            int err;
1604            char * devname;
1605            struct device * dev = NULL;
1606            unsigned long flags;
1607            __u32 daddr, mask, gw;
1608            short metric;
1609
1610            /*
1611             *      If a device is specified find it.
1612             */
1613
1614            if ((devname = r->rt_dev) != NULL)
1615            {
1616                    err = getname(devname, &devname);
1617                    if (err)
1618                            return err;
1619                    dev = dev_get(devname);
1620                    putname(devname);
1621                    if (!dev)
1622                            return -ENODEV;
1623            }
1624
1625            /*
1626             *      If the device isn't INET, don't allow it
1627             */
1628
1629            if (r->rt_dst.sa_family != AF_INET)
1630                    return -EAFNOSUPPORT;
1631
```

```
1632            /*
1633             *      Make local copies of the important bits
1634             *      We decrement the metric by one for BSD compatibility.
1635             */
1636
1637            flags = r->rt_flags;
1638            daddr = (__u32) ((struct sockaddr_in *) &r->rt_dst)->sin_addr.s_addr;
1639            mask  = (__u32) ((struct sockaddr_in *) &r->rt_genmask)->sin_addr.s_addr;
1640            gw    = (__u32) ((struct sockaddr_in *) &r->rt_gateway)->sin_addr.s_addr;
1641            metric = r->rt_metric > 0 ? r->rt_metric - 1 : 0;
1642
1643            /*
1644             *      BSD emulation: Permits route add someroute gw one-of-my-addresses
1645             *      to indicate which iface. Not as clean as the nice Linux dev technique
1646             *      but people keep using it...  (and gated likes it ;))
1647             */
1648
1649            if (!dev && (flags & RTF_GATEWAY))
1650            {
1651                    struct device *dev2;
1652                    for (dev2 = dev_base ; dev2 != NULL ; dev2 = dev2->next)
1653                    {
1654                            if ((dev2->flags & IFF_UP) && dev2->pa_addr == gw)
1655                            {
1656                                    flags &= ~RTF_GATEWAY;
1657                                    dev = dev2;
1658                                    break;
1659                            }
1660                    }
1661            }
1662
1663            /*
1664             *      Ignore faulty masks
1665             */
1666
1667            if (bad_mask(mask, daddr))
1668                    mask=0;
1669
1670            /*
1671             *      Set the mask to nothing for host routes.
1672             */
1673
1674            if (flags & RTF_HOST)
1675                    mask = 0xffffffff;
1676            else if (mask && r->rt_genmask.sa_family != AF_INET)
1677                    return -EAFNOSUPPORT;
1678
1679            /*
1680             *      You can only gateway IP via IP..
1681             */
1682
```

```
1683            if (flags & RTF_GATEWAY)
1684            {
1685                    if (r->rt_gateway.sa_family != AF_INET)
1686                            return -EAFNOSUPPORT;
1687                    if (!dev)
1688                            dev = get_gw_dev(gw);
1689            }
1690            else if (!dev)
1691                    dev = ip_dev_check(daddr);
1692
1693            /*
1694             *      Unknown device.
1695             */
1696
1697            if (dev == NULL)
1698                    return -ENETUNREACH;
1699
1700            /*
1701             *      Add the route
1702             */
1703
1704            rt_add(flags, daddr, mask, gw, dev, r->rt_mss, r->rt_window, r->rt_irtt, metric);
1705            return 0;
1706    }
1707
1708
1709    /*
1710     *      Remove a route, as requested by the user.
1711     */
1712
1713    static int rt_kill(struct rtentry *r)
1714    {
1715            struct sockaddr_in *trg;
1716            struct sockaddr_in *msk;
1717            struct sockaddr_in *gtw;
1718            char *devname;
1719            int err;
1720            struct device * dev = NULL;
1721
1722            trg = (struct sockaddr_in *) &r->rt_dst;
1723            msk = (struct sockaddr_in *) &r->rt_genmask;
1724            gtw = (struct sockaddr_in *) &r->rt_gateway;
1725            if ((devname = r->rt_dev) != NULL)
1726            {
1727                    err = getname(devname, &devname);
1728                    if (err)
1729                            return err;
1730                    dev = dev_get(devname);
1731                    putname(devname);
1732                    if (!dev)
1733                            return -ENODEV;
```

```
1734            }
1735            /*
1736             * metric can become negative here if it wasn't filled in
1737             * but that's a fortunate accident; we really use that in rt_del.
1738             */
1739            err=rt_del((__u32)trg->sin_addr.s_addr, (__u32)msk->sin_addr.s_addr, dev,
1740                    (__u32)gtw->sin_addr.s_addr, r->rt_flags, r->rt_metric - 1);
1741            return err;
1742    }
1743
1744    /*
1745     *      Handle IP routing ioctl calls. These are used to manipulate the routing tables
1746     */
1747
1748    int ip_rt_ioctl(unsigned int cmd, void *arg)
1749    {
1750            int err;
1751            struct rtentry rt;
1752
1753            switch(cmd)
1754            {
1755                    case SIOCADDRT:                 /* Add a route */
1756                    case SIOCDELRT:                 /* Delete a route */
1757                            if (!suser())
1758                                    return -EPERM;
1759                            err=verify_area(VERIFY_READ, arg, sizeof(struct rtentry));
1760                            if (err)
1761                                    return err;
1762                            memcpy_fromfs(&rt, arg, sizeof(struct rtentry));
1763                            return (cmd == SIOCDELRT) ? rt_kill(&rt) : ip_rt_new(&rt);
1764            }
1765
1766            return -EINVAL;
1767    }
1768
1769    void ip_rt_advice(struct rtable **rp, int advice)
1770    {
1771            /* Thanks! */
1772            return;
1773    }
1774
```

```
/*
 * INET           An implementation of the TCP/IP protocol suite for the
LINUX
 *          operating system.  INET is implemented using the  BSD Socket
 *          interface as the means of communication with the user level.
 *
 *          ROUTE - implementation of the IP router.
 *
 * Version: @(#)route.c 1.0.14      05/31/93
 *
 * Authors: Ross Biro, <bir7@leland.Stanford.Edu>
 *          Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *          Alan Cox, <gw4pts@gw4pts.ampr.org>
 *          Linus Torvalds, <Linus.Torvalds@helsinki.fi>
 *
 * Fixes:
 *          Alan Cox    :     Verify area fixes.
 *          Alan Cox    :     cli() protects routing changes
 *          Rui Oliveira      :     ICMP routing table updates
 *          (rco@di.uminho.pt)      Routing table insertion and update
 *          Linus Torvalds    :     Rewrote bits to be sensible
 *          Alan Cox    :     Added BSD route gw semantics
 *          Alan Cox    :     Super /proc >4K
 *          Alan Cox    :     MTU in route table
 *          Alan Cox    :     MSS actually. Also added the window
 *                            clamper.
 *          Sam Lantinga      :     Fixed route matching in rt_del()
 *          Alan Cox    :     Routing cache support.
 *          Alan Cox    :     Removed compatibility cruft.
 *          Alan Cox    :     RTF_REJECT support.
 *          Alan Cox    :     TCP irtt support.
 *          Jonathan Naylor   :     Added Metric support.
 *    Miquel van Smoorenburg  :     BSD API fixes.
 *    Miquel van Smoorenburg  :     Metrics.
 *          Alan Cox    :     Use __u32 properly
 *          Alan Cox    :     Aligned routing errors more closely with
BSD
 *                            our system is still very different.
 *          Alan Cox    :     Faster /proc handling
 *    Alexey Kuznetsov  :     Massive rework to support tree based
routing,
 *                            routing caches and better behaviour.
 *
 *          Olaf Erb    :     irtt wasnt being copied right.
 *
 *          This program is free software; you can redistribute it
and/or
 *          modify it under the terms of the GNU General Public License
 *          as published by the Free Software Foundation; either version
 *          2 of the License, or (at your option) any later version.
 */

#include <linux/config.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <asm/bitops.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/string.h>
```

```
#include <linux/socket.h>
#include <linux/sockios.h>
#include <linux/errno.h>
#include <linux/in.h>
#include <linux/inet.h>
#include <linux/netdevice.h>
#include <net/ip.h>
#include <net/protocol.h>
#include <net/route.h>
#include <net/tcp.h>
#include <linux/skbuff.h>
#include <net/sock.h>
#include <net/icmp.h>
#include <net/netlink.h>

/*
 * Forwarding Information Base definitions.
 */

struct fib_node
{
      struct fib_node         *fib_next;
      __u32             fib_dst;
      unsigned long           fib_use;
      struct fib_info         *fib_info;
      short             fib_metric;
      unsigned char           fib_tos;
};

/*
 * This structure contains data shared by many of routes.
 */

struct fib_info
{
      struct fib_info         *fib_next;
      struct fib_info         *fib_prev;
      __u32             fib_gateway;
      struct device           *fib_dev;
      int               fib_refcnt;
      unsigned long           fib_window;
      unsigned short          fib_flags;
      unsigned short          fib_mtu;
      unsigned short          fib_irtt;
};

struct fib_zone
{
      struct fib_zone   *fz_next;
      struct fib_node   **fz_hash_table;
      struct fib_node   *fz_list;
      int         fz_nent;
      int         fz_logmask;
      __u32       fz_mask;
};

static struct fib_zone  *fib_zones[33];
static struct fib_zone  *fib_zone_list;
static struct fib_node  *fib_loopback = NULL;
static struct fib_info  *fib_info_list;
```

```
/*
 * Backlogging.
 */

#define RT_BH_REDIRECT        0
#define RT_BH_GARBAGE_COLLECT       1
#define RT_BH_FREE            2

struct rt_req
{
      struct rt_req * rtr_next;
      struct device *dev;
      __u32 dst;
      __u32 gw;
      unsigned char tos;
};

int               ip_rt_lock;
unsigned          ip_rt_bh_mask;
static struct rt_req    *rt_backlog;

/*
 * Route cache.
 */

struct rtable           *ip_rt_hash_table[RT_HASH_DIVISOR];
static int        rt_cache_size;
static struct rtable    *rt_free_queue;
struct wait_queue *rt_wait;

static void rt_kick_backlog(void);
static void rt_cache_add(unsigned hash, struct rtable * rth);
static void rt_cache_flush(void);
static void rt_garbage_collect_1(void);

/*
 * Evaluate mask length.
 */

static __inline__ int rt_logmask(__u32 mask)
{
      if (!(mask = ntohl(mask)))
            return 32;
      return ffz(~mask);
}

/*
 * Create mask from length.
 */

static __inline__ __u32 rt_mask(int logmask)
{
      if (logmask >= 32)
            return 0;
      return htonl(~((1<<logmask)-1));
}

static __inline__ unsigned fz_hash_code(__u32 dst, int logmask)
{
      return ip_rt_hash_code(ntohl(dst)>>logmask);
}
```

```
/*
 * Free FIB node.
 */

static void fib_free_node(struct fib_node * f)
{
      struct fib_info * fi = f->fib_info;
      if (!--fi->fib_refcnt)
      {
#if RT_CACHE_DEBUG >= 2
            printk("fib_free_node: fi %08x/%s is free\n", fi->
fib_gateway, fi->fib_dev->name);
#endif
            if (fi->fib_next)
                  fi->fib_next->fib_prev = fi->fib_prev;
            if (fi->fib_prev)
                  fi->fib_prev->fib_next = fi->fib_next;
            if (fi == fib_info_list)
                  fib_info_list = fi->fib_next;
      }
      kfree_s(f, sizeof(struct fib_node));
}

/*
 * Find gateway route by address.
 */

static struct fib_node * fib_lookup_gateway(__u32 dst)
{
      struct fib_zone * fz;
      struct fib_node * f;

      for (fz = fib_zone_list; fz; fz = fz->fz_next)
      {
            if (fz->fz_hash_table)
                  f = fz->fz_hash_table[fz_hash_code(dst, fz->
fz_logmask)];
            else
                  f = fz->fz_list;

            for ( ; f; f = f->fib_next)
            {
                  if ((dst ^ f->fib_dst) & fz->fz_mask)
                        continue;
                  if (f->fib_info->fib_flags & RTF_GATEWAY)
                        return NULL;
                  return f;
            }
      }
      return NULL;
}

/*
 * Find local route by address.
 * FIXME: I use "longest match" principle. If destination
 *      has some non-local route, I'll not search shorter matches.
 *      It's possible, I'm wrong, but I wanted to prevent following
 *      situation:
 *    route add 193.233.7.128 netmask 255.255.255.192 gw xxxxxx
 *    route add 193.233.7.0   netmask 255.255.255.0 eth1
```

```
 *      (Two ethernets connected by serial line, one is small and other
is large)
 *      Host 193.233.7.129 is locally unreachable,
 *      but old (<=1.3.37) code will send packets destined for it to
eth1.
 *
 */

static struct fib_node * fib_lookup_local(__u32 dst)
{
      struct fib_zone * fz;
      struct fib_node * f;

      for (fz = fib_zone_list; fz; fz = fz->fz_next)
      {
            int longest_match_found = 0;

            if (fz->fz_hash_table)
                  f = fz->fz_hash_table[fz_hash_code(dst, fz->
fz_logmask)];
            else
                  f = fz->fz_list;

            for ( ; f; f = f->fib_next)
            {
                  if ((dst ^ f->fib_dst) & fz->fz_mask)
                        continue;
                  if (!(f->fib_info->fib_flags & RTF_GATEWAY))
                        return f;
                  longest_match_found = 1;
            }
            if (longest_match_found)
                  return NULL;
      }
      return NULL;
}

/*
 * Main lookup routine.
 *    IMPORTANT NOTE: this algorithm has small difference from <=1.3.37
visible
 *    by user. It doesn't route non-CIDR broadcasts by default.
 *
 *    F.e.
 *          ifconfig eth0 193.233.7.65 netmask 255.255.255.192 broadcast
193.233.7.255
 *    is valid, but if you really are not able (not allowed, do not
want) to
 *    use CIDR compliant broadcast 193.233.7.127, you should add host
route:
 *          route add -host 193.233.7.255 eth0
 */

static struct fib_node * fib_lookup(__u32 dst)
{
      struct fib_zone * fz;
      struct fib_node * f;

      for (fz = fib_zone_list; fz; fz = fz->fz_next)
      {
            if (fz->fz_hash_table)
```

```
                  f = fz->fz_hash_table[fz_hash_code(dst, fz->
fz_logmask)];
            else
                  f = fz->fz_list;

            for ( ; f; f = f->fib_next)
            {
                  if ((dst ^ f->fib_dst) & fz->fz_mask)
                        continue;
                  return f;
            }
      }
      return NULL;
}

static __inline__ struct device * get_gw_dev(__u32 gw)
{
      struct fib_node * f;
      f = fib_lookup_gateway(gw);
      if (f)
            return f->fib_info->fib_dev;
      return NULL;
}

/*
 *    Used by 'rt_add()' when we can't get the netmask any other way..
 *
 *    If the lower byte or two are zero, we guess the mask based on the
 *    number of zero 8-bit net numbers, otherwise we use the "default"
 *    masks judging by the destination address and our device netmask.
 */

static __u32 unsigned long default_mask(__u32 dst)
{
      dst = ntohl(dst);
      if (IN_CLASSA(dst))
            return htonl(IN_CLASSA_NET);
      if (IN_CLASSB(dst))
            return htonl(IN_CLASSB_NET);
      return htonl(IN_CLASSC_NET);
}


/*
 *    If no mask is specified then generate a default entry.
 */

static __u32 guess_mask(__u32 dst, struct device * dev)
{
      __u32 mask;

      if (!dst)
            return 0;
      mask = default_mask(dst);
      if ((dst ^ dev->pa_addr) & mask)
            return mask;
      return dev->pa_mask;
}


/*
```

```
 *    Check if a mask is acceptable.
 */

static inline int bad_mask(__u32 mask, __u32 addr)
{
      if (addr & (mask = ~mask))
            return 1;
      mask = ntohl(mask);
      if (mask & (mask+1))
            return 1;
      return 0;
}


static int fib_del_list(struct fib_node **fp, __u32 dst,
            struct device * dev, __u32 gtw, short flags, short metric,
__u32 mask)
{
      struct fib_node *f;
      int found=0;

      while((f = *fp) != NULL)
      {
            struct fib_info * fi = f->fib_info;

            /*
             *    Make sure the destination and netmask match.
             *    metric, gateway and device are also checked
             *    if they were specified.
             */
            if (f->fib_dst != dst ||
                (gtw && fi->fib_gateway != gtw) ||
                (metric >= 0 && f->fib_metric != metric) ||
                (dev && fi->fib_dev != dev) )
            {
                  fp = &f->fib_next;
                  continue;
            }
            cli();
            *fp = f->fib_next;
            if (fib_loopback == f)
                  fib_loopback = NULL;
            sti();
            ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags,
metric, fi->fib_dev->name);
            fib_free_node(f);
            found++;
      }
      return found;
}

static __inline__ int fib_del_1(__u32 dst, __u32 mask,
            struct device * dev, __u32 gtw, short flags, short metric)
{
      struct fib_node **fp;
      struct fib_zone *fz;
      int found=0;

      if (!mask)
      {
            for (fz=fib_zone_list; fz; fz = fz->fz_next)
```

```
            {
                  int tmp;
                  if (fz->fz_hash_table)
                        fp = &fz->fz_hash_table[fz_hash_code(dst, fz->
fz_logmask)];
                  else
                        fp = &fz->fz_list;

                  tmp = fib_del_list(fp, dst, dev, gtw, flags, metric,
mask);
                  fz->fz_nent -= tmp;
                  found += tmp;
            }
      }
      else
      {
            if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
            {
                  if (fz->fz_hash_table)
                        fp = &fz->fz_hash_table[fz_hash_code(dst, fz->
fz_logmask)];
                  else
                        fp = &fz->fz_list;

                  found = fib_del_list(fp, dst, dev, gtw, flags, metric,
mask);
                  fz->fz_nent -= found;
            }
      }

      if (found)
      {
            rt_cache_flush();
            return 0;
      }
      return -ESRCH;
}


static struct fib_info * fib_create_info(__u32 gw, struct device * dev,
                                 unsigned short flags, unsigned short mss,
                                 unsigned long window, unsigned short
irtt)
{
      struct fib_info * fi;

      if (!(flags & RTF_MSS))
      {
            mss = dev->mtu;
#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
            /*
             *    If MTU was not specified, use default.
             *    If you want to increase MTU for some net (local
subnet)
             *    use "route add .... mss xxx".
             *
             *    The MTU isnt currently always used and computed as it
             *    should be as far as I can tell. [Still verifying this
is right]
             */
            if ((flags & RTF_GATEWAY) && mss > 576)
```

```
                  mss = 576;
#endif
      }
      if (!(flags & RTF_WINDOW))
            window = 0;
      if (!(flags & RTF_IRTT))
            irtt = 0;

      for (fi=fib_info_list; fi; fi = fi->fib_next)
      {
            if (fi->fib_gateway != gw ||
                fi->fib_dev != dev  ||
                fi->fib_flags != flags ||
                fi->fib_mtu != mss ||
                fi->fib_window != window ||
                fi->fib_irtt != irtt)
                  continue;
            fi->fib_refcnt++;
#if RT_CACHE_DEBUG >= 2
            printk("fib_create_info: fi %08x/%s is duplicate\n", fi->
fib_gateway, fi->fib_dev->name);
#endif
            return fi;
      }
      fi = (struct fib_info*)kmalloc(sizeof(struct fib_info),
GFP_KERNEL);
      if (!fi)
            return NULL;
      memset(fi, 0, sizeof(struct fib_info));
      fi->fib_flags = flags;
      fi->fib_dev = dev;
      fi->fib_gateway = gw;
      fi->fib_mtu = mss;
      fi->fib_window = window;
      fi->fib_refcnt++;
      fi->fib_next = fib_info_list;
      fi->fib_prev = NULL;
      fi->fib_irtt = irtt;
      if (fib_info_list)
            fib_info_list->fib_prev = fi;
      fib_info_list = fi;
#if RT_CACHE_DEBUG >= 2
      printk("fib_create_info: fi %08x/%s is created\n", fi->
fib_gateway, fi->fib_dev->name);
#endif
      return fi;
}


static __inline__ void fib_add_1(short flags, __u32 dst, __u32 mask,
      __u32 gw, struct device *dev, unsigned short mss,
      unsigned long window, unsigned short irtt, short metric)
{
      struct fib_node *f, *f1;
      struct fib_node **fp;
      struct fib_node **dup_fp = NULL;
      struct fib_zone * fz;
      struct fib_info * fi;
      int logmask;

      if (flags & RTF_HOST)
```

```
            mask = 0xffffffff;
      /*
       * If mask is not specified, try to guess it.
       */
      else if (!mask)
      {
            if (!((dst ^ dev->pa_addr) & dev->pa_mask))
            {
                  mask = dev->pa_mask;
                  flags &= ~RTF_GATEWAY;
                  if (flags & RTF_DYNAMIC)
                  {
                        printk("Dynamic route to my own net rejected
\n");
                        return;
                  }
            }
            else
                  mask = guess_mask(dst, dev);
            dst &= mask;
      }

      /*
       *    A gateway must be reachable and not a local address
       */

      if (gw == dev->pa_addr)
            flags &= ~RTF_GATEWAY;

      if (flags & RTF_GATEWAY)
      {
            /*
             *    Don't try to add a gateway we can't reach..
             */

            if (dev != get_gw_dev(gw))
                  return;

            flags |= RTF_GATEWAY;
      }
      else
            gw = 0;

      /*
       *    Allocate an entry and fill it in.
       */

      f = (struct fib_node *) kmalloc(sizeof(struct fib_node),
GFP_KERNEL);
      if (f == NULL)
            return;

      memset(f, 0, sizeof(struct fib_node));
      f->fib_dst = dst;
      f->fib_metric = metric;
      f->fib_tos    = 0;

      if  ((fi = fib_create_info(gw, dev, flags, mss, window, irtt)) ==
NULL)
      {
            kfree_s(f, sizeof(struct fib_node));
```

```
                return;
        }
        f->fib_info = fi;

        logmask = rt_logmask(mask);
        fz = fib_zones[logmask];


        if (!fz)
        {
                int i;
                fz = kmalloc(sizeof(struct fib_zone), GFP_KERNEL);
                if (!fz)
                {
                        fib_free_node(f);
                        return;
                }
                memset(fz, 0, sizeof(struct fib_zone));
                fz->fz_logmask = logmask;
                fz->fz_mask = mask;
                for (i=logmask-1; i>=0; i--)
                        if (fib_zones[i])
                                break;
                cli();
                if (i<0)
                {
                        fz->fz_next = fib_zone_list;
                        fib_zone_list = fz;
                }
                else
                {
                        fz->fz_next = fib_zones[i]->fz_next;
                        fib_zones[i]->fz_next = fz;
                }
                fib_zones[logmask] = fz;
                sti();
        }

        /*
         * If zone overgrows RTZ_HASHING_LIMIT, create hash table.
         */

        if (fz->fz_nent >= RTZ_HASHING_LIMIT && !fz->fz_hash_table &&
logmask<32)
        {
                struct fib_node ** ht;
#if RT_CACHE_DEBUG
                printk("fib_add_1: hashing for zone %d started\n", logmask);
#endif
                ht = kmalloc(RTZ_HASH_DIVISOR*sizeof(struct rtable*),
GFP_KERNEL);

                if (ht)
                {
                        memset(ht, 0, RTZ_HASH_DIVISOR*sizeof(struct
fib_node*));
                        cli();
                        f1 = fz->fz_list;
                        while (f1)
                        {
                                struct fib_node * next;
```

```
                        unsigned hash = fz_hash_code(f1->fib_dst,
logmask);
                        next = f1->fib_next;
                        f1->fib_next = ht[hash];
                        ht[hash] = f1;
                        f1 = next;
                  }
                  fz->fz_list = NULL;
                  fz->fz_hash_table = ht;
                  sti();
            }
      }

      if (fz->fz_hash_table)
            fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
      else
            fp = &fz->fz_list;

      /*
       * Scan list to find the first route with the same destination
       */
      while ((f1 = *fp) != NULL)
      {
            if (f1->fib_dst == dst)
                  break;
            fp = &f1->fib_next;
      }

      /*
       * Find route with the same destination and less (or equal)
metric.
       */
      while ((f1 = *fp) != NULL && f1->fib_dst == dst)
      {
            if (f1->fib_metric >= metric)
                  break;
            /*
             *    Record route with the same destination and gateway,
             *    but less metric. We'll delete it
             *    after instantiation of new route.
             */
            if (f1->fib_info->fib_gateway == gw)
                  dup_fp = fp;
            fp = &f1->fib_next;
      }

      /*
       * Is it already present?
       */

      if (f1 && f1->fib_metric == metric && f1->fib_info == fi)
      {
            fib_free_node(f);
            return;
      }

      /*
       * Insert new entry to the list.
       */

      cli();
```

```
	f->fib_next = f1;
	*fp = f;
	if (!fib_loopback && (fi->fib_dev->flags & IFF_LOOPBACK))
		fib_loopback = f;
	sti();
	fz->fz_nent++;
	ip_netlink_msg(RTMSG_NEWROUTE, dst, gw, mask, flags, metric, fi->
fib_dev->name);

	/*
	 *	Delete route with the same destination and gateway.
	 *	Note that we should have at most one such route.
	 */
	if (dup_fp)
		fp = dup_fp;
	else
		fp = &f->fib_next;

	while ((f1 = *fp) != NULL && f1->fib_dst == dst)
	{
		if (f1->fib_info->fib_gateway == gw)
		{
			cli();
			*fp = f1->fib_next;
			if (fib_loopback == f1)
				fib_loopback = NULL;
			sti();
			ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags,
metric, f1->fib_info->fib_dev->name);
			fib_free_node(f1);
			fz->fz_nent--;
			break;
		}
		fp = &f1->fib_next;
	}
	rt_cache_flush();
	return;
}

static int rt_flush_list(struct fib_node ** fp, struct device *dev)
{
	int found = 0;
	struct fib_node *f;

	while ((f = *fp) != NULL) {
		if (f->fib_info->fib_dev != dev) {
			fp = &f->fib_next;
			continue;
		}
		cli();
		*fp = f->fib_next;
		if (fib_loopback == f)
			fib_loopback = NULL;
		sti();
		fib_free_node(f);
		found++;
	}
	return found;
}

static __inline__ void fib_flush_1(struct device *dev)
```

```
{
      struct fib_zone *fz;
      int found = 0;

      for (fz = fib_zone_list; fz; fz = fz->fz_next)
      {
            if (fz->fz_hash_table)
            {
                  int i;
                  int tmp = 0;
                  for (i=0; i<RTZ_HASH_DIVISOR; i++)
                        tmp += rt_flush_list(&fz->fz_hash_table[i],
dev);
                  fz->fz_nent -= tmp;
                  found += tmp;
            }
            else
            {
                  int tmp;
                  tmp = rt_flush_list(&fz->fz_list, dev);
                  fz->fz_nent -= tmp;
                  found += tmp;
            }
      }

      if (found)
            rt_cache_flush();
}


/*
 *    Called from the PROCfs module. This outputs /proc/net/route.
 *
 *    We preserve the old format but pad the buffers out. This means
that
 *    we can spin over the other entries as we read them. Remember the
 *    gated BGP4 code could need to read 60,000+ routes on occasion
(thats
 *    about 7Mb of data). To do that ok we will need to also cache the
 *    last route we got to (reads will generally be following on from
 *    one another without gaps).
 */

int rt_get_info(char *buffer, char **start, off_t offset, int length,
int dummy)
{
      struct fib_zone *fz;
      struct fib_node *f;
      int len=0;
      off_t pos=0;
      char temp[129];
      int i;

      pos = 128;

      if (offset<128)
      {
            sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\tIRTT");
            len = 128;
      }
```

```
      while  (ip_rt_lock)
            sleep_on(&rt_wait);
      ip_rt_fast_lock();

      for (fz=fib_zone_list; fz; fz = fz->fz_next)
      {
            int maxslot;
            struct fib_node ** fp;

            if (fz->fz_nent == 0)
                  continue;

            if (pos + 128*fz->fz_nent <= offset)
            {
                  pos += 128*fz->fz_nent;
                  len = 0;
                  continue;
            }

            if (fz->fz_hash_table)
            {
                  maxslot = RTZ_HASH_DIVISOR;
                  fp    = fz->fz_hash_table;
            }
            else
            {
                  maxslot     = 1;
                  fp    = &fz->fz_list;
            }

            for (i=0; i < maxslot; i++, fp++)
            {

                  for (f = *fp; f; f = f->fib_next)
                  {
                        struct fib_info * fi;
                        /*
                         *    Spin through entries until we are ready
                         */
                        pos += 128;

                        if (pos <= offset)
                        {
                              len=0;
                              continue;
                        }

                        fi = f->fib_info;
                        sprintf(temp, "%s\t%08lX\t%08lX\t%02X\t%d\t%lu
\t%d\t%08lX\t%d\t%lu\t%u",
                              fi->fib_dev->name, (unsigned long)f->
fib_dst, (unsigned long)fi->fib_gateway,
                              fi->fib_flags, 0, f->fib_use, f->
fib_metric,
                              (unsigned long)fz->fz_mask, (int)fi->
fib_mtu, fi->fib_window, (int)fi->fib_irtt);
                        sprintf(buffer+len,"%-127s\n",temp);

                        len += 128;
                        if (pos >= offset+length)
```

```
                                goto done;
                        }
                }
          }

done:
      ip_rt_unlock();
      wake_up(&rt_wait);

      *start = buffer+len-(pos-offset);
      len = pos - offset;
      if (len>length)
            len = length;
      return len;
}

int rt_cache_get_info(char *buffer, char **start, off_t offset, int
length, int dummy)
{
      int len=0;
      off_t pos=0;
      char temp[129];
      struct rtable *r;
      int i;

      pos = 128;

      if (offset<128)
      {
            sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tSource\t\tMTU\tWindow\tIRTT\tHH\tARP\n");
            len = 128;
      }


      while  (ip_rt_lock)
            sleep_on(&rt_wait);
      ip_rt_fast_lock();

      for (i = 0; i<RT_HASH_DIVISOR; i++)
      {
            for (r = ip_rt_hash_table[i]; r; r = r->rt_next)
            {
                  /*
                   *    Spin through entries until we are ready
                   */
                  pos += 128;

                  if (pos <= offset)
                  {
                        len = 0;
                        continue;
                  }

                  sprintf(temp, "%s\t%08lX\t%08lX\t%02X\t%ld\t%lu\t%d\t%
08lX\t%d\t%lu\t%u\t%ld\t%ld",
                        r->rt_dev->name, (unsigned long)r->rt_dst,
(unsigned long)r->rt_gateway,
                        r->rt_flags, r->rt_refcnt, r->rt_use, 0,
                        (unsigned long)r->rt_src, (int)r->rt_mtu, r->
rt_window, (int)r->rt_irtt, r->rt_hh ? r->rt_hh->hh_refcnt : -1, r->
```

```
rt_hh ? r->rt_hh->hh_uptodate : 0);
                  sprintf(buffer+len,"%-127s\n",temp);
                  len += 128;
                  if (pos >= offset+length)
                        goto done;
            }
        }

done:
      ip_rt_unlock();
      wake_up(&rt_wait);

      *start = buffer+len-(pos-offset);
      len = pos-offset;
      if (len>length)
            len = length;
      return len;
}


static void rt_free(struct rtable * rt)
{
      unsigned long flags;

      save_flags(flags);
      cli();
      if (!rt->rt_refcnt)
      {
            struct hh_cache * hh = rt->rt_hh;
            rt->rt_hh = NULL;
            if (hh && !--hh->hh_refcnt)
            {
                  restore_flags(flags);
                  kfree_s(hh, sizeof(struct hh_cache));
            }
            restore_flags(flags);
            kfree_s(rt, sizeof(struct rt_table));
            return;
      }
      rt->rt_next = rt_free_queue;
      rt->rt_flags &= ~RTF_UP;
      rt_free_queue = rt;
      ip_rt_bh_mask |= RT_BH_FREE;
#if RT_CACHE_DEBUG >= 2
      printk("rt_free: %08x\n", rt->rt_dst);
#endif
      restore_flags(flags);
}

/*
 * RT "bottom half" handlers. Called with masked inetrrupts.
 */

static __inline__ void rt_kick_free_queue(void)
{
      struct rtable *rt, **rtp;

      rtp = &rt_free_queue;

      while ((rt = *rtp) != NULL)
      {
```

```
            if  (!rt->rt_refcnt)
            {
                  struct hh_cache * hh = rt->rt_hh;
#if RT_CACHE_DEBUG >= 2
                  __u32 daddr = rt->rt_dst;
#endif
                  *rtp = rt->rt_next;
                  rt->rt_hh = NULL;
                  if (hh && !--hh->hh_refcnt)
                  {
                        sti();
                        kfree_s(hh, sizeof(struct hh_cache));
                  }
                  sti();
                  kfree_s(rt, sizeof(struct rt_table));
#if RT_CACHE_DEBUG >= 2
                  printk("rt_kick_free_queue: %08x is free\n", daddr);
#endif
                  cli();
                  continue;
            }
            rtp = &rt->rt_next;
      }
}

void ip_rt_run_bh() {
      unsigned long flags;
      save_flags(flags);
      cli();
      if (ip_rt_bh_mask && !ip_rt_lock)
      {
            if (ip_rt_bh_mask & RT_BH_REDIRECT)
                  rt_kick_backlog();

            if (ip_rt_bh_mask & RT_BH_GARBAGE_COLLECT)
            {
                  ip_rt_fast_lock();
                  ip_rt_bh_mask &= ~RT_BH_GARBAGE_COLLECT;
                  sti();
                  rt_garbage_collect_1();
                  cli();
                  ip_rt_fast_unlock();
            }

            if (ip_rt_bh_mask & RT_BH_FREE)
                  rt_kick_free_queue();
      }
      restore_flags(flags);
}


void ip_rt_check_expire()
{
      ip_rt_fast_lock();
      if (ip_rt_lock == 1)
      {
            int i;
            struct rtable *rth, **rthp;
            unsigned long flags;
            unsigned long now = jiffies;
```

```
            save_flags(flags);
            for (i=0; i<RT_HASH_DIVISOR; i++)
            {
                  rthp = &ip_rt_hash_table[i];

                  while ((rth = *rthp) != NULL)
                  {
                        struct rtable * rth_next = rth->rt_next;

                        /*
                         * Cleanup aged off entries.
                         */

                        cli();
                        if (!rth->rt_refcnt && rth->rt_lastuse +
RT_CACHE_TIMEOUT < now)
                        {
                              *rthp = rth_next;
                              sti();
                              rt_cache_size--;
#if RT_CACHE_DEBUG >= 2
                              printk("rt_check_expire clean %02x@%08x
\n", i, rth->rt_dst);
#endif
                              rt_free(rth);
                              continue;
                        }
                        sti();

                        if (!rth_next)
                              break;

                        /*
                         * LRU ordering.
                         */

                        if (rth->rt_lastuse + RT_CACHE_BUBBLE_THRESHOULD
< rth_next->rt_lastuse ||
                            (rth->rt_lastuse < rth_next->rt_lastuse &&
                             rth->rt_use < rth_next->rt_use))
                        {
#if RT_CACHE_DEBUG >= 2
                              printk("rt_check_expire bubbled %02x@%08x
<->%08x\n", i, rth->rt_dst, rth_next->rt_dst);
#endif
                              cli();
                              *rthp = rth_next;
                              rth->rt_next = rth_next->rt_next;
                              rth_next->rt_next = rth;
                              sti();
                              rthp = &rth_next->rt_next;
                              continue;
                        }
                        rthp = &rth->rt_next;
                  }
            }
            restore_flags(flags);
            rt_kick_free_queue();
      }
      ip_rt_unlock();
}
```

```
static void rt_redirect_1(__u32 dst, __u32 gw, struct device *dev)
{
      struct rtable *rt;
      unsigned long hash = ip_rt_hash_code(dst);

      if (gw == dev->pa_addr)
            return;
      if (dev != get_gw_dev(gw))
            return;
      rt = (struct rtable *) kmalloc(sizeof(struct rtable), GFP_ATOMIC);
      if (rt == NULL)
            return;
      memset(rt, 0, sizeof(struct rtable));
      rt->rt_flags = RTF_DYNAMIC | RTF_MODIFIED | RTF_HOST | RTF_GATEWAY
| RTF_UP;
      rt->rt_dst = dst;
      rt->rt_dev = dev;
      rt->rt_gateway = gw;
      rt->rt_src = dev->pa_addr;
      rt->rt_mtu = dev->mtu;
#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
      if (dev->mtu > 576)
            rt->rt_mtu = 576;
#endif
      rt->rt_lastuse  = jiffies;
      rt->rt_refcnt  = 1;
      rt_cache_add(hash, rt);
      ip_rt_put(rt);
      return;
}

static void rt_cache_flush(void)
{
      int i;
      struct rtable * rth, * next;

      for (i=0; i<RT_HASH_DIVISOR; i++)
      {
            int nr=0;

            cli();
            if (!(rth = ip_rt_hash_table[i]))
            {
                  sti();
                  continue;
            }

            ip_rt_hash_table[i] = NULL;
            sti();

            for (; rth; rth=next)
            {
                  next = rth->rt_next;
                  rt_cache_size--;
                  nr++;
                  rth->rt_next = NULL;
                  rt_free(rth);
            }
#if RT_CACHE_DEBUG >= 2
            if (nr > 0)
```

```
                  printk("rt_cache_flush: %d@%02x\n", nr, i);
#endif
      }
#if RT_CACHE_DEBUG >= 1
      if (rt_cache_size)
      {
            printk("rt_cache_flush: bug rt_cache_size=%d\n",
rt_cache_size);
            rt_cache_size = 0;
      }
#endif
}

static void rt_garbage_collect_1(void)
{
      int i;
      unsigned expire = RT_CACHE_TIMEOUT>>1;
      struct rtable * rth, **rthp;
      unsigned long now = jiffies;

      for (;;)
      {
            for (i=0; i<RT_HASH_DIVISOR; i++)
            {
                  if (!ip_rt_hash_table[i])
                        continue;
                  for (rthp=&ip_rt_hash_table[i]; (rth=*rthp);
rthp=&rth->rt_next)
                  {
                        if (rth->rt_lastuse + expire*(rth->rt_refcnt+1)
> now)
                              continue;
                        rt_cache_size--;
                        cli();
                        *rthp=rth->rt_next;
                        rth->rt_next = NULL;
                        sti();
                        rt_free(rth);
                        break;
                  }
            }
            if (rt_cache_size < RT_CACHE_SIZE_MAX)
                  return;
            expire >>= 1;
      }
}

static __inline__ void rt_req_enqueue(struct rt_req **q, struct rt_req
*rtr)
{
      unsigned long flags;
      struct rt_req * tail;

      save_flags(flags);
      cli();
      tail = *q;
      if (!tail)
            rtr->rtr_next = rtr;
      else
      {
            rtr->rtr_next = tail->rtr_next;
```

```
            tail->rtr_next = rtr;
      }
      *q = rtr;
      restore_flags(flags);
      return;
}

/*
 * Caller should mask interrupts.
 */

static __inline__ struct rt_req * rt_req_dequeue(struct rt_req **q)
{
      struct rt_req * rtr;

      if (*q)
      {
            rtr = (*q)->rtr_next;
            (*q)->rtr_next = rtr->rtr_next;
            if (rtr->rtr_next == rtr)
                  *q = NULL;
            rtr->rtr_next = NULL;
            return rtr;
      }
      return NULL;
}

/*
   Called with masked interrupts
 */

static void rt_kick_backlog()
{
      if (!ip_rt_lock)
      {
            struct rt_req * rtr;

            ip_rt_fast_lock();

            while ((rtr = rt_req_dequeue(&rt_backlog)) != NULL)
            {
                  sti();
                  rt_redirect_1(rtr->dst, rtr->gw, rtr->dev);
                  kfree_s(rtr, sizeof(struct rt_req));
                  cli();
            }

            ip_rt_bh_mask &= ~RT_BH_REDIRECT;

            ip_rt_fast_unlock();
      }
}

/*
 * rt_{del|add|flush} called only from USER process. Waiting is OK.
 */

static int rt_del(__u32 dst, __u32 mask,
            struct device * dev, __u32 gtw, short rt_flags, short
metric)
{
```

```
        int retval;

        while (ip_rt_lock)
                sleep_on(&rt_wait);
        ip_rt_fast_lock();
        retval = fib_del_1(dst, mask, dev, gtw, rt_flags, metric);
        ip_rt_unlock();
        wake_up(&rt_wait);
        return retval;
}

static void rt_add(short flags, __u32 dst, __u32 mask,
        __u32 gw, struct device *dev, unsigned short mss,
        unsigned long window, unsigned short irtt, short metric)
{
        while (ip_rt_lock)
                sleep_on(&rt_wait);
        ip_rt_fast_lock();
        fib_add_1(flags, dst, mask, gw, dev, mss, window, irtt, metric);
        ip_rt_unlock();
        wake_up(&rt_wait);
}

void ip_rt_flush(struct device *dev)
{
        while (ip_rt_lock)
                sleep_on(&rt_wait);
        ip_rt_fast_lock();
        fib_flush_1(dev);
        ip_rt_unlock();
        wake_up(&rt_wait);
}

/*
   Called by ICMP module.
 */

void ip_rt_redirect(__u32 src, __u32 dst, __u32 gw, struct device *dev)
{
        struct rt_req * rtr;
        struct rtable * rt;

        rt = ip_rt_route(dst, 0);
        if (!rt)
                return;

        if (rt->rt_gateway != src ||
            rt->rt_dev != dev ||
            ((gw^dev->pa_addr)&dev->pa_mask) ||
            ip_chk_addr(gw))
        {
                ip_rt_put(rt);
                return;
        }
        ip_rt_put(rt);

        ip_rt_fast_lock();
        if (ip_rt_lock == 1)
        {
                rt_redirect_1(dst, gw, dev);
                ip_rt_unlock();
```

```
                return;
        }

        rtr = kmalloc(sizeof(struct rt_req), GFP_ATOMIC);
        if (rtr)
        {
                rtr->dst = dst;
                rtr->gw = gw;
                rtr->dev = dev;
                rt_req_enqueue(&rt_backlog, rtr);
                ip_rt_bh_mask |= RT_BH_REDIRECT;
        }
        ip_rt_unlock();
}


static __inline__ void rt_garbage_collect(void)
{
        if (ip_rt_lock == 1)
        {
                rt_garbage_collect_1();
                return;
        }
        ip_rt_bh_mask |= RT_BH_GARBAGE_COLLECT;
}

static void rt_cache_add(unsigned hash, struct rtable * rth)
{
        unsigned long     flags;
        struct rtable     **rthp;
        __u32       daddr = rth->rt_dst;
        unsigned long     now = jiffies;

#if RT_CACHE_DEBUG >= 2
        if (ip_rt_lock != 1)
        {
                printk("rt_cache_add: ip_rt_lock==%d\n", ip_rt_lock);
                return;
        }
#endif

        save_flags(flags);

        if (rth->rt_dev->header_cache_bind)
        {
                struct rtable * rtg = rth;

                if (rth->rt_gateway != daddr)
                {
                        ip_rt_fast_unlock();
                        rtg = ip_rt_route(rth->rt_gateway, 0);
                        ip_rt_fast_lock();
                }

                if (rtg)
                {
                        if (rtg == rth)
                                rtg->rt_dev->header_cache_bind(&rtg->rt_hh,
rtg->rt_dev, ETH_P_IP, rtg->rt_dst);
                        else
                        {
```

```
                        if (rtg->rt_hh)
                              ATOMIC_INCR(&rtg->rt_hh->hh_refcnt);
                        rth->rt_hh = rtg->rt_hh;
                        ip_rt_put(rtg);
                  }
            }
      }

      if (rt_cache_size >= RT_CACHE_SIZE_MAX)
            rt_garbage_collect();

      cli();
      rth->rt_next = ip_rt_hash_table[hash];
#if RT_CACHE_DEBUG >= 2
      if (rth->rt_next)
      {
            struct rtable * trth;
            printk("rt_cache @%02x: %08x", hash, daddr);
            for (trth=rth->rt_next; trth; trth=trth->rt_next)
                  printk(" . %08x", trth->rt_dst);
            printk("\n");
      }
#endif
      ip_rt_hash_table[hash] = rth;
      rthp = &rth->rt_next;
      sti();
      rt_cache_size++;

      /*
       * Cleanup duplicate (and aged off) entries.
       */

      while ((rth = *rthp) != NULL)
      {

            cli();
            if ((!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT
< now)
                || rth->rt_dst == daddr)
            {
                  *rthp = rth->rt_next;
                  rt_cache_size--;
                  sti();
#if RT_CACHE_DEBUG >= 2
                  printk("rt_cache clean %02x@%08x\n", hash, rth->
rt_dst);
#endif
                  rt_free(rth);
                  continue;
            }
            sti();
            rthp = &rth->rt_next;
      }
      restore_flags(flags);
}

/*
   RT should be already locked.

   We could improve this by keeping a chain of say 32 struct rtable's
   last freed for fast recycling.
```

```
 */

struct rtable * ip_rt_slow_route (__u32 daddr, int local)
{
      unsigned hash = ip_rt_hash_code(daddr)^local;
      struct rtable * rth;
      struct fib_node * f;
      struct fib_info * fi;
      __u32 saddr;

#if RT_CACHE_DEBUG >= 2
      printk("rt_cache miss @%08x\n", daddr);
#endif

      rth = kmalloc(sizeof(struct rtable), GFP_ATOMIC);
      if (!rth)
      {
            ip_rt_unlock();
            return NULL;
      }

      if (local)
            f = fib_lookup_local(daddr);
      else
            f = fib_lookup (daddr);

      if (f)
      {
            fi = f->fib_info;
            f->fib_use++;
      }

      if (!f || (fi->fib_flags & RTF_REJECT))
      {
#if RT_CACHE_DEBUG >= 2
            printk("rt_route failed @%08x\n", daddr);
#endif
            ip_rt_unlock();
            kfree_s(rth, sizeof(struct rtable));
            return NULL;
      }

      saddr = fi->fib_dev->pa_addr;

      if (daddr == fi->fib_dev->pa_addr)
      {
            f->fib_use--;
            if ((f = fib_loopback) != NULL)
            {
                  f->fib_use++;
                  fi = f->fib_info;
            }
      }

      if (!f)
      {
            ip_rt_unlock();
            kfree_s(rth, sizeof(struct rtable));
            return NULL;
      }
```

```
	rth->rt_dst = daddr;
	rth->rt_src = saddr;
	rth->rt_lastuse   = jiffies;
	rth->rt_refcnt    = 1;
	rth->rt_use = 1;
	rth->rt_next      = NULL;
	rth->rt_hh  = NULL;
	rth->rt_gateway   = fi->fib_gateway;
	rth->rt_dev = fi->fib_dev;
	rth->rt_mtu = fi->fib_mtu;
	rth->rt_window    = fi->fib_window;
	rth->rt_irtt      = fi->fib_irtt;
	rth->rt_tos = f->fib_tos;
	rth->rt_flags   = fi->fib_flags | RTF_HOST;
	if (local)
		rth->rt_flags   |= RTF_LOCAL;

	if (!(rth->rt_flags & RTF_GATEWAY))
		rth->rt_gateway = rth->rt_dst;

	if (ip_rt_lock == 1)
		rt_cache_add(hash, rth);
	else
	{
		rt_free(rth);
#if RT_CACHE_DEBUG >= 1
		printk("rt_cache: route to %08x was born dead\n", daddr);
#endif
	}

	ip_rt_unlock();
	return rth;
}

void ip_rt_put(struct rtable * rt)
{
	if (rt)
		ATOMIC_DECR(&rt->rt_refcnt);
}

struct rtable * ip_rt_route(__u32 daddr, int local)
{
	struct rtable * rth;

	ip_rt_fast_lock();

	for (rth=ip_rt_hash_table[ip_rt_hash_code(daddr)^local]; rth;
rth=rth->rt_next)
	{
		if (rth->rt_dst == daddr)
		{
			rth->rt_lastuse = jiffies;
			ATOMIC_INCR(&rth->rt_use);
			ATOMIC_INCR(&rth->rt_refcnt);
			ip_rt_unlock();
			return rth;
		}
	}
	return ip_rt_slow_route (daddr, local);
}
```

```
/*
 *    Process a route add request from the user, or from a kernel
 *    task.
 */

int ip_rt_new(struct rtentry *r)
{
      int err;
      char * devname;
      struct device * dev = NULL;
      unsigned long flags;
      __u32 daddr, mask, gw;
      short metric;

      /*
       *    If a device is specified find it.
       */

      if ((devname = r->rt_dev) != NULL)
      {
            err = getname(devname, &devname);
            if (err)
                  return err;
            dev = dev_get(devname);
            putname(devname);
            if (!dev)
                  return -ENODEV;
      }

      /*
       *    If the device isn't INET, don't allow it
       */

      if (r->rt_dst.sa_family != AF_INET)
            return -EAFNOSUPPORT;

      /*
       *    Make local copies of the important bits
       *    We decrement the metric by one for BSD compatibility.
       */

      flags = r->rt_flags;
      daddr = (__u32) ((struct sockaddr_in *) &r->rt_dst)->
sin_addr.s_addr;
      mask  = (__u32) ((struct sockaddr_in *) &r->rt_genmask)->
sin_addr.s_addr;
      gw    = (__u32) ((struct sockaddr_in *) &r->rt_gateway)->
sin_addr.s_addr;
      metric = r->rt_metric > 0 ? r->rt_metric - 1 : 0;

      /*
       *    BSD emulation: Permits route add someroute gw one-of-my-
addresses
       *    to indicate which iface. Not as clean as the nice Linux dev
technique
       *    but people keep using it...  (and gated likes it ;))
       */

      if (!dev && (flags & RTF_GATEWAY))
```

```
	{
		struct device *dev2;
		for (dev2 = dev_base ; dev2 != NULL ; dev2 = dev2->next)
		{
			if ((dev2->flags & IFF_UP) && dev2->pa_addr == gw)
			{
				flags &= ~RTF_GATEWAY;
				dev = dev2;
				break;
			}
		}
	}

	/*
	 *	Ignore faulty masks
	 */

	if (bad_mask(mask, daddr))
		mask=0;

	/*
	 *	Set the mask to nothing for host routes.
	 */

	if (flags & RTF_HOST)
		mask = 0xffffffff;
	else if (mask && r->rt_genmask.sa_family != AF_INET)
		return -EAFNOSUPPORT;

	/*
	 *	You can only gateway IP via IP..
	 */

	if (flags & RTF_GATEWAY)
	{
		if (r->rt_gateway.sa_family != AF_INET)
			return -EAFNOSUPPORT;
		if (!dev)
			dev = get_gw_dev(gw);
	}
	else if (!dev)
		dev = ip_dev_check(daddr);

	/*
	 *	Unknown device.
	 */

	if (dev == NULL)
		return -ENETUNREACH;

	/*
	 *	Add the route
	 */

	rt_add(flags, daddr, mask, gw, dev, r->rt_mss, r->rt_window, r->
rt_irtt, metric);
	return 0;
}


/*
```

```
 *    Remove a route, as requested by the user.
 */

static int rt_kill(struct rtentry *r)
{
      struct sockaddr_in *trg;
      struct sockaddr_in *msk;
      struct sockaddr_in *gtw;
      char *devname;
      int err;
      struct device * dev = NULL;

      trg = (struct sockaddr_in *) &r->rt_dst;
      msk = (struct sockaddr_in *) &r->rt_genmask;
      gtw = (struct sockaddr_in *) &r->rt_gateway;
      if ((devname = r->rt_dev) != NULL)
      {
            err = getname(devname, &devname);
            if (err)
                  return err;
            dev = dev_get(devname);
            putname(devname);
            if (!dev)
                  return -ENODEV;
      }
      /*
       * metric can become negative here if it wasn't filled in
       * but that's a fortunate accident; we really use that in rt_del.
       */
      err=rt_del((__u32)trg->sin_addr.s_addr, (__u32)msk->
sin_addr.s_addr, dev,
            (__u32)gtw->sin_addr.s_addr, r->rt_flags, r->rt_metric - 1);
      return err;
}

/*
 *    Handle IP routing ioctl calls. These are used to manipulate the
routing tables
 */

int ip_rt_ioctl(unsigned int cmd, void *arg)
{
      int err;
      struct rtentry rt;

      switch(cmd)
      {
            case SIOCADDRT:         /* Add a route */
            case SIOCDELRT:         /* Delete a route */
                  if (!suser())
                        return -EPERM;
                  err=verify_area(VERIFY_READ, arg, sizeof(struct
rtentry));
                  if (err)
                        return err;
                  memcpy_fromfs(&rt, arg, sizeof(struct rtentry));
                  return (cmd == SIOCDELRT) ? rt_kill(&rt) : ip_rt_new
(&rt);
      }

      return -EINVAL;
```

```
}

void ip_rt_advice(struct rtable **rp, int advice)
{
      /* Thanks! */
      return;
}
```

```
1/*
2 * INET        An implementation of the TCP/IP protocol suite for the LINUX
3 *             operating system.  INET is implemented using the  BSD Socket
4 *             interface as the means of communication with the user level.
5 *
6 *             ROUTE - implementation of the IP router.
7 *
8 * Version:     @(#)route.c     1.0.14  05/31/93
9 *
10 * Authors:     Ross Biro, <bir7@leland.Stanford.Edu>
11 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12 *              Alan Cox, <gw4pts@gw4pts.ampr.org>
13 *              Linus Torvalds, <Linus.Torvalds@helsinki.fi>
14 *
15 * Fixes:
16 *              Alan Cox        :       Verify area fixes.
17 *              Alan Cox        :       cli() protects routing changes
18 *              Rui Oliveira    :       ICMP routing table updates
19 *              (rco@di.uminho.pt)      Routing table insertion and update
20 *              Linus Torvalds  :       Rewrote bits to be sensible
21 *              Alan Cox        :       Added BSD route gw semantics
22 *              Alan Cox        :       Super /proc >4K
23 *              Alan Cox        :       MTU in route table
24 *              Alan Cox        :       MSS actually. Also added the window
25 *                                      clamper.
26 *              Sam Lantinga    :       Fixed route matching in rt_del()
27 *              Alan Cox        :       Routing cache support.
28 *              Alan Cox        :       Removed compatibility cruft.
29 *              Alan Cox        :       RTF_REJECT support.
30 *              Alan Cox        :       TCP irtt support.
31 *              Jonathan Naylor :       Added Metric support.
32 *      Miquel van Smoorenburg  :       BSD API fixes.
33 *      Miquel van Smoorenburg  :       Metrics.
34 *              Alan Cox        :       Use __u32 properly
35 *              Alan Cox        :       Aligned routing errors more closely with BSD
36 *                                      our system is still very different.
37 *              Alan Cox        :       Faster /proc handling
38 *      Alexey Kuznetsov        :       Massive rework to support tree based routing,
39 *                                      routing caches and better behaviour.
40 *
41 *              Olaf Erb        :       irtt wasn't being copied right.
42 *              Bjorn Ekwall    :       Kerneld route support.
43 *              Alan Cox        :       Multicast fixed (I hope)
44 *              Pavel Krauz     :       Limited broadcast fixed
45 *
46 *              This program is free software; you can redistribute it and/or
47 *              modify it under the terms of the GNU General Public License
48 *              as published by the Free Software Foundation; either version
49 *              2 of the License, or (at your option) any later version.
50 */
51
```

```
52#include <linux/config.h>
53#include <asm/segment.h>
54#include <asm/system.h>
55#include <asm/bitops.h>
56#include <linux/types.h>
57#include <linux/kernel.h>
58#include <linux/sched.h>
59#include <linux/mm.h>
60#include <linux/string.h>
61#include <linux/socket.h>
62#include <linux/sockios.h>
63#include <linux/errno.h>
64#include <linux/in.h>
65#include <linux/inet.h>
66#include <linux/netdevice.h>
67#include <linux/if_arp.h>
68#include <net/ip.h>
69#include <net/protocol.h>
70#include <net/route.h>
71#include <net/tcp.h>
72#include <linux/skbuff.h>
73#include <net/sock.h>
74#include <net/icmp.h>
75#include <net/netlink.h>
76#ifdef CONFIG_KERNELD
77#include <linux/kerneld.h>
78#endif
79
80/*
81 * Forwarding Information Base definitions.
82 */
83
84struct fib_node
85{
86        struct fib_node         *fib_next;
87        __u32                   fib_dst;
88        unsigned long           fib_use;
89        struct fib_info         *fib_info;
90        short                   fib_metric;
91        unsigned char           fib_tos;
92};
93
94/*
95 * This structure contains data shared by many of routes.
96 */
97
98struct fib_info
99{
100       struct fib_info         *fib_next;
101       struct fib_info         *fib_prev;
102       __u32                   fib_gateway;
```

```
103     struct device           *fib_dev;
104     int                     fib_refcnt;
105     unsigned long           fib_window;
106     unsigned short          fib_flags;
107     unsigned short          fib_mtu;
108     unsigned short          fib_irtt;
109};
110
111struct fib_zone
112{
113     struct fib_zone *fz_next;
114     struct fib_node **fz_hash_table;
115     struct fib_node *fz_list;
116     int             fz_nent;
117     int             fz_logmask;
118     __u32           fz_mask;
119};
120
121static struct fib_zone  *fib_zones[33];
122static struct fib_zone  *fib_zone_list;
123static struct fib_node  *fib_loopback = NULL;
124static struct fib_info  *fib_info_list;
125
126/*
127 * Backlogging.
128 */
129
130#define RT_BH_REDIRECT          0
131#define RT_BH_GARBAGE_COLLECT   1
132#define RT_BH_FREE              2
133
134struct rt_req
135{
136     struct rt_req * rtr_next;
137     struct device *dev;
138     __u32 dst;
139     __u32 gw;
140     unsigned char tos;
141};
142
143int                    ip_rt_lock;
144unsigned               ip_rt_bh_mask;
145static struct rt_req    *rt_backlog;
146
147/*
148 * Route cache.
149 */
150
151struct rtable          *ip_rt_hash_table[RT_HASH_DIVISOR];
152static int             rt_cache_size;
153static struct rtable    *rt_free_queue;
```

```
154struct wait_queue       *rt_wait;
155
156static void rt_kick_backlog(void);
157static void rt_cache_add(unsigned hash, struct rtable * rth);
158static void rt_cache_flush(void);
159static void rt_garbage_collect_1(void);
160
161/*
162 * Evaluate mask length.
163 */
164
165static __inline__ int rt_logmask(__u32 mask)
166{
167       if (!(mask = ntohl(mask)))
168              return 32;
169       return ffz(~mask);
170}
171
172/*
173 * Create mask from length.
174 */
175
176static __inline__ __u32 rt_mask(int logmask)
177{
178       if (logmask >= 32)
179              return 0;
180       return htonl(~((1<<logmask)-1));
181}
182
183static __inline__ unsigned fz_hash_code(__u32 dst, int logmask)
184{
185       return ip_rt_hash_code(ntohl(dst)>>logmask);
186}
187
188/*
189 * Free FIB node.
190 */
191
192static void fib_free_node(struct fib_node * f)
193{
194       struct fib_info * fi = f->fib_info;
195       if (!--fi->fib_refcnt)
196       {
197#if RT_CACHE_DEBUG >= 2
198              printk("fib_free_node: fi %08x/%s is free\n", fi->fib_gateway, fi->fib_dev->name);
199#endif
200              if (fi->fib_next)
201                     fi->fib_next->fib_prev = fi->fib_prev;
202              if (fi->fib_prev)
203                     fi->fib_prev->fib_next = fi->fib_next;
204              if (fi == fib_info_list)
```

```
205                     fib_info_list = fi->fib_next;
206             }
207             kfree_s(f, sizeof(struct fib_node));
208}
209
210/*
211 * Find gateway route by address.
212 */
213
214static struct fib_node * fib_lookup_gateway(__u32 dst)
215{
216             struct fib_zone * fz;
217             struct fib_node * f;
218
219             for (fz = fib_zone_list; fz; fz = fz->fz_next)
220             {
221                     if (fz->fz_hash_table)
222                             f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
223                     else
224                             f = fz->fz_list;
225
226                     for ( ; f; f = f->fib_next)
227                     {
228                             if ((dst ^ f->fib_dst) & fz->fz_mask)
229                                     continue;
230                             if (f->fib_info->fib_flags & RTF_GATEWAY)
231                                     return NULL;
232                             return f;
233                     }
234             }
235             return NULL;
236}
237
238/*
239 * Find local route by address.
240 * FIXME: I use "longest match" principle. If destination
241 *        has some non-local route, I'll not search shorter matches.
242 *        It's possible, I'm wrong, but I wanted to prevent following
243 *        situation:
244 *      route add 193.233.7.128 netmask 255.255.255.192 gw xxxxxx
245 *      route add 193.233.7.0   netmask 255.255.255.0 eth1
246 *        (Two ethernets connected by serial line, one is small and other is large)
247 *        Host 193.233.7.129 is locally unreachable,
248 *        but old (<=1.3.37) code will send packets destined for it to eth1.
249 *
250 */
251
252static struct fib_node * fib_lookup_local(__u32 dst)
253{
254             struct fib_zone * fz;
255             struct fib_node * f;
```

```
256
257       for (fz = fib_zone_list; fz; fz = fz->fz_next)
258       {
259             int longest_match_found = 0;
260
261             if (fz->fz_hash_table)
262                   f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
263             else
264                   f = fz->fz_list;
265
266             for ( ; f; f = f->fib_next)
267             {
268                   if ((dst ^ f->fib_dst) & fz->fz_mask)
269                         continue;
270                   if (!(f->fib_info->fib_flags & RTF_GATEWAY))
271                         return f;
272                   longest_match_found = 1;
273             }
274             if (longest_match_found)
275                   return NULL;
276       }
277       return NULL;
278}
279
280/*
281 * Main lookup routine.
282 *      IMPORTANT NOTE: this algorithm has small difference from <=1.3.37 visible
283 *      by user. It doesn't route non-CIDR broadcasts by default.
284 *
285 *      F.e.
286 *            ifconfig eth0 193.233.7.65 netmask 255.255.255.192 broadcast 193.233.7.255
287 *      is valid, but if you really are not able (not allowed, do not want) to
288 *      use CIDR compliant broadcast 193.233.7.127, you should add host route:
289 *            route add -host 193.233.7.255 eth0
290 */
291
292static struct fib_node * fib_lookup(__u32 dst)
293{
294       struct fib_zone * fz;
295       struct fib_node * f;
296
297       for (fz = fib_zone_list; fz; fz = fz->fz_next)
298       {
299             if (fz->fz_hash_table)
300                   f = fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
301             else
302                   f = fz->fz_list;
303
304             for ( ; f; f = f->fib_next)
305             {
306                   if ((dst ^ f->fib_dst) & fz->fz_mask)
```

```
307                         continue;
308                 return f;
309             }
310     }
311     return NULL;
312}
313
314static __inline__ struct device * get_gw_dev(__u32 gw)
315{
316     struct fib_node * f;
317     f = fib_lookup_gateway(gw);
318     if (f)
319         return f->fib_info->fib_dev;
320     return NULL;
321}
322
323/*
324 *      Check if a mask is acceptable.
325 */
326
327static inline int bad_mask(__u32 mask, __u32 addr)
328{
329     if (addr & (mask = ~mask))
330         return 1;
331     mask = ntohl(mask);
332     if (mask & (mask+1))
333         return 1;
334     return 0;
335}
336
337
338static int fib_del_list(struct fib_node **fp, __u32 dst,
339         struct device * dev, __u32 gtw, short flags, short metric, __u32 mask)
340{
341     struct fib_node *f;
342     int found=0;
343
344     while((f = *fp) != NULL)
345     {
346         struct fib_info * fi = f->fib_info;
347
348         /*
349          *      Make sure the destination and netmask match.
350          *      metric, gateway and device are also checked
351          *      if they were specified.
352          */
353         if (f->fib_dst != dst ||
354            (gtw && fi->fib_gateway != gtw) ||
355            (metric >= 0 && f->fib_metric != metric) ||
356            (dev && fi->fib_dev != dev) )
357         {
```

```
358                 fp = &f->fib_next;
359                 continue;
360         }
361         cli();
362         *fp = f->fib_next;
363         if (fib_loopback == f)
364                 fib_loopback = NULL;
365         sti();
366         ip_netlink_msg(RTMSG_DELROUTE, dst, gtw, mask, flags, metric, fi->fib_dev->name);
367         fib_free_node(f);
368         found++;
369     }
370     return found;
371}
372
373static __inline__ int fib_del_1(__u32 dst, __u32 mask,
374         struct device * dev, __u32 gtw, short flags, short metric)
375{
376     struct fib_node **fp;
377     struct fib_zone *fz;
378     int found=0;
379
380     if (!mask)
381     {
382         for (fz=fib_zone_list; fz; fz = fz->fz_next)
383         {
384             int tmp;
385             if (fz->fz_hash_table)
386                 fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
387             else
388                 fp = &fz->fz_list;
389
390             tmp = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
391             fz->fz_nent -= tmp;
392             found += tmp;
393         }
394     }
395     else
396     {
397         if ((fz = fib_zones[rt_logmask(mask)]) != NULL)
398         {
399             if (fz->fz_hash_table)
400                 fp = &fz->fz_hash_table[fz_hash_code(dst, fz->fz_logmask)];
401             else
402                 fp = &fz->fz_list;
403
404             found = fib_del_list(fp, dst, dev, gtw, flags, metric, mask);
405             fz->fz_nent -= found;
406         }
407     }
408
```

```
409     if (found)
410     {
411             rt_cache_flush();
412             return 0;
413     }
414     return -ESRCH;
415}
416
417
418static struct fib_info * fib_create_info(__u32 gw, struct device * dev,
419                                 unsigned short flags, unsigned short mss,
420                                 unsigned long window, unsigned short irtt)
421{
422     struct fib_info * fi;
423
424     if (!(flags & RTF_MSS))
425     {
426             mss = dev->mtu;
427#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
428             /*
429              *      If MTU was not specified, use default.
430              *      If you want to increase MTU for some net (local subnet)
431              *      use "route add .... mss xxx".
432              *
433              *      The MTU isn't currently always used and computed as it
434              *      should be as far as I can tell. [Still verifying this is right]
435              */
436             if ((flags & RTF_GATEWAY) && mss > 576)
437                     mss = 576;
438#endif
439     }
440     if (!(flags & RTF_WINDOW))
441             window = 0;
442     if (!(flags & RTF_IRTT))
443             irtt = 0;
444
445     for (fi=fib_info_list; fi; fi = fi->fib_next)
446     {
447             if (fi->fib_gateway != gw ||
448                 fi->fib_dev != dev  ||
449                 fi->fib_flags != flags ||
450                 fi->fib_mtu != mss ||
451                 fi->fib_window != window ||
452                 fi->fib_irtt != irtt)
453                     continue;
454             fi->fib_refcnt++;
455#if RT_CACHE_DEBUG >= 2
456             printk("fib_create_info: fi %08x/%s is duplicate\n", fi->fib_gateway, fi->fib_dev->name);
457#endif
458             return fi;
459     }
```

```
460     fi = (struct fib_info*)kmalloc(sizeof(struct fib_info), GFP_KERNEL);
461     if (!fi)
462             return NULL;
463     memset(fi, 0, sizeof(struct fib_info));
464     fi->fib_flags = flags;
465     fi->fib_dev = dev;
466     fi->fib_gateway = gw;
467     fi->fib_mtu = mss;
468     fi->fib_window = window;
469     fi->fib_refcnt++;
470     fi->fib_next = fib_info_list;
471     fi->fib_prev = NULL;
472     fi->fib_irtt = irtt;
473     if (fib_info_list)
474             fib_info_list->fib_prev = fi;
475     fib_info_list = fi;
476#if RT_CACHE_DEBUG >= 2
477     printk("fib_create_info: fi %08x/%s is created\n", fi->fib_gateway, fi->fib_dev->name);
478#endif
479     return fi;
480}
481
482
483static __inline__ void fib_add_1(short flags, __u32 dst, __u32 mask,
484     __u32 gw, struct device *dev, unsigned short mss,
485     unsigned long window, unsigned short irtt, short metric)
486{
487     struct fib_node *f, *f1;
488     struct fib_node **fp;
489     struct fib_node **dup_fp = NULL;
490     struct fib_zone * fz;
491     struct fib_info * fi;
492     int logmask;
493
494     /*
495      *      Allocate an entry and fill it in.
496      */
497
498     f = (struct fib_node *) kmalloc(sizeof(struct fib_node), GFP_KERNEL);
499     if (f == NULL)
500             return;
501
502     memset(f, 0, sizeof(struct fib_node));
503     f->fib_dst = dst;
504     f->fib_metric = metric;
505     f->fib_tos    = 0;
506
507     if  ((fi = fib_create_info(gw, dev, flags, mss, window, irtt)) == NULL)
508     {
509             kfree_s(f, sizeof(struct fib_node));
510             return;
```

```
511     }
512     f->fib_info = fi;
513
514     logmask = rt_logmask(mask);
515     fz = fib_zones[logmask];
516
517
518     if (!fz)
519     {
520             int i;
521             fz = kmalloc(sizeof(struct fib_zone), GFP_KERNEL);
522             if (!fz)
523             {
524                     fib_free_node(f);
525                     return;
526             }
527             memset(fz, 0, sizeof(struct fib_zone));
528             fz->fz_logmask = logmask;
529             fz->fz_mask = mask;
530             for (i=logmask-1; i>=0; i--)
531                     if (fib_zones[i])
532                             break;
533             cli();
534             if (i<0)
535             {
536                     fz->fz_next = fib_zone_list;
537                     fib_zone_list = fz;
538             }
539             else
540             {
541                     fz->fz_next = fib_zones[i]->fz_next;
542                     fib_zones[i]->fz_next = fz;
543             }
544             fib_zones[logmask] = fz;
545             sti();
546     }
547
548     /*
549      * If zone overgrows RTZ_HASHING_LIMIT, create hash table.
550      */
551
552     if (fz->fz_nent >= RTZ_HASHING_LIMIT && !fz->fz_hash_table && logmask<32)
553     {
554             struct fib_node ** ht;
555#if RT_CACHE_DEBUG >= 2
556             printk("fib_add_1: hashing for zone %d started\n", logmask);
557#endif
558             ht = kmalloc(RTZ_HASH_DIVISOR*sizeof(struct rtable*), GFP_KERNEL);
559
560             if (ht)
561             {
```

```
562                 memset(ht, 0, RTZ_HASH_DIVISOR*sizeof(struct fib_node*));
563                 cli();
564                 f1 = fz->fz_list;
565                 while (f1)
566                 {
567                         struct fib_node * next;
568                         unsigned hash = fz_hash_code(f1->fib_dst, logmask);
569                         next = f1->fib_next;
570                         f1->fib_next = ht[hash];
571                         ht[hash] = f1;
572                         f1 = next;
573                 }
574                 fz->fz_list = NULL;
575                 fz->fz_hash_table = ht;
576                 sti();
577         }
578     }
579
580     if (fz->fz_hash_table)
581         fp = &fz->fz_hash_table[fz_hash_code(dst, logmask)];
582     else
583         fp = &fz->fz_list;
584
585     /*
586      * Scan list to find the first route with the same destination
587      */
588     while ((f1 = *fp) != NULL)
589     {
590         if (f1->fib_dst == dst)
591             break;
592         fp = &f1->fib_next;
593     }
594
595     /*
596      * Find route with the same destination and less (or equal) metric.
597      */
598     while ((f1 = *fp) != NULL && f1->fib_dst == dst)
599     {
600         if (f1->fib_metric >= metric)
601             break;
602         /*
603          *      Record route with the same destination and gateway,
604          *      but less metric. We'll delete it
605          *      after instantiation of new route.
606          */
607         if (f1->fib_info->fib_gateway == gw &&
608            (gw || f1->fib_info->fib_dev == dev))
609             dup_fp = fp;
610         fp = &f1->fib_next;
611     }
612
```

```
613     /*
614      * Is it already present?
615      */
616
617     if (f1 && f1->fib_metric == metric && f1->fib_info == fi)
618     {
619             fib_free_node(f);
620             return;
621     }
622
623     /*
624      * Insert new entry to the list.
625      */
626
627     cli();
628     f->fib_next = f1;
629     *fp = f;
630     if (!fib_loopback && (fi->fib_dev->flags & IFF_LOOPBACK))
631             fib_loopback = f;
632     sti();
633     fz->fz_nent++;
634     ip_netlink_msg(RTMSG_NEWROUTE, dst, gw, mask, flags, metric, fi->fib_dev->name);
635
636     /*
637      *      Delete route with the same destination and gateway.
638      *      Note that we should have at most one such route.
639      */
640     if (dup_fp)
641             fp = dup_fp;
642     else
643             fp = &f->fib_next;
644
645     while ((f1 = *fp) != NULL && f1->fib_dst == dst)
646     {
647             if (f1->fib_info->fib_gateway == gw &&
648                (gw || f1->fib_info->fib_dev == dev))
649             {
650                     cli();
651                     *fp = f1->fib_next;
652                     if (fib_loopback == f1)
653                             fib_loopback = NULL;
654                     sti();
655                     ip_netlink_msg(RTMSG_DELROUTE, dst, gw, mask, flags, metric, f1->fib_info-
>fib_dev->name);
656                     fib_free_node(f1);
657                     fz->fz_nent--;
658                     break;
659             }
660             fp = &f1->fib_next;
661     }
662     rt_cache_flush();
```

```
663       return;
664}
665
666static int rt_flush_list(struct fib_node ** fp, struct device *dev)
667{
668       int found = 0;
669       struct fib_node *f;
670
671       while ((f = *fp) != NULL) {
672/*
673 *      "Magic" device route is allowed to point to loopback,
674 *      discard it too.
675 */
676             if (f->fib_info->fib_dev != dev &&
677                 (f->fib_info->fib_dev != &loopback_dev || f->fib_dst != dev->pa_addr)) {
678                   fp = &f->fib_next;
679                   continue;
680             }
681             cli();
682             *fp = f->fib_next;
683             if (fib_loopback == f)
684                   fib_loopback = NULL;
685             sti();
686             fib_free_node(f);
687             found++;
688       }
689       return found;
690}
691
692static __inline__ void fib_flush_1(struct device *dev)
693{
694       struct fib_zone *fz;
695       int found = 0;
696
697       for (fz = fib_zone_list; fz; fz = fz->fz_next)
698       {
699             if (fz->fz_hash_table)
700             {
701                   int i;
702                   int tmp = 0;
703                   for (i=0; i<RTZ_HASH_DIVISOR; i++)
704                         tmp += rt_flush_list(&fz->fz_hash_table[i], dev);
705                   fz->fz_nent -= tmp;
706                   found += tmp;
707             }
708             else
709             {
710                   int tmp;
711                   tmp = rt_flush_list(&fz->fz_list, dev);
712                   fz->fz_nent -= tmp;
713                   found += tmp;
```

```
714             }
715         }
716
717         if (found)
718             rt_cache_flush();
719}
720
721
722/*
723 *      Called from the PROCfs module. This outputs /proc/net/route.
724 *
725 *      We preserve the old format but pad the buffers out. This means that
726 *      we can spin over the other entries as we read them. Remember the
727 *      gated BGP4 code could need to read 60,000+ routes on occasion (that's
728 *      about 7Mb of data). To do that ok we will need to also cache the
729 *      last route we got to (reads will generally be following on from
730 *      one another without gaps).
731 */
732
733int rt_get_info(char *buffer, char **start, off_t offset, int length, int dummy)
734{
735         struct fib_zone *fz;
736         struct fib_node *f;
737         int len=0;
738         off_t pos=0;
739         char temp[129];
740         int i;
741
742         pos = 128;
743
744         if (offset<128)
745         {
746             sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tMask\t\tMTU\tWindow\tIRTT");
747             len = 128;
748         }
749
750         while  (ip_rt_lock)
751             sleep_on(&rt_wait);
752         ip_rt_fast_lock();
753
754         for (fz=fib_zone_list; fz; fz = fz->fz_next)
755         {
756             int maxslot;
757             struct fib_node ** fp;
758
759             if (fz->fz_nent == 0)
760                 continue;
761
762             if (pos + 128*fz->fz_nent <= offset)
763             {
```

```
764                     pos += 128*fz->fz_nent;
765                     len = 0;
766                     continue;
767             }
768
769             if (fz->fz_hash_table)
770             {
771                     maxslot = RTZ_HASH_DIVISOR;
772                     fp      = fz->fz_hash_table;
773             }
774             else
775             {
776                     maxslot = 1;
777                     fp      = &fz->fz_list;
778             }
779
780             for (i=0; i < maxslot; i++, fp++)
781             {
782
783                     for (f = *fp; f; f = f->fib_next)
784                     {
785                             struct fib_info * fi;
786                             /*
787                              *      Spin through entries until we are ready
788                              */
789                             pos += 128;
790
791                             if (pos <= offset)
792                             {
793                                     len=0;
794                                     continue;
795                             }
796
797                             fi = f->fib_info;
798                             sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%d\t%lu\t%d\t%08lX\t%d\t%lu\t%u",
799                                     fi->fib_dev->name, (unsigned long)f->fib_dst, (unsigned long)fi-
>fib_gateway,
800                                     fi->fib_flags, 0, f->fib_use, f->fib_metric,
801                                     (unsigned long)fz->fz_mask, (int)fi->fib_mtu, fi->fib_window, (int)fi-
>fib_irtt);
802                             sprintf(buffer+len,"%-127s\n",temp);
803
804                             len += 128;
805                             if (pos >= offset+length)
806                                     goto done;
807                     }
808             }
809     }
810
811done:
```

```
812       ip_rt_unlock();
813       wake_up(&rt_wait);
814
815       *start = buffer+len-(pos-offset);
816       len = pos - offset;
817       if (len>length)
818             len = length;
819       return len;
820}
821
822int rt_cache_get_info(char *buffer, char **start, off_t offset, int length, int dummy)
823{
824       int len=0;
825       off_t pos=0;
826       char temp[129];
827       struct rtable *r;
828       int i;
829
830       pos = 128;
831
832       if (offset<128)
833       {
834             sprintf(buffer,"%-127s\n","Iface\tDestination\tGateway
\tFlags\tRefCnt\tUse\tMetric\tSource\t\tMTU\tWindow\tIRTT\tHH\tARP");
835             len = 128;
836       }
837
838
839       while  (ip_rt_lock)
840             sleep_on(&rt_wait);
841       ip_rt_fast_lock();
842
843       for (i = 0; i<RT_HASH_DIVISOR; i++)
844       {
845             for (r = ip_rt_hash_table[i]; r; r = r->rt_next)
846             {
847                   /*
848                    *      Spin through entries until we are ready
849                    */
850                   pos += 128;
851
852                   if (pos <= offset)
853                   {
854                         len = 0;
855                         continue;
856                   }
857
858                   sprintf(temp,
"%s\t%08lX\t%08lX\t%02X\t%d\t%u\t%d\t%08lX\t%d\t%lu\t%u\t%d\t%1d",
859                         r->rt_dev->name, (unsigned long)r->rt_dst, (unsigned long)r->rt_gateway,
860                         r->rt_flags, r->rt_refcnt, r->rt_use, 0,
```

```
 861                            (unsigned long)r->rt_src, (int)r->rt_mtu, r->rt_window, (int)r->rt_irtt, r->rt_hh ?
r->rt_hh->hh_refcnt : -1, r->rt_hh ? r->rt_hh->hh_uptodate : 0);
 862                    sprintf(buffer+len,"%-127s\n",temp);
 863                    len += 128;
 864                    if (pos >= offset+length)
 865                            goto done;
 866            }
 867    }
 868
 869done:
 870    ip_rt_unlock();
 871    wake_up(&rt_wait);
 872
 873    *start = buffer+len-(pos-offset);
 874    len = pos-offset;
 875    if (len>length)
 876            len = length;
 877    return len;
 878}
 879
 880
 881static void rt_free(struct rtable * rt)
 882{
 883    unsigned long flags;
 884
 885    save_flags(flags);
 886    cli();
 887    if (!rt->rt_refcnt)
 888    {
 889            struct hh_cache * hh = rt->rt_hh;
 890            rt->rt_hh = NULL;
 891            restore_flags(flags);
 892            if (hh && atomic_dec_and_test(&hh->hh_refcnt))
 893                    kfree_s(hh, sizeof(struct hh_cache));
 894            kfree_s(rt, sizeof(struct rt_table));
 895            return;
 896    }
 897    rt->rt_next = rt_free_queue;
 898    rt->rt_flags &= ~RTF_UP;
 899    rt_free_queue = rt;
 900    ip_rt_bh_mask |= RT_BH_FREE;
 901#if RT_CACHE_DEBUG >= 2
 902    printk("rt_free: %08x\n", rt->rt_dst);
 903#endif
 904    restore_flags(flags);
 905}
 906
 907/*
 908 * RT "bottom half" handlers. Called with masked interrupts.
 909 */
 910
```

```
911static __inline__ void rt_kick_free_queue(void)
912{
913     struct rtable *rt, **rtp;
914
915     rtp = &rt_free_queue;
916
917     while ((rt = *rtp) != NULL)
918     {
919          if  (!rt->rt_refcnt)
920          {
921               struct hh_cache * hh = rt->rt_hh;
922#if RT_CACHE_DEBUG >= 2
923               __u32 daddr = rt->rt_dst;
924#endif
925               *rtp = rt->rt_next;
926               rt->rt_hh = NULL;
927               sti();
928               if (hh && atomic_dec_and_test(&hh->hh_refcnt))
929                    kfree_s(hh, sizeof(struct hh_cache));
930               kfree_s(rt, sizeof(struct rt_table));
931#if RT_CACHE_DEBUG >= 2
932               printk("rt_kick_free_queue: %08x is free\n", daddr);
933#endif
934               cli();
935               continue;
936          }
937          rtp = &rt->rt_next;
938     }
939}
940
941void ip_rt_run_bh()
942{
943     unsigned long flags;
944     save_flags(flags);
945     cli();
946     if (ip_rt_bh_mask && !ip_rt_lock)
947     {
948          if (ip_rt_bh_mask & RT_BH_REDIRECT)
949               rt_kick_backlog();
950
951          if (ip_rt_bh_mask & RT_BH_GARBAGE_COLLECT)
952          {
953               ip_rt_fast_lock();
954               ip_rt_bh_mask &= ~RT_BH_GARBAGE_COLLECT;
955               sti();
956               rt_garbage_collect_1();
957               cli();
958               ip_rt_fast_unlock();
959          }
960
961          if (ip_rt_bh_mask & RT_BH_FREE)
```

```
962                     rt_kick_free_queue();
963             }
964             restore_flags(flags);
965}
966
967
968void ip_rt_check_expire()
969{
970             ip_rt_fast_lock();
971             if (ip_rt_lock == 1)
972             {
973                     int i;
974                     struct rtable *rth, **rthp;
975                     unsigned long flags;
976                     unsigned long now = jiffies;
977
978                     save_flags(flags);
979                     for (i=0; i<RT_HASH_DIVISOR; i++)
980                     {
981                             rthp = &ip_rt_hash_table[i];
982
983                             while ((rth = *rthp) != NULL)
984                             {
985                                     struct rtable * rth_next = rth->rt_next;
986
987                                     /*
988                                      * Cleanup aged off entries.
989                                      */
990
991                                     cli();
992                                     if (!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT < now)
993                                     {
994                                             *rthp = rth_next;
995                                             sti();
996                                             rt_cache_size--;
997#if RT_CACHE_DEBUG >= 2
998                                             printk("rt_check_expire clean %02x@%08x\n", i, rth->rt_dst);
999#endif
1000                                            rt_free(rth);
1001                                            continue;
1002                                    }
1003                                    sti();
1004
1005                                    if (!rth_next)
1006                                            break;
1007
1008                                    /*
1009                                     * LRU ordering.
1010                                     */
1011
```

```
1012                        if (rth->rt_lastuse + RT_CACHE_BUBBLE_THRESHOLD < rth_next-
>rt_lastuse ||
1013                           (rth->rt_lastuse < rth_next->rt_lastuse &&
1014                            rth->rt_use < rth_next->rt_use))
1015                        {
1016#if RT_CACHE_DEBUG >= 2
1017                                printk("rt_check_expire bubbled %02x@%08x<->%08x\n", i, rth->rt_dst,
rth_next->rt_dst);
1018#endif
1019                                cli();
1020                                *rthp = rth_next;
1021                                rth->rt_next = rth_next->rt_next;
1022                                rth_next->rt_next = rth;
1023                                sti();
1024                                rthp = &rth_next->rt_next;
1025                                continue;
1026                        }
1027                        rthp = &rth->rt_next;
1028                    }
1029            }
1030            restore_flags(flags);
1031            rt_kick_free_queue();
1032        }
1033        ip_rt_unlock();
1034}
1035
1036static void rt_redirect_1(__u32 dst, __u32 gw, struct device *dev)
1037{
1038        struct rtable *rt;
1039        unsigned long hash = ip_rt_hash_code(dst);
1040
1041        if (gw == dev->pa_addr)
1042                return;
1043        if (dev != get_gw_dev(gw))
1044                return;
1045        rt = (struct rtable *) kmalloc(sizeof(struct rtable), GFP_ATOMIC);
1046        if (rt == NULL)
1047                return;
1048        memset(rt, 0, sizeof(struct rtable));
1049        rt->rt_flags = RTF_DYNAMIC | RTF_MODIFIED | RTF_HOST | RTF_GATEWAY |
RTF_UP;
1050        rt->rt_dst = dst;
1051        rt->rt_dev = dev;
1052        rt->rt_gateway = gw;
1053        rt->rt_src = dev->pa_addr;
1054        rt->rt_mtu = dev->mtu;
1055#ifdef CONFIG_NO_PATH_MTU_DISCOVERY
1056        if (dev->mtu > 576)
1057                rt->rt_mtu = 576;
1058#endif
1059        rt->rt_lastuse  = jiffies;
```

```
1060        rt->rt_refcnt  = 1;
1061        rt_cache_add(hash, rt);
1062        ip_rt_put(rt);
1063        return;
1064}
1065
1066static void rt_cache_flush(void)
1067{
1068        int i;
1069        struct rtable * rth, * next;
1070
1071        for (i=0; i<RT_HASH_DIVISOR; i++)
1072        {
1073                int nr=0;
1074
1075                cli();
1076                if (!(rth = ip_rt_hash_table[i]))
1077                {
1078                        sti();
1079                        continue;
1080                }
1081
1082                ip_rt_hash_table[i] = NULL;
1083                sti();
1084
1085                for (; rth; rth=next)
1086                {
1087                        next = rth->rt_next;
1088                        rt_cache_size--;
1089                        nr++;
1090                        rth->rt_next = NULL;
1091                        rt_free(rth);
1092                }
1093#if RT_CACHE_DEBUG >= 2
1094                if (nr > 0)
1095                        printk("rt_cache_flush: %d@%02x\n", nr, i);
1096#endif
1097        }
1098#if RT_CACHE_DEBUG >= 1
1099        if (rt_cache_size)
1100        {
1101                printk("rt_cache_flush: bug rt_cache_size=%d\n", rt_cache_size);
1102                rt_cache_size = 0;
1103        }
1104#endif
1105}
1106
1107static void rt_garbage_collect_1(void)
1108{
1109        int i;
1110        unsigned expire = RT_CACHE_TIMEOUT>>1;
```

```
1111        struct rtable * rth, **rthp;
1112        unsigned long now = jiffies;
1113
1114        for (;;)
1115        {
1116                for (i=0; i<RT_HASH_DIVISOR; i++)
1117                {
1118                        if (!ip_rt_hash_table[i])
1119                                continue;
1120                        for (rthp=&ip_rt_hash_table[i]; (rth=*rthp); rthp=&rth->rt_next)
1121                        {
1122                                if (rth->rt_lastuse + expire*(rth->rt_refcnt+1) > now)
1123                                        continue;
1124                                rt_cache_size--;
1125                                cli();
1126                                *rthp=rth->rt_next;
1127                                rth->rt_next = NULL;
1128                                sti();
1129                                rt_free(rth);
1130                                break;
1131                        }
1132                }
1133                if (rt_cache_size < RT_CACHE_SIZE_MAX)
1134                        return;
1135                expire >>= 1;
1136        }
1137}
1138
1139static __inline__ void rt_req_enqueue(struct rt_req **q, struct rt_req *rtr)
1140{
1141        unsigned long flags;
1142        struct rt_req * tail;
1143
1144        save_flags(flags);
1145        cli();
1146        tail = *q;
1147        if (!tail)
1148                rtr->rtr_next = rtr;
1149        else
1150        {
1151                rtr->rtr_next = tail->rtr_next;
1152                tail->rtr_next = rtr;
1153        }
1154        *q = rtr;
1155        restore_flags(flags);
1156        return;
1157}
1158
1159/*
1160 * Caller should mask interrupts.
1161 */
```

```
1162
1163static __inline__ struct rt_req * rt_req_dequeue(struct rt_req **q)
1164{
1165        struct rt_req * rtr;
1166
1167        if (*q)
1168        {
1169                rtr = (*q)->rtr_next;
1170                (*q)->rtr_next = rtr->rtr_next;
1171                if (rtr->rtr_next == rtr)
1172                        *q = NULL;
1173                rtr->rtr_next = NULL;
1174                return rtr;
1175        }
1176        return NULL;
1177}
1178
1179/*
1180   Called with masked interrupts
1181 */
1182
1183static void rt_kick_backlog()
1184{
1185        if (!ip_rt_lock)
1186        {
1187                struct rt_req * rtr;
1188
1189                ip_rt_fast_lock();
1190
1191                while ((rtr = rt_req_dequeue(&rt_backlog)) != NULL)
1192                {
1193                        sti();
1194                        rt_redirect_1(rtr->dst, rtr->gw, rtr->dev);
1195                        kfree_s(rtr, sizeof(struct rt_req));
1196                        cli();
1197                }
1198
1199                ip_rt_bh_mask &= ~RT_BH_REDIRECT;
1200
1201                ip_rt_fast_unlock();
1202        }
1203}
1204
1205/*
1206 * rt_{del|add|flush} called only from USER process. Waiting is OK.
1207 */
1208
1209static int rt_del(__u32 dst, __u32 mask,
1210                struct device * dev, __u32 gtw, short rt_flags, short metric)
1211{
1212        int retval;
```

```
1213
1214        while (ip_rt_lock)
1215                sleep_on(&rt_wait);
1216        ip_rt_fast_lock();
1217        retval = fib_del_1(dst, mask, dev, gtw, rt_flags, metric);
1218        ip_rt_unlock();
1219        wake_up(&rt_wait);
1220        return retval;
1221}
1222
1223static void rt_add(short flags, __u32 dst, __u32 mask,
1224        __u32 gw, struct device *dev, unsigned short mss,
1225        unsigned long window, unsigned short irtt, short metric)
1226{
1227        while (ip_rt_lock)
1228                sleep_on(&rt_wait);
1229        ip_rt_fast_lock();
1230        fib_add_1(flags, dst, mask, gw, dev, mss, window, irtt, metric);
1231        ip_rt_unlock();
1232        wake_up(&rt_wait);
1233}
1234
1235void ip_rt_flush(struct device *dev)
1236{
1237        while (ip_rt_lock)
1238                sleep_on(&rt_wait);
1239        ip_rt_fast_lock();
1240        fib_flush_1(dev);
1241        ip_rt_unlock();
1242        wake_up(&rt_wait);
1243}
1244
1245/*
1246   Called by ICMP module.
1247 */
1248
1249void ip_rt_redirect(__u32 src, __u32 dst, __u32 gw, struct device *dev)
1250{
1251        struct rt_req * rtr;
1252        struct rtable * rt;
1253
1254        rt = ip_rt_route(dst, 0);
1255        if (!rt)
1256                return;
1257
1258        if (rt->rt_gateway != src ||
1259           rt->rt_dev != dev ||
1260           ((gw^dev->pa_addr)&dev->pa_mask) ||
1261           ip_chk_addr(gw))
1262        {
1263                ip_rt_put(rt);
```

```
1264            return;
1265    }
1266    ip_rt_put(rt);
1267
1268    ip_rt_fast_lock();
1269    if (ip_rt_lock == 1)
1270    {
1271            rt_redirect_1(dst, gw, dev);
1272            ip_rt_unlock();
1273            return;
1274    }
1275
1276    rtr = kmalloc(sizeof(struct rt_req), GFP_ATOMIC);
1277    if (rtr)
1278    {
1279            rtr->dst = dst;
1280            rtr->gw = gw;
1281            rtr->dev = dev;
1282            rt_req_enqueue(&rt_backlog, rtr);
1283            ip_rt_bh_mask |= RT_BH_REDIRECT;
1284    }
1285    ip_rt_unlock();
1286}
1287
1288
1289static __inline__ void rt_garbage_collect(void)
1290{
1291    if (ip_rt_lock == 1)
1292    {
1293            rt_garbage_collect_1();
1294            return;
1295    }
1296    ip_rt_bh_mask |= RT_BH_GARBAGE_COLLECT;
1297}
1298
1299static void rt_cache_add(unsigned hash, struct rtable * rth)
1300{
1301    unsigned long   flags;
1302    struct rtable   **rthp;
1303    __u32           daddr = rth->rt_dst;
1304    unsigned long   now = jiffies;
1305
1306#if RT_CACHE_DEBUG >= 2
1307    if (ip_rt_lock != 1)
1308    {
1309            printk("rt_cache_add: ip_rt_lock==%d\n", ip_rt_lock);
1310            return;
1311    }
1312#endif
1313
1314    save_flags(flags);
```

```
1315
1316        if (rth->rt_dev->header_cache_bind)
1317        {
1318                struct rtable * rtg = rth;
1319
1320                if (rth->rt_gateway != daddr)
1321                {
1322                        ip_rt_fast_unlock();
1323                        rtg = ip_rt_route(rth->rt_gateway, 0);
1324                        ip_rt_fast_lock();
1325                }
1326
1327                if (rtg)
1328                {
1329                        if (rtg == rth)
1330                                rtg->rt_dev->header_cache_bind(&rtg->rt_hh, rtg->rt_dev, ETH_P_IP, rtg-
>rt_dst);
1331                        else
1332                        {
1333                                if (rtg->rt_hh)
1334                                        atomic_inc(&rtg->rt_hh->hh_refcnt);
1335                                rth->rt_hh = rtg->rt_hh;
1336                                ip_rt_put(rtg);
1337                        }
1338                }
1339        }
1340
1341        if (rt_cache_size >= RT_CACHE_SIZE_MAX)
1342                rt_garbage_collect();
1343
1344        cli();
1345        rth->rt_next = ip_rt_hash_table[hash];
1346#if RT_CACHE_DEBUG >= 2
1347        if (rth->rt_next)
1348        {
1349                struct rtable * trth;
1350                printk("rt_cache @%02x: %08x", hash, daddr);
1351                for (trth=rth->rt_next; trth; trth=trth->rt_next)
1352                        printk(" . %08x", trth->rt_dst);
1353                printk("\n");
1354        }
1355#endif
1356        ip_rt_hash_table[hash] = rth;
1357        rthp = &rth->rt_next;
1358        sti();
1359        rt_cache_size++;
1360
1361        /*
1362         * Cleanup duplicate (and aged off) entries.
1363         */
1364
```

```
1365        while ((rth = *rthp) != NULL)
1366        {
1367
1368                cli();
1369                if ((!rth->rt_refcnt && rth->rt_lastuse + RT_CACHE_TIMEOUT < now)
1370                    || rth->rt_dst == daddr)
1371                {
1372                        *rthp = rth->rt_next;
1373                        rt_cache_size--;
1374                        sti();
1375#if RT_CACHE_DEBUG >= 2
1376                        printk("rt_cache clean %02x@%08x\n", hash, rth->rt_dst);
1377#endif
1378                        rt_free(rth);
1379                        continue;
1380                }
1381                sti();
1382                rthp = &rth->rt_next;
1383        }
1384        restore_flags(flags);
1385}
1386
1387/*
1388   RT should be already locked.
1389
1390   We could improve this by keeping a chain of say 32 struct rtable's
1391   last freed for fast recycling.
1392
1393 */
1394
1395struct rtable * ip_rt_slow_route (__u32 daddr, int local)
1396{
1397        unsigned hash = ip_rt_hash_code(daddr)^local;
1398        struct rtable * rth;
1399        struct fib_node * f;
1400        struct fib_info * fi;
1401        __u32 saddr;
1402
1403#if RT_CACHE_DEBUG >= 2
1404        printk("rt_cache miss @%08x\n", daddr);
1405#endif
1406
1407        rth = kmalloc(sizeof(struct rtable), GFP_ATOMIC);
1408        if (!rth)
1409        {
1410                ip_rt_unlock();
1411                return NULL;
1412        }
1413
1414        if (local)
1415                f = fib_lookup_local(daddr);
```

```
1416        else
1417                f = fib_lookup (daddr);
1418
1419        if (f)
1420        {
1421                fi = f->fib_info;
1422                f->fib_use++;
1423        }
1424
1425        if (!f || (fi->fib_flags & RTF_REJECT))
1426        {
1427#ifdef CONFIG_KERNELD
1428                char wanted_route[20];
1429#endif
1430#if RT_CACHE_DEBUG >= 2
1431                printk("rt_route failed @%08x\n", daddr);
1432#endif
1433                ip_rt_unlock();
1434                kfree_s(rth, sizeof(struct rtable));
1435#ifdef CONFIG_KERNELD
1436                daddr=ntohl(daddr);
1437                sprintf(wanted_route, "%d.%d.%d.%d",
1438                        (int)(daddr >> 24) & 0xff, (int)(daddr >> 16) & 0xff,
1439                        (int)(daddr >> 8) & 0xff, (int)daddr & 0xff);
1440                kerneld_route(wanted_route);    /* Dynamic route request */
1441#endif
1442                return NULL;
1443        }
1444
1445        saddr = fi->fib_dev->pa_addr;
1446
1447        if (daddr == fi->fib_dev->pa_addr)
1448        {
1449                f->fib_use--;
1450                if ((f = fib_loopback) != NULL)
1451                {
1452                        f->fib_use++;
1453                        fi = f->fib_info;
1454                }
1455        }
1456
1457        if (!f)
1458        {
1459                ip_rt_unlock();
1460                kfree_s(rth, sizeof(struct rtable));
1461                return NULL;
1462        }
1463
1464        rth->rt_dst     = daddr;
1465        rth->rt_src     = saddr;
1466        rth->rt_lastuse = jiffies;
```

```
1467        rth->rt_refcnt  = 1;
1468        rth->rt_use     = 1;
1469        rth->rt_next    = NULL;
1470        rth->rt_hh      = NULL;
1471        rth->rt_gateway = fi->fib_gateway;
1472        rth->rt_dev     = fi->fib_dev;
1473        rth->rt_mtu     = fi->fib_mtu;
1474        rth->rt_window  = fi->fib_window;
1475        rth->rt_irtt    = fi->fib_irtt;
1476        rth->rt_tos     = f->fib_tos;
1477        rth->rt_flags   = fi->fib_flags | RTF_HOST;
1478        if (local)
1479                rth->rt_flags   |= RTF_LOCAL;
1480
1481        if (!(rth->rt_flags & RTF_GATEWAY))
1482                rth->rt_gateway = rth->rt_dst;
1483        /*
1484         *      Multicast or limited broadcast is never gatewayed.
1485         */
1486        if (MULTICAST(daddr) || daddr == 0xFFFFFFFF)
1487                rth->rt_gateway = rth->rt_dst;
1488
1489        if (ip_rt_lock == 1)
1490                rt_cache_add(hash, rth);
1491        else
1492        {
1493                rt_free(rth);
1494#if RT_CACHE_DEBUG >= 1
1495                printk(KERN_DEBUG "rt_cache: route to %08x was born dead\n", daddr);
1496#endif
1497        }
1498
1499        ip_rt_unlock();
1500        return rth;
1501}
1502
1503void ip_rt_put(struct rtable * rt)
1504{
1505        if (rt)
1506                atomic_dec(&rt->rt_refcnt);
1507}
1508
1509struct rtable * ip_rt_route(__u32 daddr, int local)
1510{
1511        struct rtable * rth;
1512
1513        ip_rt_fast_lock();
1514
1515        for (rth=ip_rt_hash_table[ip_rt_hash_code(daddr)^local]; rth; rth=rth->rt_next)
1516        {
1517                if (rth->rt_dst == daddr)
```

```
1518            {
1519                    rth->rt_lastuse = jiffies;
1520                    atomic_inc(&rth->rt_use);
1521                    atomic_inc(&rth->rt_refcnt);
1522                    ip_rt_unlock();
1523                    return rth;
1524            }
1525    }
1526    return ip_rt_slow_route (daddr, local);
1527}
1528
1529/*
1530 *      Process a route add request from the user, or from a kernel
1531 *      task.
1532 */
1533
1534int ip_rt_new(struct rtentry *r)
1535{
1536    int err;
1537    char * devname;
1538    struct device * dev = NULL;
1539    unsigned long flags;
1540    __u32 daddr, mask, gw;
1541    short metric;
1542
1543    /*
1544     *      If a device is specified find it.
1545     */
1546
1547    if ((devname = r->rt_dev) != NULL)
1548    {
1549            err = getname(devname, &devname);
1550            if (err)
1551                    return err;
1552            dev = dev_get(devname);
1553            putname(devname);
1554            if (!dev)
1555                    return -ENODEV;
1556    }
1557
1558    /*
1559     *      If the device isn't INET, don't allow it
1560     */
1561
1562    if (r->rt_dst.sa_family != AF_INET)
1563            return -EAFNOSUPPORT;
1564
1565    /*
1566     *      Make local copies of the important bits
1567     *      We decrement the metric by one for BSD compatibility.
1568     */
```

```
1569
1570    flags = r->rt_flags;
1571    daddr = (__u32) ((struct sockaddr_in *) &r->rt_dst)->sin_addr.s_addr;
1572    mask  = (__u32) ((struct sockaddr_in *) &r->rt_genmask)->sin_addr.s_addr;
1573    gw    = (__u32) ((struct sockaddr_in *) &r->rt_gateway)->sin_addr.s_addr;
1574    metric = r->rt_metric > 0 ? r->rt_metric - 1 : 0;
1575
1576    /*
1577     *      BSD emulation: Permits route add someroute gw one-of-my-addresses
1578     *      to indicate which iface. Not as clean as the nice Linux dev technique
1579     *      but people keep using it...  (and gated likes it ;))
1580     */
1581
1582    if (!dev && (flags & RTF_GATEWAY))
1583    {
1584            struct device *dev2;
1585            for (dev2 = dev_base ; dev2 != NULL ; dev2 = dev2->next)
1586            {
1587                    if ((dev2->flags & IFF_UP) && dev2->pa_addr == gw)
1588                    {
1589                            flags &= ~RTF_GATEWAY;
1590                            dev = dev2;
1591                            break;
1592                    }
1593            }
1594    }
1595
1596    if (flags & RTF_HOST)
1597            mask = 0xffffffff;
1598    else if (mask && r->rt_genmask.sa_family != AF_INET)
1599            return -EAFNOSUPPORT;
1600
1601    if (flags & RTF_GATEWAY)
1602    {
1603            if (r->rt_gateway.sa_family != AF_INET)
1604                    return -EAFNOSUPPORT;
1605
1606            /*
1607             *      Don't try to add a gateway we can't reach..
1608             *      Tunnel devices are exempt from this rule.
1609             */
1610
1611            if (!dev)
1612                    dev = get_gw_dev(gw);
1613            else if (dev != get_gw_dev(gw) && dev->type != ARPHRD_TUNNEL)
1614                    return -EINVAL;
1615            if (!dev)
1616                    return -ENETUNREACH;
1617    }
1618    else
1619    {
```

```
1620            gw = 0;
1621            if (!dev)
1622                    dev = ip_dev_bynet(daddr, mask);
1623            if (!dev)
1624                    return -ENETUNREACH;
1625            if (!mask)
1626            {
1627                    if (((daddr ^ dev->pa_addr) & dev->pa_mask) == 0)
1628                            mask = dev->pa_mask;
1629            }
1630    }
1631
1632#ifndef CONFIG_IP_CLASSLESS
1633    if (!mask)
1634            mask = ip_get_mask(daddr);
1635#endif
1636
1637    if (bad_mask(mask, daddr))
1638            return -EINVAL;
1639
1640    /*
1641     *      Add the route
1642     */
1643
1644    rt_add(flags, daddr, mask, gw, dev, r->rt_mss, r->rt_window, r->rt_irtt, metric);
1645    return 0;
1646}
1647
1648
1649/*
1650 *      Remove a route, as requested by the user.
1651 */
1652
1653int ip_rt_kill(struct rtentry *r)
1654{
1655    struct sockaddr_in *trg;
1656    struct sockaddr_in *msk;
1657    struct sockaddr_in *gtw;
1658    char *devname;
1659    int err;
1660    struct device * dev = NULL;
1661
1662    trg = (struct sockaddr_in *) &r->rt_dst;
1663    msk = (struct sockaddr_in *) &r->rt_genmask;
1664    gtw = (struct sockaddr_in *) &r->rt_gateway;
1665    if ((devname = r->rt_dev) != NULL)
1666    {
1667            err = getname(devname, &devname);
1668            if (err)
1669                    return err;
1670            dev = dev_get(devname);
```

```
1671            putname(devname);
1672            if (!dev)
1673                    return -ENODEV;
1674    }
1675    /*
1676     * metric can become negative here if it wasn't filled in
1677     * but that's a fortunate accident; we really use that in rt_del.
1678     */
1679    err=rt_del((__u32)trg->sin_addr.s_addr, (__u32)msk->sin_addr.s_addr, dev,
1680            (__u32)gtw->sin_addr.s_addr, r->rt_flags, r->rt_metric - 1);
1681    return err;
1682}
1683
1684/*
1685 *      Handle IP routing ioctl calls. These are used to manipulate the routing tables
1686 */
1687
1688int ip_rt_ioctl(unsigned int cmd, void *arg)
1689{
1690    int err;
1691    struct rtentry rt;
1692
1693    switch(cmd)
1694    {
1695            case SIOCADDRT:         /* Add a route */
1696            case SIOCDELRT:         /* Delete a route */
1697                    if (!suser())
1698                            return -EPERM;
1699                    err=verify_area(VERIFY_READ, arg, sizeof(struct rtentry));
1700                    if (err)
1701                            return err;
1702                    memcpy_fromfs(&rt, arg, sizeof(struct rtentry));
1703                    return (cmd == SIOCDELRT) ? ip_rt_kill(&rt) : ip_rt_new(&rt);
1704    }
1705
1706    return -EINVAL;
1707}
1708
1709void ip_rt_advice(struct rtable **rp, int advice)
1710{
1711    /* Thanks! */
1712    return;
1713}
1714
1715void ip_rt_update(int event, struct device *dev)
1716{
1717/*
1718 *      This causes too much grief to do now.
1719 */
1720#ifdef COMING_IN_2_1
1721    if (event == NETDEV_UP)
```

```
1722                rt_add(RTF_HOST|RTF_UP, dev->pa_addr, ~0, 0, dev, 0, 0, 0, 0);
1723        else if (event == NETDEV_DOWN)
1724                rt_del(dev->pa_addr, ~0, dev, 0, RTF_HOST|RTF_UP, 0);
1725#endif
1726}
```

```
1/*
2 * INET        An implementation of the TCP/IP protocol suite for the LINUX
3 *             operating system.  INET  is implemented using the  BSD Socket
4 *             interface as the means of communication with the user level.
5 *
6 *             Definitions for the IP router.
7 *
8 * Version:     @(#)route.h     1.0.4   05/27/93
9 *
10 * Authors:     Ross Biro, <bir7@leland.Stanford.Edu>
11 *             Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12 * Fixes:
13 *             Alan Cox        :       Reformatted. Added ip_rt_local()
14 *             Alan Cox        :       Support for TCP parameters.
15 *             Alexey Kuznetsov:       Major changes for new routing code.
16 *
17 *      FIXME:
18 *             Make atomic ops more generic and hide them in asm/...
19 *
20 *             This program is free software; you can redistribute it and/or
21 *             modify it under the terms of the GNU General Public License
22 *             as published by the Free Software Foundation; either version
23 *             2 of the License, or (at your option) any later version.
24 */
25#ifndef _ROUTE_H
26#define _ROUTE_H
27
28#include <linux/config.h>
29
30/*
31 * 0 - no debugging messages
32 * 1 - rare events and bugs situations (default)
33 * 2 - trace mode.
34 */
35#define RT_CACHE_DEBUG          0
36
37#define RT_HASH_DIVISOR         256
38#define RT_CACHE_SIZE_MAX       256
39
40#define RTZ_HASH_DIVISOR        256
41
42#if RT_CACHE_DEBUG >= 2
43#define RTZ_HASHING_LIMIT 0
44#else
45#define RTZ_HASHING_LIMIT 16
46#endif
47
48/*
49 * Maximal time to live for unused entry.
50 */
51#define RT_CACHE_TIMEOUT                (HZ*300)
```

```
52
53/*
54 * Prevents LRU trashing, entries considered equivalent,
55 * if the difference between last use times is less then this number.
56 */
57#define RT_CACHE_BUBBLE_THRESHOLD       (HZ*5)
58
59#include <linux/route.h>
60
61#ifdef __KERNEL__
62#define RTF_LOCAL 0x8000
63#endif
64
65struct rtable
66{
67      struct rtable          *rt_next;
68      __u32                  rt_dst;
69      __u32                  rt_src;
70      __u32                  rt_gateway;
71      atomic_t               rt_refcnt;
72      atomic_t               rt_use;
73      unsigned long          rt_window;
74      atomic_t               rt_lastuse;
75      struct hh_cache        *rt_hh;
76      struct device          *rt_dev;
77      unsigned short         rt_flags;
78      unsigned short         rt_mtu;
79      unsigned short         rt_irtt;
80      unsigned char          rt_tos;
81};
82
83extern void            ip_rt_flush(struct device *dev);
84extern void            ip_rt_update(int event, struct device *dev);
85extern void            ip_rt_redirect(__u32 src, __u32 dst, __u32 gw, struct device *dev);
86extern struct rtable   *ip_rt_slow_route(__u32 daddr, int local);
87extern int             rt_get_info(char * buffer, char **start, off_t offset, int length, int dummy);
88extern int             rt_cache_get_info(char *buffer, char **start, off_t offset, int length, int dummy);
89extern int             ip_rt_ioctl(unsigned int cmd, void *arg);
90extern int             ip_rt_new(struct rtentry *rt);
91extern int             ip_rt_kill(struct rtentry *rt);
92extern void            ip_rt_check_expire(void);
93extern void            ip_rt_advice(struct rtable **rp, int advice);
94
95extern void            ip_rt_run_bh(void);
96extern atomic_t        ip_rt_lock;
97extern unsigned        ip_rt_bh_mask;
98extern struct rtable   *ip_rt_hash_table[RT_HASH_DIVISOR];
99
100extern __inline__ void ip_rt_fast_lock(void)
101{
102     atomic_inc(&ip_rt_lock);
```

```
103}
104
105extern __inline__ void ip_rt_fast_unlock(void)
106{
107	atomic_dec(&ip_rt_lock);
108}
109
110extern __inline__ void ip_rt_unlock(void)
111{
112	if (atomic_dec_and_test(&ip_rt_lock) && ip_rt_bh_mask)
113		ip_rt_run_bh();
114}
115
116extern __inline__ unsigned ip_rt_hash_code(__u32 addr)
117{
118	unsigned tmp = addr + (addr>>16);
119	return (tmp + (tmp>>8)) & 0xFF;
120}
121
122
123extern __inline__ void ip_rt_put(struct rtable * rt)
124#ifndef MODULE
125{
126	if (rt)
127		atomic_dec(&rt->rt_refcnt);
128}
129#else
130;
131#endif
132
133#ifdef CONFIG_KERNELD
134extern struct rtable * ip_rt_route(__u32 daddr, int local);
135#else
136extern __inline__ struct rtable * ip_rt_route(__u32 daddr, int local)
137#ifndef MODULE
138{
139	struct rtable * rth;
140
141	ip_rt_fast_lock();
142
143	for (rth=ip_rt_hash_table[ip_rt_hash_code(daddr)^local]; rth; rth=rth->rt_next)
144	{
145		if (rth->rt_dst == daddr)
146		{
147			rth->rt_lastuse = jiffies;
148			atomic_inc(&rth->rt_use);
149			atomic_inc(&rth->rt_refcnt);
150			ip_rt_unlock();
151			return rth;
152		}
153	}
```

```
154       return ip_rt_slow_route (daddr, local);
155}
156#else
157;
158#endif
159#endif
160
161extern __inline__ struct rtable * ip_check_route(struct rtable ** rp,
162                                                 __u32 daddr, int local)
163{
164       struct rtable * rt = *rp;
165
166       if (!rt || rt->rt_dst != daddr || !(rt->rt_flags&RTF_UP)
167           || ((local==1)^((rt->rt_flags&RTF_LOCAL) != 0)))
168       {
169             ip_rt_put(rt);
170             rt = ip_rt_route(daddr, local);
171             *rp = rt;
172       }
173       return rt;
174}
175
176
177#endif  /* _ROUTE_H */
```

From: Nathan Day
Sent: Tuesday, October 20, 2009 09:33 AM
To: kuznet@parallels.com
Subject: FW: contact request
Attachments: image001.png

Alexey,

My name is Nathan Day, I work for SoftLayer Technologies. We have been a Parallels/SWSoft reseller since we opened for business.

We need to ask for your help.

From the comments in the Linux kernel module /net/ipv4/route.c, it appears that you have submitted code to this part of the kernel.

A company is claiming that they own a patent that covers the rt_hash and rt_intern_hash functions in route.c. The patent is: http://www.google.com/patents/about?id=X4QXAAAAEBAJ.

Would it be possible to discuss this further with you? We are trying to find where the code came from to find out if there are code examples that existed before the patent was issued.

Thank you,

Nathan Day
CTO
nday@softlayer.com
214.442.0551 direct
972.989.7797 cell
866.398.7638 toll-free
214.442.0601 fax

SoftLayer Technologies, Inc.
6400 International Parkway, Suite 2000
Plano, TX 75093
http://www.softlayer.com

From: Scott Webber [mailto:swebber@parallels.com]
Sent: Monday, October 19, 2009 5:27 PM
To: Nathan Day
Cc: Alexey Kuznetsov; Greg Howard
Subject: RE: contact request

Hello Nathan,

Alexey is still a software developer for Parallels, I have copied him here.

At your service,

Scott

Scott Webber
Service Provider Engineering
||Parallels, Inc
660 SW 39th St, Suite 205
Renton, WA 98057
Office: (425) 282-1783

Mobile: (425) 988-4210
mailto:swebber@parallels.com
Skype:scott.webber.parallels
MS Live: scott.webber.parallels@live.com

http://www.parallels.com/spp/partnerprogram/

From: Greg Howard
Sent: Monday, October 19, 2009 3:10 PM
To: Nathan Day
Cc: Scott Webber
Subject: RE: contact request

Hi Nate, hope you are well. I am on the road today but Scott (lead VZ SE) copied here will see if Alexey is still on the team - if not, he can take any questions back to the Virtualization team in Russia.

All the best, G

From: Nathan Day [mailto:nday@softlayer.com]
Sent: Monday, October 19, 2009 4:44 PM
To: Greg Howard
Subject: contact request

Greg,

I have an odd request.

There is a gentleman named Alexey Kuznetsov who has been contributing to the Linux kernel for years. He is a Parallels employee (former SW-Soft) from some references we have seen. We have a specific linux kernel question about some code he worked on.

Could you see if he still works for Parallels and if so, put me in contact with him? Email is fine.

Thanks,

Nathan Day
CTO
nday@softlayer.com
214.442.0551 direct
972.989.7797 cell
866.398.7638 toll-free
214.442.0601 fax

SoftLayer Technologies, Inc.
6400 International Parkway, Suite 2000
Plano, TX 75093
http://www.softlayer.com

From: Alexey Kuznetsov
Sent: Tuesday, October 20, 2009 10:17 AM
To: Nathan Day
CC: kuznet@parallels.com
Subject: Re: FW: contact request

Hello!
> A company is claiming that they own a patent that covers the rt_hash and
> rt_intern_hash functions in route.c. The patent is:
> http://www.google.com/patents/about?id=X4QXAAAAEBAJ.
This is ridiculous. The patent is obviously invalid, hashing
cannot be covered by a patent, it is described in all the textbooks
since Computer Science emerged. :-)
Even formally the patent is filed on Jan 2, 1997. The code in net/ipv4/route.c
was written in 1995 (see linux kernel source archives) and got to its final
stable version in linux-2.0, which is dated by Jul 1996. Since that
time code was only optimized, all the history is linux repositories.

> We are trying to
> find where the code came from to find out if there are code examples
> that existed before the patent was issued.
1. Any textbook.
2. As I said above, the code itself was writen before the patent has been
   issued.
> Would it be possible to discuss this further with you?
No. I wrote tens of thousands of code during my life
and hope to write some more. :-) I definitely cannot devote
remaining years of my life to collect memories how I invented
that or another trivial thing. Probably later, when I will write
some memoires. :-)
Alexey

From: Alexey Kuznetsov [kuznet@ms2.inr.ac.ru]
Sent: Friday, December 03, 2010 5:58 AM
To: Absher, Alton
Cc: 'kuznet@parallels.com'
Subject: Re: Linux route.c question

Hello!

> If you are willing to help us, we are willing to pay your standard consulting rate for your time. Please let me know if you would be willing to speak with us. If so, please provide me with (1) your hourly rate, and (2) the best times to talk with us for about thirty minutes.

I have already spent time analyzing the patent 5893120 after Nathan Day of SoftLayer Technologies (who is probably one of your clients now) contacted me about year ago.

Is this the same case?

So, probably I can help. 1. For free. 2. By e-mail.

FYI I am afraid I am not a correct person to contact. My analysis showed that code written by me does not actually collide with forementioned patent, my code uses quite different techniques.
But current linux kernel actually contains logic which could be considered as infringing the patent: it was commited on Juanuary 2008 by Eric Dumazet <eric.dumazet@gmail.com>. (commit 29e75252da20f3ab9e132c68c9aed156b87beae6). Even though Eric wrote this piece, the idea was floating for ages, I thought it was either mine or David Miller's, we did not implement this earlier only because it was not considered enough important.
But unfortunately I could not find any references describing the idea before 1999, when the patent was issued. So, I must say the position can be difficult to defend.
I believe you should seek for an expert in loopholes of patent rules, the algorithm is still not direct replica of one described in the patent and expert could find a place to stand.

Alexey

From: Alexey Kuznetsov [kuznet@ms2.inr.ac.ru]
Sent: Friday, December 10, 2010 6:38 AM
To: Absher, Alton
Subject: Re: Linux route.c question

Hello!

> Thank you for returning my email. Yes, this is the same case. Can we have a very brief (10 minutes or less) telephone conversation first?

Well, OK.

But, first, I need some confirmation of your identity. I am not utterly paranoid, so that e-mail from any person at redhat.com would be enough.

Alexey

From: Absher, Alton
Sent: Wednesday, December 08, 2010 5:58 PM
To: 'Alexey Kuznetsov'
Subject: RE: Linux route.c question

Hello Alexey,

Thank you for returning my email. Yes, this is the same case. Can we have a very brief (10 minutes or less) telephone conversation first? Let me know a convenient time for you. I recognize that we are in different time zones, but I am available to talk at any time that is convenient for you.

Regards,
Alton

-----Original Message-----
From: Alexey Kuznetsov [mailto:kuznet@ms2.inr.ac.ru]
Sent: Friday, December 03, 2010 5:58 AM
To: Absher, Alton
Cc: 'kuznet@parallels.com'
Subject: Re: Linux route.c question

Hello!

> If you are willing to help us, we are willing to pay your standard consulting rate for your time. Please let me know if you would be willing to speak with us. If so, please provide me with (1) your hourly rate, and (2) the best times to talk with us for about thirty minutes.

I have already spent time analyzing the patent 5893120 after Nathan Day of SoftLayer Technologies (who is probably one of your clients now) contacted me about year ago.

Is this the same case?

So, probably I can help. 1. For free. 2. By e-mail.

FYI I am afraid I am not a correct person to contact. My analysis showed that code written by me does not actually collide with forementioned patent, my code uses quite different techniques.
But current linux kernel actually contains logic which could be considered as infringing the patent: it was commited on Juanuary 2008 by Eric Dumazet <eric.dumazet@gmail.com>. (commit 29e75252da20f3ab9e132c68c9aed156b87beae6). Even though Eric wrote this piece, the idea was floating for ages, I thought it was either mine or David Miller's, we did not implement this earlier only because it was not considered enough important.
But unfortunately I could not find any references describing the idea before 1999, when the patent was issued. So, I must say the position can be difficult to defend.
I believe you should seek for an expert in loopholes of patent rules, the algorithm is still not direct replica of one described in the patent and expert could find a place to stand.

Alexey

From: Absher, Alton
Sent: Thursday, December 02, 2010 2:26 PM
To: 'kuznet@parallels.com'
Subject: Linux route.c question

Dear Mr. Kuznetsov,

I am a patent attorney representing Red Hat and several of Red Hat's customers who have been sued for patent infringement based on code that you contributed to the Linux kernel. Specifically, they are alleging that the code that manages the Linux routing cache infringes a patent.

If you are willing to help us, we are willing to pay your standard consulting rate for your time. Please let me know if you would be willing to speak with us. If so, please provide me with (1) your hourly rate, and (2) the best times to talk with us for about thirty minutes.

Regards,
Alton Absher

Alton Absher
Kilpatrick Stockton LLP
1001 West Fourth Street | Winston-Salem, NC 27101-2400
office 336 607 7307 | cell 336 926 0211 | fax 336 734 2755
aabsher@kilpatrickstockton.com | My Profile

From: Alexey Kuznetsov [kuznet@ms2.inr.ac.ru]
Sent: Monday, December 13, 2010 4:45 AM
To: Absher, Alton
Subject: Re: Linux route.c question

Hello!

F.e. you may call today (Monday).

Phone: +7 (495) 7832977 ext. 70427

For me convenient time is 16:00 GMT (I assume you are in timezone GMT-5, so that this should be 11:00 for you)

Otherwise, we can schedule call for Wednesday, the same time.

Alexey

From: Alexey Kuznetsov [kuznet@ms2.inr.ac.ru]
Sent: Monday, December 13, 2010 9:03 AM
To: Absher, Alton
Subject: Re: Linux route.c question

Hello!

> Today at 16:00 GMT works for me.

OK.

> In connection with the call, please see the attached code and change log. We will briefly discuss the rt_cache_add() function.

Thanks. I did not even look so far behind. :-) Is not this enough to invalidate the patent?

Alexey

From: Absher, Alton
Sent: Monday, December 13, 2010 8:47 AM
To: 'Alexey Kuznetsov'
Subject: RE: Linux route.c question

Attachments: Linux 1.3.42 - route.c; Linux 1.3.42 - route.c - Nov. 17 1995 changelog.txt


Linux 1.3.42 - route.c (40 KB)...


Linux 1.3.42 - route.c - Nov. ...

Hello Alexey,

Today at 16:00 GMT works for me. In connection with the call, please see the attached code and change log. We will briefly discuss the rt_cache_add() function.

Regards,
Alton

Alton Absher
Kilpatrick Stockton LLP
1001 West Fourth Street | Winston-Salem, NC 27101-2400 office 336 607 7307 | cell 336 926 0211 | fax 336 734 2755 aabsher@kilpatrickstockton.com | www.kilpatrickstockton.com

-----Original Message-----
From: Alexey Kuznetsov [mailto:kuznet@ms2.inr.ac.ru]
Sent: Monday, December 13, 2010 4:45 AM
To: Absher, Alton
Subject: Re: Linux route.c question

Hello!

F.e. you may call today (Monday).

Phone: +7 (495) 7832977 ext. 70427

For me convenient time is 16:00 GMT (I assume you are in timezone GMT-5, so that this should be 11:00 for you)

Otherwise, we can schedule call for Wednesday, the same time.

Alexey

From: Absher, Alton
Sent: Tuesday, December 14, 2010 5:38 PM
To: Alexey Kuznetsov
Subject: route.c declaration
Attachments: Declaration-of-Alexey-Kuznetsov.pdf; Exhibit_A.pdf; Exhibit_B.pdf; Exhibit_C.pdf; Exhibit_D.pdf; Exhibit_E.pdf; Exhibit_F.pdf

Alexey,

It was nice speaking with you on Monday. As we discussed, I have drafted a declaration for you to review and sign. Please review the statements to confirm that you have personal knowledge that they are true. If you have questions, please let me know so that we can set up a call to discuss. If you have personal knowledge that the statements in the declaration are true, please sign it, and email me a signed copy.

I also need for you to mail (snail mail) me the original after you sign. If you have access to FedEx, I can give you a number to charge the shipping to (so that you don't have to spend any of your money to ship it). If FedEx is not convenient for you, let me know and we can make another arrangement.

Thank you again for your help.

Regards,
Alton

KILPATRICK STOCKTON LLP
Attorneys at Law

Alton Absher
Kilpatrick Stockton LLP
1001 West Fourth Street | Winston-Salem, NC 27101-2400
office 336 607 7307 | cell 336 926 0211 | fax 336 734 2755
aabsher@kilpatrickstockton.com | My Profile

From: Alexey Kuznetsov [kuznet@ms2.inr.ac.ru]
Sent: Wednesday, December 15, 2010 8:53 AM
To: Absher, Alton
Subject: Re: route.c declaration

Attachments: Document (1).pdf; Document (2).pdf


Document (1).pdf
(657 KB)


Document (2).pdf
(641 KB)

On Tue, Dec 14, 2010 at 05:38:03PM -0500, Absher, Alton wrote:

> Alexey,
>
> It was nice speaking with you on Monday. As we discussed, I have drafted a declaration for you to review and sign. Please review the statements to confirm that you have personal knowledge that they are true. If you have questions, please let me know so that we can set up a call to discuss. If you have personal knowledge that the statements in the declaration are true, please sign it, and email me a signed copy.

Everything is correct. Scans of two pages of signed document are enclosed.

> I also need for you to mail (snail mail) me the original after you sign. If you have access to FedEx, I can give you a number to charge the shipping to (so that you don't have to spend any of your money to ship it). If FedEx is not convenient for you, let me know and we can make another arrangement.

Seems, fedex is OK.

Alexey

From: Absher, Alton
Sent: Wednesday, December 15, 2010 9:01 AM
To: 'Alexey Kuznetsov'
Subject: RE: route.c declaration

Thanks Alexey. Our FedEx number is 027406777. Please send the signed original to me at the address below:

Alton Absher
Kilpatrick Stockton LLP
1001 West Fourth Street
Winston-Salem, NC 27101-2400
United States of America

-----Original Message-----
From: Alexey Kuznetsov [mailto:kuznet@ms2.inr.ac.ru]
Sent: Wednesday, December 15, 2010 8:53 AM
To: Absher, Alton
Subject: Re: route.c declaration

On Tue, Dec 14, 2010 at 05:38:03PM -0500, Absher, Alton wrote:
> Alexey,
>
> It was nice speaking with you on Monday. As we discussed, I have drafted a declaration for you to review and sign. Please review the statements to confirm that you have personal knowledge that they are true. If you have questions, please let me know so that we can set up a call to discuss. If you have personal knowledge that the statements in the declaration are true, please sign it, and email me a signed copy.

Everything is correct. Scans of two pages of signed document are enclosed.

> I also need for you to mail (snail mail) me the original after you sign. If you have access to FedEx, I can give you a number to charge the shipping to (so that you don't have to spend any of your money to ship it). If FedEx is not convenient for you, let me know and we can make another arrangement.

Seems, fedex is OK.

Alexey

