Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version: 2.0.40 2.2.26 2.4.37 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 3.17 3.18 3.19 4.0 4.1 *4.2*

Linux/net/ipv4/route.c

```
/*
 * 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.
 *
 * Authors:     Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Alan Cox, <gw4pts@gw4pts.ampr.org>
 *              Linus Torvalds, <Linus.Torvalds@helsinki.fi>
 *              Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
 *
 * 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 wasn't being copied right.
 *              Bjorn Ekwall    :       Kerneld route support.
 *              Alan Cox        :       Multicast fixed (I hope)
 *              Pavel Krauz     :       Limited broadcast fixed
 *              Mike McLagan    :       Routing by source
 *      Alexey Kuznetsov        :       End of old history. Split to fib.c and
 *                                      route.c and rewritten from scratch.
 *              Andi Kleen      :       Load-limit warning messages.
 *      Vitaly E. Lavrov        :       Transparent proxy revived after year coma.
 *      Vitaly E. Lavrov        :       Race condition in ip_route_input_slow.
 *      Tobias Ringstrom        :       Uninitialized res.type in ip_route_output_slow.
 *      Vladimir V. Ivanov      :       IP rule info (flowid) is really useful.
 *              Marc Boucher    :       routing by fwmark
 *      Robert Olsson           :       Added rt_cache statistics
 *      Arnaldo C. Melo         :       Convert proc stuff to seq_file
 *      Eric Dumazet            :       hashed spinlocks and rt_check_expire() fixes.
 *      Ilia Sotnikov           :       Ignore TOS on PMTUD and Redirect
 *      Ilia Sotnikov           :       Removed TOS from hash calculations
 *
 *              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.
 */

#define pr_fmt(fmt) "IPv4: " fmt

#include <linux/module.h>
#include <asm/uaccess.h>
#include <linux/bitops.h>
#include <linux/types.h>
#include <linux/kernel.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 <linux/proc_fs.h>
#include <linux/init.h>
#include <linux/skbuff.h>
#include <linux/inetdevice.h>
#include <linux/igmp.h>
```

```
 85 #include <linux/pkt_sched.h>
 86 #include <linux/mroute.h>
 87 #include <linux/netfilter_ipv4.h>
 88 #include <linux/random.h>
 89 #include <linux/rcupdate.h>
 90 #include <linux/times.h>
 91 #include <linux/slab.h>
 92 #include <linux/jhash.h>
 93 #include <net/dst.h>
 94 #include <net/net_namespace.h>
 95 #include <net/protocol.h>
 96 #include <net/ip.h>
 97 #include <net/route.h>
 98 #include <net/inetpeer.h>
 99 #include <net/sock.h>
100 #include <net/ip_fib.h>
101 #include <net/arp.h>
102 #include <net/tcp.h>
103 #include <net/icmp.h>
104 #include <net/xfrm.h>
105 #include <net/netevent.h>
106 #include <net/rtnetlink.h>
107 #ifdef CONFIG_SYSCTL
108 #include <linux/sysctl.h>
109 #include <linux/kmemleak.h>
110 #endif
111 #include <net/secure_seq.h>
112
113 #define RT_FL_TOS(oldflp4) \
114         ((oldflp4)->flowi4_tos & (IPTOS_RT_MASK | RTO_ONLINK))
115
116 #define RT_GC_TIMEOUT (300*HZ)
117
118 static int ip_rt_max_size;
119 static int ip_rt_redirect_number __read_mostly  = 9;
120 static int ip_rt_redirect_load __read_mostly    = HZ / 50;
121 static int ip_rt_redirect_silence __read_mostly = ((HZ / 50) << (9 + 1));
122 static int ip_rt_error_cost __read_mostly       = HZ;
123 static int ip_rt_error_burst __read_mostly      = 5 * HZ;
124 static int ip_rt_mtu_expires __read_mostly      = 10 * 60 * HZ;
125 static int ip_rt_min_pmtu __read_mostly         = 512 + 20 + 20;
126 static int ip_rt_min_advmss __read_mostly       = 256;
127
128 /*
129  *      Interface to generic destination cache.
130  */
131
132 static struct dst_entry *ipv4_dst_check(struct dst_entry *dst, u32 cookie);
133 static unsigned int      ipv4_default_advmss(const struct dst_entry *dst);
134 static unsigned int      ipv4_mtu(const struct dst_entry *dst);
135 static struct dst_entry *ipv4_negative_advice(struct dst_entry *dst);
136 static void              ipv4_link_failure(struct sk_buff *skb);
137 static void              ip_rt_update_pmtu(struct dst_entry *dst, struct sock *sk,
138                                            struct sk_buff *skb, u32 mtu);
139 static void              ip_do_redirect(struct dst_entry *dst, struct sock *sk,
140                                         struct sk_buff *skb);
141 static void             ipv4_dst_destroy(struct dst_entry *dst);
142
143 static u32 *ipv4_cow_metrics(struct dst_entry *dst, unsigned long old)
144 {
145         WARN_ON(1);
146         return NULL;
147 }
148
149 static struct neighbour *ipv4_neigh_lookup(const struct dst_entry *dst,
150                                            struct sk_buff *skb,
151                                            const void *daddr);
152
153 static struct dst_ops ipv4_dst_ops = {
154         .family =               AF_INET,
155         .check =                ipv4_dst_check,
156         .default_advmss =       ipv4_default_advmss,
157         .mtu =                  ipv4_mtu,
158         .cow_metrics =          ipv4_cow_metrics,
159         .destroy =              ipv4_dst_destroy,
160         .negative_advice =      ipv4_negative_advice,
161         .link_failure =         ipv4_link_failure,
162         .update_pmtu =          ip_rt_update_pmtu,
163         .redirect =             ip_do_redirect,
164         .local_out =            __ip_local_out,
165         .neigh_lookup =         ipv4_neigh_lookup,
166 };
167
168 #define ECN_OR_COST(class)      TC_PRIO_##class
169
170 const __u8 ip_tos2prio[16] = {
171         TC_PRIO_BESTEFFORT,
172         ECN_OR_COST(BESTEFFORT),
173         TC_PRIO_BESTEFFORT,
174         ECN_OR_COST(BESTEFFORT),
175         TC_PRIO_BULK,
176         ECN_OR_COST(BULK),
177         TC_PRIO_BULK,
178         ECN_OR_COST(BULK),
179         TC_PRIO_INTERACTIVE,
180         ECN_OR_COST(INTERACTIVE),
181         TC_PRIO_INTERACTIVE,
182         ECN_OR_COST(INTERACTIVE),
183         TC_PRIO_INTERACTIVE_BULK,
184         ECN_OR_COST(INTERACTIVE_BULK),
185         TC_PRIO_INTERACTIVE_BULK,
186         ECN_OR_COST(INTERACTIVE_BULK)
187 };
188 EXPORT_SYMBOL(ip_tos2prio);
```

```
189
190 static DEFINE_PER_CPU(struct rt_cache_stat, rt_cache_stat);
191 #define RT_CACHE_STAT_INC(field) raw_cpu_inc(rt_cache_stat.field)
192
193 #ifdef CONFIG_PROC_FS
194 static void *rt_cache_seq_start(struct seq_file *seq, loff_t *pos)
195 {
196         if (*pos)
197                 return NULL;
198         return SEQ_START_TOKEN;
199 }
200
201 static void *rt_cache_seq_next(struct seq_file *seq, void *v, loff_t *pos)
202 {
203         ++*pos;
204         return NULL;
205 }
206
207 static void rt_cache_seq_stop(struct seq_file *seq, void *v)
208 {
209 }
210
211 static int rt_cache_seq_show(struct seq_file *seq, void *v)
212 {
213         if (v == SEQ_START_TOKEN)
214                 seq_printf(seq, "%-127s\n",
215                            "Iface\tDestination\tGateway \tFlags\t\tRefCnt\tUse\t"
216                            "Metric\tSource\t\tMTU\tWindow\tIRTT\tTOS\tHHRef\t"
217                            "HHUptod\tSpecDst");
218         return 0;
219 }
220
221 static const struct seq_operations rt_cache_seq_ops = {
222         .start  = rt_cache_seq_start,
223         .next   = rt_cache_seq_next,
224         .stop   = rt_cache_seq_stop,
225         .show   = rt_cache_seq_show,
226 };
227
228 static int rt_cache_seq_open(struct inode *inode, struct file *file)
229 {
230         return seq_open(file, &rt_cache_seq_ops);
231 }
232
233 static const struct file_operations rt_cache_seq_fops = {
234         .owner   = THIS_MODULE,
235         .open    = rt_cache_seq_open,
236         .read    = seq_read,
237         .llseek  = seq_lseek,
238         .release = seq_release,
239 };
240
241
242 static void *rt_cpu_seq_start(struct seq_file *seq, loff_t *pos)
243 {
244         int cpu;
245
246         if (*pos == 0)
247                 return SEQ_START_TOKEN;
248
249         for (cpu = *pos-1; cpu < nr_cpu_ids; ++cpu) {
250                 if (!cpu_possible(cpu))
251                         continue;
252                 *pos = cpu+1;
253                 return &per_cpu(rt_cache_stat, cpu);
254         }
255         return NULL;
256 }
257
258 static void *rt_cpu_seq_next(struct seq_file *seq, void *v, loff_t *pos)
259 {
260         int cpu;
261
262         for (cpu = *pos; cpu < nr_cpu_ids; ++cpu) {
263                 if (!cpu_possible(cpu))
264                         continue;
265                 *pos = cpu+1;
266                 return &per_cpu(rt_cache_stat, cpu);
267         }
268         return NULL;
269
270 }
271
272 static void rt_cpu_seq_stop(struct seq_file *seq, void *v)
273 {
274
275 }
276
277 static int rt_cpu_seq_show(struct seq_file *seq, void *v)
278 {
279         struct rt_cache_stat *st = v;
280
281         if (v == SEQ_START_TOKEN) {
282                 seq_printf(seq, "entries  in_hit in_slow_tot in_slow_mc in_no_route in_brd in_martian_dst in_martian_src  out_hit out_slow_tot out
283                 return 0;
284         }
285
286         seq_printf(seq,"%08x  %08x %08x %08x %08x %08x %08x %08x "
287                    " %08x %08x %08x %08x %08x %08x %08x %08x %08x \n",
288                    dst_entries_get_slow(&ipv4_dst_ops),
289                    0, /* st->in_hit */
290                    st->in_slow_tot,
291                    st->in_slow_mc,
292                    st->in_no_route,
```

```
293                     st->in_brd,
294                     st->in_martian_dst,
295                     st->in_martian_src,
296
297                     0, /* st->out_hit */
298                     st->out_slow_tot,
299                     st->out_slow_mc,
300
301                     0, /* st->gc_total */
302                     0, /* st->gc_ignored */
303                     0, /* st->gc_goal_miss */
304                     0, /* st->gc_dst_overflow */
305                     0, /* st->in_hlist_search */
306                     0  /* st->out_hlist_search */
307                 );
308         return 0;
309 }
310
311 static const struct seq_operations rt_cpu_seq_ops = {
312         .start  = rt_cpu_seq_start,
313         .next   = rt_cpu_seq_next,
314         .stop   = rt_cpu_seq_stop,
315         .show   = rt_cpu_seq_show,
316 };
317
318
319 static int rt_cpu_seq_open(struct inode *inode, struct file *file)
320 {
321         return seq_open(file, &rt_cpu_seq_ops);
322 }
323
324 static const struct file_operations rt_cpu_seq_fops = {
325         .owner   = THIS_MODULE,
326         .open    = rt_cpu_seq_open,
327         .read    = seq_read,
328         .llseek  = seq_lseek,
329         .release = seq_release,
330 };
331
332 #ifdef CONFIG_IP_ROUTE_CLASSID
333 static int rt_acct_proc_show(struct seq_file *m, void *v)
334 {
335         struct ip_rt_acct *dst, *src;
336         unsigned int i, j;
337
338         dst = kcalloc(256, sizeof(struct ip_rt_acct), GFP_KERNEL);
339         if (!dst)
340                 return -ENOMEM;
341
342         for_each_possible_cpu(i) {
343                 src = (struct ip_rt_acct *)per_cpu_ptr(ip_rt_acct, i);
344                 for (j = 0; j < 256; j++) {
345                         dst[j].o_bytes   += src[j].o_bytes;
346                         dst[j].o_packets += src[j].o_packets;
347                         dst[j].i_bytes   += src[j].i_bytes;
348                         dst[j].i_packets += src[j].i_packets;
349                 }
350         }
351
352         seq_write(m, dst, 256 * sizeof(struct ip_rt_acct));
353         kfree(dst);
354         return 0;
355 }
356
357 static int rt_acct_proc_open(struct inode *inode, struct file *file)
358 {
359         return single_open(file, rt_acct_proc_show, NULL);
360 }
361
362 static const struct file_operations rt_acct_proc_fops = {
363         .owner          = THIS_MODULE,
364         .open           = rt_acct_proc_open,
365         .read           = seq_read,
366         .llseek         = seq_lseek,
367         .release        = single_release,
368 };
369 #endif
370
371 static int __net_init ip_rt_do_proc_init(struct net *net)
372 {
373         struct proc_dir_entry *pde;
374
375         pde = proc_create("rt_cache", S_IRUGO, net->proc_net,
376                           &rt_cache_seq_fops);
377         if (!pde)
378                 goto err1;
379
380         pde = proc_create("rt_cache", S_IRUGO,
381                           net->proc_net_stat, &rt_cpu_seq_fops);
382         if (!pde)
383                 goto err2;
384
385 #ifdef CONFIG_IP_ROUTE_CLASSID
386         pde = proc_create("rt_acct", 0, net->proc_net, &rt_acct_proc_fops);
387         if (!pde)
388                 goto err3;
389 #endif
390         return 0;
391
392 #ifdef CONFIG_IP_ROUTE_CLASSID
393 err3:
394         remove_proc_entry("rt_cache", net->proc_net_stat);
395 #endif
396 err2:
```

```
397         remove_proc_entry("rt_cache", net->proc_net);
398 err1:
399         return -ENOMEM;
400 }
401
402 static void __net_exit ip_rt_do_proc_exit(struct net *net)
403 {
404         remove_proc_entry("rt_cache", net->proc_net_stat);
405         remove_proc_entry("rt_cache", net->proc_net);
406 #ifdef CONFIG_IP_ROUTE_CLASSID
407         remove_proc_entry("rt_acct", net->proc_net);
408 #endif
409 }
410
411 static struct pernet_operations ip_rt_proc_ops __net_initdata =  {
412         .init = ip_rt_do_proc_init,
413         .exit = ip_rt_do_proc_exit,
414 };
415
416 static int __init ip_rt_proc_init(void)
417 {
418         return register_pernet_subsys(&ip_rt_proc_ops);
419 }
420
421 #else
422 static inline int ip_rt_proc_init(void)
423 {
424         return 0;
425 }
426 #endif /* CONFIG_PROC_FS */
427
428 static inline bool rt_is_expired(const struct rtable *rth)
429 {
430         return rth->rt_genid != rt_genid_ipv4(dev_net(rth->dst.dev));
431 }
432
433 void rt_cache_flush(struct net *net)
434 {
435         rt_genid_bump_ipv4(net);
436 }
437
438 static struct neighbour *ipv4_neigh_lookup(const struct dst_entry *dst,
439                                            struct sk_buff *skb,
440                                            const void *daddr)
441 {
442         struct net_device *dev = dst->dev;
443         const __be32 *pkey = daddr;
444         const struct rtable *rt;
445         struct neighbour *n;
446
447         rt = (const struct rtable *) dst;
448         if (rt->rt_gateway)
449                 pkey = (const __be32 *) &rt->rt_gateway;
450         else if (skb)
451                 pkey = &ip_hdr(skb)->daddr;
452
453         n = __ipv4_neigh_lookup(dev, *(__force u32 *)pkey);
454         if (n)
455                 return n;
456         return neigh_create(&arp_tbl, pkey, dev);
457 }
458
459 #define IP_IDENTS_SZ 2048u
460
461 static atomic_t *ip_idents __read_mostly;
462 static u32 *ip_tstamps __read_mostly;
463
464 /* In order to protect privacy, we add a perturbation to identifiers
465  * if one generator is seldom used. This makes hard for an attacker
466  * to infer how many packets were sent between two points in time.
467  */
468 u32 ip_idents_reserve(u32 hash, int segs)
469 {
470         u32 *p_tstamp = ip_tstamps + hash % IP_IDENTS_SZ;
471         atomic_t *p_id = ip_idents + hash % IP_IDENTS_SZ;
472         u32 old = ACCESS_ONCE(*p_tstamp);
473         u32 now = (u32)jiffies;
474         u32 delta = 0;
475
476         if (old != now && cmpxchg(p_tstamp, old, now) == old)
477                 delta = prandom_u32_max(now - old);
478
479         return atomic_add_return(segs + delta, p_id) - segs;
480 }
481 EXPORT_SYMBOL(ip_idents_reserve);
482
483 void __ip_select_ident(struct net *net, struct iphdr *iph, int segs)
484 {
485         static u32 ip_idents_hashrnd __read_mostly;
486         u32 hash, id;
487
488         net_get_random_once(&ip_idents_hashrnd, sizeof(ip_idents_hashrnd));
489
490         hash = jhash_3words((__force u32)iph->daddr,
491                             (__force u32)iph->saddr,
492                             iph->protocol ^ net_hash_mix(net),
493                             ip_idents_hashrnd);
494         id = ip_idents_reserve(hash, segs);
495         iph->id = htons(id);
496 }
497 EXPORT_SYMBOL(__ip_select_ident);
498
499 static void __build_flow_key(struct flowi4 *fl4, const struct sock *sk,
500                              const struct iphdr *iph,
```

```
                                int oif, u8 tos,
                                u8 prot, u32 mark, int flow_flags)
{
        if (sk) {
                const struct inet_sock *inet = inet_sk(sk);

                oif = sk->sk_bound_dev_if;
                mark = sk->sk_mark;
                tos = RT_CONN_FLAGS(sk);
                prot = inet->hdrincl ? IPPROTO_RAW : sk->sk_protocol;
        }
        flowi4_init_output(fl4, oif, mark, tos,
                           RT_SCOPE_UNIVERSE, prot,
                           flow_flags,
                           iph->daddr, iph->saddr, 0, 0);
}

static void build_skb_flow_key(struct flowi4 *fl4, const struct sk_buff *skb,
                               const struct sock *sk)
{
        const struct iphdr *iph = ip_hdr(skb);
        int oif = skb->dev->ifindex;
        u8 tos = RT_TOS(iph->tos);
        u8 prot = iph->protocol;
        u32 mark = skb->mark;

        __build_flow_key(fl4, sk, iph, oif, tos, prot, mark, 0);
}

static void build_sk_flow_key(struct flowi4 *fl4, const struct sock *sk)
{
        const struct inet_sock *inet = inet_sk(sk);
        const struct ip_options_rcu *inet_opt;
        __be32 daddr = inet->inet_daddr;

        rcu_read_lock();
        inet_opt = rcu_dereference(inet->inet_opt);
        if (inet_opt && inet_opt->opt.srr)
                daddr = inet_opt->opt.faddr;
        flowi4_init_output(fl4, sk->sk_bound_dev_if, sk->sk_mark,
                           RT_CONN_FLAGS(sk), RT_SCOPE_UNIVERSE,
                           inet->hdrincl ? IPPROTO_RAW : sk->sk_protocol,
                           inet_sk_flowi_flags(sk),
                           daddr, inet->inet_saddr, 0, 0);
        rcu_read_unlock();
}

static void ip_rt_build_flow_key(struct flowi4 *fl4, const struct sock *sk,
                                 const struct sk_buff *skb)
{
        if (skb)
                build_skb_flow_key(fl4, skb, sk);
        else
                build_sk_flow_key(fl4, sk);
}

static inline void rt_free(struct rtable *rt)
{
        call_rcu(&rt->dst.rcu_head, dst_rcu_free);
}

static DEFINE_SPINLOCK(fnhe_lock);

static void fnhe_flush_routes(struct fib_nh_exception *fnhe)
{
        struct rtable *rt;

        rt = rcu_dereference(fnhe->fnhe_rth_input);
        if (rt) {
                RCU_INIT_POINTER(fnhe->fnhe_rth_input, NULL);
                rt_free(rt);
        }
        rt = rcu_dereference(fnhe->fnhe_rth_output);
        if (rt) {
                RCU_INIT_POINTER(fnhe->fnhe_rth_output, NULL);
                rt_free(rt);
        }
}

static struct fib_nh_exception *fnhe_oldest(struct fnhe_hash_bucket *hash)
{
        struct fib_nh_exception *fnhe, *oldest;

        oldest = rcu_dereference(hash->chain);
        for (fnhe = rcu_dereference(oldest->fnhe_next); fnhe;
             fnhe = rcu_dereference(fnhe->fnhe_next)) {
                if (time_before(fnhe->fnhe_stamp, oldest->fnhe_stamp))
                        oldest = fnhe;
        }
        fnhe_flush_routes(oldest);
        return oldest;
}

static inline u32 fnhe_hashfun(__be32 daddr)
{
        static u32 fnhe_hashrnd __read_mostly;
        u32 hval;

        net_get_random_once(&fnhe_hashrnd, sizeof(fnhe_hashrnd));
        hval = jhash_1word((__force u32) daddr, fnhe_hashrnd);
        return hash_32(hval, FNHE_HASH_SHIFT);
}

static void fill_route_from_fnhe(struct rtable *rt, struct fib_nh_exception *fnhe)
```

```
605 {
606         rt->rt_pmtu = fnhe->fnhe_pmtu;
607         rt->dst.expires = fnhe->fnhe_expires;
608
609         if (fnhe->fnhe_gw) {
610                 rt->rt_flags |= RTCF_REDIRECTED;
611                 rt->rt_gateway = fnhe->fnhe_gw;
612                 rt->rt_uses_gateway = 1;
613         }
614 }
615
616 static void update_or_create_fnhe(struct fib_nh *nh, __be32 daddr, __be32 gw,
617                                   u32 pmtu, unsigned long expires)
618 {
619         struct fnhe_hash_bucket *hash;
620         struct fib_nh_exception *fnhe;
621         struct rtable *rt;
622         unsigned int i;
623         int depth;
624         u32 hval = fnhe_hashfun(daddr);
625
626         spin_lock_bh(&fnhe_lock);
627
628         hash = rcu_dereference(nh->nh_exceptions);
629         if (!hash) {
630                 hash = kzalloc(FNHE_HASH_SIZE * sizeof(*hash), GFP_ATOMIC);
631                 if (!hash)
632                         goto out_unlock;
633                 rcu_assign_pointer(nh->nh_exceptions, hash);
634         }
635
636         hash += hval;
637
638         depth = 0;
639         for (fnhe = rcu_dereference(hash->chain); fnhe;
640              fnhe = rcu_dereference(fnhe->fnhe_next)) {
641                 if (fnhe->fnhe_daddr == daddr)
642                         break;
643                 depth++;
644         }
645
646         if (fnhe) {
647                 if (gw)
648                         fnhe->fnhe_gw = gw;
649                 if (pmtu) {
650                         fnhe->fnhe_pmtu = pmtu;
651                         fnhe->fnhe_expires = max(1UL, expires);
652                 }
653                 /* Update all cached dsts too */
654                 rt = rcu_dereference(fnhe->fnhe_rth_input);
655                 if (rt)
656                         fill_route_from_fnhe(rt, fnhe);
657                 rt = rcu_dereference(fnhe->fnhe_rth_output);
658                 if (rt)
659                         fill_route_from_fnhe(rt, fnhe);
660         } else {
661                 if (depth > FNHE_RECLAIM_DEPTH)
662                         fnhe = fnhe_oldest(hash);
663                 else {
664                         fnhe = kzalloc(sizeof(*fnhe), GFP_ATOMIC);
665                         if (!fnhe)
666                                 goto out_unlock;
667
668                         fnhe->fnhe_next = hash->chain;
669                         rcu_assign_pointer(hash->chain, fnhe);
670                 }
671                 fnhe->fnhe_genid = fnhe_genid(dev_net(nh->nh_dev));
672                 fnhe->fnhe_daddr = daddr;
673                 fnhe->fnhe_gw = gw;
674                 fnhe->fnhe_pmtu = pmtu;
675                 fnhe->fnhe_expires = expires;
676
677                 /* Exception created; mark the cached routes for the nexthop
678                  * stale, so anyone caching it rechecks if this exception
679                  * applies to them.
680                  */
681                 rt = rcu_dereference(nh->nh_rth_input);
682                 if (rt)
683                         rt->dst.obsolete = DST_OBSOLETE_KILL;
684
685                 for_each_possible_cpu(i) {
686                         struct rtable __rcu **prt;
687                         prt = per_cpu_ptr(nh->nh_pcpu_rth_output, i);
688                         rt = rcu_dereference(*prt);
689                         if (rt)
690                                 rt->dst.obsolete = DST_OBSOLETE_KILL;
691                 }
692         }
693
694         fnhe->fnhe_stamp = jiffies;
695
696 out_unlock:
697         spin_unlock_bh(&fnhe_lock);
698 }
699
700 static void __ip_do_redirect(struct rtable *rt, struct sk_buff *skb, struct flowi4 *fl4,
701                              bool kill_route)
702 {
703         __be32 new_gw = icmp_hdr(skb)->un.gateway;
704         __be32 old_gw = ip_hdr(skb)->saddr;
705         struct net_device *dev = skb->dev;
706         struct in_device *in_dev;
707         struct fib_result res;
708         struct neighbour *n;
```

```
709         struct net *net;
710
711         switch (icmp_hdr(skb)->code & 7) {
712         case ICMP_REDIR_NET:
713         case ICMP_REDIR_NETTOS:
714         case ICMP_REDIR_HOST:
715         case ICMP_REDIR_HOSTTOS:
716                 break;
717
718         default:
719                 return;
720         }
721
722         if (rt->rt_gateway != old_gw)
723                 return;
724
725         in_dev = __in_dev_get_rcu(dev);
726         if (!in_dev)
727                 return;
728
729         net = dev_net(dev);
730         if (new_gw == old_gw || !IN_DEV_RX_REDIRECTS(in_dev) ||
731             ipv4_is_multicast(new_gw) || ipv4_is_lbcast(new_gw) ||
732             ipv4_is_zeronet(new_gw))
733                 goto reject_redirect;
734
735         if (!IN_DEV_SHARED_MEDIA(in_dev)) {
736                 if (!inet_addr_onlink(in_dev, new_gw, old_gw))
737                         goto reject_redirect;
738                 if (IN_DEV_SEC_REDIRECTS(in_dev) && ip_fib_check_default(new_gw, dev))
739                         goto reject_redirect;
740         } else {
741                 if (inet_addr_type(net, new_gw) != RTN_UNICAST)
742                         goto reject_redirect;
743         }
744
745         n = ipv4_neigh_lookup(&rt->dst, NULL, &new_gw);
746         if (!IS_ERR(n)) {
747                 if (!(n->nud_state & NUD_VALID)) {
748                         neigh_event_send(n, NULL);
749                 } else {
750                         if (fib_lookup(net, fl4, &res, 0) == 0) {
751                                 struct fib_nh *nh = &FIB_RES_NH(res);
752
753                                 update_or_create_fnhe(nh, fl4->daddr, new_gw,
754                                                       0, 0);
755                         }
756                         if (kill_route)
757                                 rt->dst.obsolete = DST_OBSOLETE_KILL;
758                         call_netevent_notifiers(NETEVENT_NEIGH_UPDATE, n);
759                 }
760                 neigh_release(n);
761         }
762         return;
763
764 reject_redirect:
765 #ifdef CONFIG_IP_ROUTE_VERBOSE
766         if (IN_DEV_LOG_MARTIANS(in_dev)) {
767                 const struct iphdr *iph = (const struct iphdr *) skb->data;
768                 __be32 daddr = iph->daddr;
769                 __be32 saddr = iph->saddr;
770
771                 net_info_ratelimited("Redirect from %pI4 on %s about %pI4 ignored\n"
772                                      "  Advised path = %pI4 -> %pI4\n",
773                                      &old_gw, dev->name, &new_gw,
774                                      &saddr, &daddr);
775         }
776 #endif
777         ;
778 }
779
780 static void ip_do_redirect(struct dst_entry *dst, struct sock *sk, struct sk_buff *skb)
781 {
782         struct rtable *rt;
783         struct flowi4 fl4;
784         const struct iphdr *iph = (const struct iphdr *) skb->data;
785         int oif = skb->dev->ifindex;
786         u8 tos = RT_TOS(iph->tos);
787         u8 prot = iph->protocol;
788         u32 mark = skb->mark;
789
790         rt = (struct rtable *) dst;
791
792         __build_flow_key(&fl4, sk, iph, oif, tos, prot, mark, 0);
793         __ip_do_redirect(rt, skb, &fl4, true);
794 }
795
796 static struct dst_entry *ipv4_negative_advice(struct dst_entry *dst)
797 {
798         struct rtable *rt = (struct rtable *)dst;
799         struct dst_entry *ret = dst;
800
801         if (rt) {
802                 if (dst->obsolete > 0) {
803                         ip_rt_put(rt);
804                         ret = NULL;
805                 } else if ((rt->rt_flags & RTCF_REDIRECTED) ||
806                            rt->dst.expires) {
807                         ip_rt_put(rt);
808                         ret = NULL;
809                 }
810         }
811         return ret;
812 }
```

```
813
814 /*
815  * Algorithm:
816  *      1. The first ip_rt_redirect_number redirects are sent
817  *         with exponential backoff, then we stop sending them at all,
818  *         assuming that the host ignores our redirects.
819  *      2. If we did not see packets requiring redirects
820  *         during ip_rt_redirect_silence, we assume that the host
821  *         forgot redirected route and start to send redirects again.
822  *
823  * This algorithm is much cheaper and more intelligent than dumb load limiting
824  * in icmp.c.
825  *
826  * NOTE. Do not forget to inhibit load limiting for redirects (redundant)
827  * and "frag. need" (breaks PMTU discovery) in icmp.c.
828  */
829
830 void ip_rt_send_redirect(struct sk_buff *skb)
831 {
832         struct rtable *rt = skb_rtable(skb);
833         struct in_device *in_dev;
834         struct inet_peer *peer;
835         struct net *net;
836         int log_martians;
837
838         rcu_read_lock();
839         in_dev = __in_dev_get_rcu(rt->dst.dev);
840         if (!in_dev || !IN_DEV_TX_REDIRECTS(in_dev)) {
841                 rcu_read_unlock();
842                 return;
843         }
844         log_martians = IN_DEV_LOG_MARTIANS(in_dev);
845         rcu_read_unlock();
846
847         net = dev_net(rt->dst.dev);
848         peer = inet_getpeer_v4(net->ipv4.peers, ip_hdr(skb)->saddr, 1);
849         if (!peer) {
850                 icmp_send(skb, ICMP_REDIRECT, ICMP_REDIR_HOST,
851                           rt_nexthop(rt, ip_hdr(skb)->daddr));
852                 return;
853         }
854
855         /* No redirected packets during ip_rt_redirect_silence;
856          * reset the algorithm.
857          */
858         if (time_after(jiffies, peer->rate_last + ip_rt_redirect_silence))
859                 peer->rate_tokens = 0;
860
861         /* Too many ignored redirects; do not send anything
862          * set dst.rate_last to the last seen redirected packet.
863          */
864         if (peer->rate_tokens >= ip_rt_redirect_number) {
865                 peer->rate_last = jiffies;
866                 goto out_put_peer;
867         }
868
869         /* Check for load limit; set rate_last to the latest sent
870          * redirect.
871          */
872         if (peer->rate_tokens == 0 ||
873             time_after(jiffies,
874                        (peer->rate_last +
875                         (ip_rt_redirect_load << peer->rate_tokens)))) {
876                 __be32 gw = rt_nexthop(rt, ip_hdr(skb)->daddr);
877
878                 icmp_send(skb, ICMP_REDIRECT, ICMP_REDIR_HOST, gw);
879                 peer->rate_last = jiffies;
880                 ++peer->rate_tokens;
881 #ifdef CONFIG_IP_ROUTE_VERBOSE
882                 if (log_martians &&
883                     peer->rate_tokens == ip_rt_redirect_number)
884                         net_warn_ratelimited("host %pI4/if%d ignores redirects for %pI4 to %pI4\n",
885                                              &ip_hdr(skb)->saddr, inet_iif(skb),
886                                              &ip_hdr(skb)->daddr, &gw);
887 #endif
888         }
889 out_put_peer:
890         inet_putpeer(peer);
891 }
892
893 static int ip_error(struct sk_buff *skb)
894 {
895         struct in_device *in_dev = __in_dev_get_rcu(skb->dev);
896         struct rtable *rt = skb_rtable(skb);
897         struct inet_peer *peer;
898         unsigned long now;
899         struct net *net;
900         bool send;
901         int code;
902
903         /* IP on this device is disabled. */
904         if (!in_dev)
905                 goto out;
906
907         net = dev_net(rt->dst.dev);
908         if (!IN_DEV_FORWARD(in_dev)) {
909                 switch (rt->dst.error) {
910                 case EHOSTUNREACH:
911                         IP_INC_STATS_BH(net, IPSTATS_MIB_INADDRERRORS);
912                         break;
913
914                 case ENETUNREACH:
915                         IP_INC_STATS_BH(net, IPSTATS_MIB_INNOROUTES);
916                         break;
```

```
917                 }
918                 goto out;
919         }
920
921         switch (rt->dst.error) {
922         case EINVAL:
923         default:
924                 goto out;
925         case EHOSTUNREACH:
926                 code = ICMP_HOST_UNREACH;
927                 break;
928         case ENETUNREACH:
929                 code = ICMP_NET_UNREACH;
930                 IP_INC_STATS_BH(net, IPSTATS_MIB_INNOROUTES);
931                 break;
932         case EACCES:
933                 code = ICMP_PKT_FILTERED;
934                 break;
935         }
936
937         peer = inet_getpeer_v4(net->ipv4.peers, ip_hdr(skb)->saddr, 1);
938
939         send = true;
940         if (peer) {
941                 now = jiffies;
942                 peer->rate_tokens += now - peer->rate_last;
943                 if (peer->rate_tokens > ip_rt_error_burst)
944                         peer->rate_tokens = ip_rt_error_burst;
945                 peer->rate_last = now;
946                 if (peer->rate_tokens >= ip_rt_error_cost)
947                         peer->rate_tokens -= ip_rt_error_cost;
948                 else
949                         send = false;
950                 inet_putpeer(peer);
951         }
952         if (send)
953                 icmp_send(skb, ICMP_DEST_UNREACH, code, 0);
954
955 out:    kfree_skb(skb);
956         return 0;
957 }
958
959 static void __ip_rt_update_pmtu(struct rtable *rt, struct flowi4 *fl4, u32 mtu)
960 {
961         struct dst_entry *dst = &rt->dst;
962         struct fib_result res;
963
964         if (dst_metric_locked(dst, RTAX_MTU))
965                 return;
966
967         if (ipv4_mtu(dst) < mtu)
968                 return;
969
970         if (mtu < ip_rt_min_pmtu)
971                 mtu = ip_rt_min_pmtu;
972
973         if (rt->rt_pmtu == mtu &&
974             time_before(jiffies, dst->expires - ip_rt_mtu_expires / 2))
975                 return;
976
977         rcu_read_lock();
978         if (fib_lookup(dev_net(dst->dev), fl4, &res, 0) == 0) {
979                 struct fib_nh *nh = &FIB_RES_NH(res);
980
981                 update_or_create_fnhe(nh, fl4->daddr, 0, mtu,
982                                       jiffies + ip_rt_mtu_expires);
983         }
984         rcu_read_unlock();
985 }
986
987 static void ip_rt_update_pmtu(struct dst_entry *dst, struct sock *sk,
988                               struct sk_buff *skb, u32 mtu)
989 {
990         struct rtable *rt = (struct rtable *) dst;
991         struct flowi4 fl4;
992
993         ip_rt_build_flow_key(&fl4, sk, skb);
994         __ip_rt_update_pmtu(rt, &fl4, mtu);
995 }
996
997 void ipv4_update_pmtu(struct sk_buff *skb, struct net *net, u32 mtu,
998                       int oif, u32 mark, u8 protocol, int flow_flags)
999 {
1000         const struct iphdr *iph = (const struct iphdr *) skb->data;
1001         struct flowi4 fl4;
1002         struct rtable *rt;
1003
1004         if (!mark)
1005                 mark = IP4_REPLY_MARK(net, skb->mark);
1006
1007         __build_flow_key(&fl4, NULL, iph, oif,
1008                          RT_TOS(iph->tos), protocol, mark, flow_flags);
1009         rt = __ip_route_output_key(net, &fl4);
1010         if (!IS_ERR(rt)) {
1011                 __ip_rt_update_pmtu(rt, &fl4, mtu);
1012                 ip_rt_put(rt);
1013         }
1014 }
1015 EXPORT_SYMBOL_GPL(ipv4_update_pmtu);
1016
1017 static void __ipv4_sk_update_pmtu(struct sk_buff *skb, struct sock *sk, u32 mtu)
1018 {
1019         const struct iphdr *iph = (const struct iphdr *) skb->data;
1020         struct flowi4 fl4;
```

```
1021         struct rtable *rt;
1022
1023         __build_flow_key(&fl4, sk, iph, 0, 0, 0, 0, 0);
1024
1025         if (!fl4.flowi4_mark)
1026                 fl4.flowi4_mark = IP4_REPLY_MARK(sock_net(sk), skb->mark);
1027
1028         rt = __ip_route_output_key(sock_net(sk), &fl4);
1029         if (!IS_ERR(rt)) {
1030                 __ip_rt_update_pmtu(rt, &fl4, mtu);
1031                 ip_rt_put(rt);
1032         }
1033 }
1034
1035 void ipv4_sk_update_pmtu(struct sk_buff *skb, struct sock *sk, u32 mtu)
1036 {
1037         const struct iphdr *iph = (const struct iphdr *) skb->data;
1038         struct flowi4 fl4;
1039         struct rtable *rt;
1040         struct dst_entry *odst = NULL;
1041         bool new = false;
1042
1043         bh_lock_sock(sk);
1044
1045         if (!ip_sk_accept_pmtu(sk))
1046                 goto out;
1047
1048         odst = sk_dst_get(sk);
1049
1050         if (sock_owned_by_user(sk) || !odst) {
1051                 __ipv4_sk_update_pmtu(skb, sk, mtu);
1052                 goto out;
1053         }
1054
1055         __build_flow_key(&fl4, sk, iph, 0, 0, 0, 0, 0);
1056
1057         rt = (struct rtable *)odst;
1058         if (odst->obsolete && !odst->ops->check(odst, 0)) {
1059                 rt = ip_route_output_flow(sock_net(sk), &fl4, sk);
1060                 if (IS_ERR(rt))
1061                         goto out;
1062
1063                 new = true;
1064         }
1065
1066         __ip_rt_update_pmtu((struct rtable *) rt->dst.path, &fl4, mtu);
1067
1068         if (!dst_check(&rt->dst, 0)) {
1069                 if (new)
1070                         dst_release(&rt->dst);
1071
1072                 rt = ip_route_output_flow(sock_net(sk), &fl4, sk);
1073                 if (IS_ERR(rt))
1074                         goto out;
1075
1076                 new = true;
1077         }
1078
1079         if (new)
1080                 sk_dst_set(sk, &rt->dst);
1081
1082 out:
1083         bh_unlock_sock(sk);
1084         dst_release(odst);
1085 }
1086 EXPORT_SYMBOL_GPL(ipv4_sk_update_pmtu);
1087
1088 void ipv4_redirect(struct sk_buff *skb, struct net *net,
1089                    int oif, u32 mark, u8 protocol, int flow_flags)
1090 {
1091         const struct iphdr *iph = (const struct iphdr *) skb->data;
1092         struct flowi4 fl4;
1093         struct rtable *rt;
1094
1095         __build_flow_key(&fl4, NULL, iph, oif,
1096                          RT_TOS(iph->tos), protocol, mark, flow_flags);
1097         rt = __ip_route_output_key(net, &fl4);
1098         if (!IS_ERR(rt)) {
1099                 __ip_do_redirect(rt, skb, &fl4, false);
1100                 ip_rt_put(rt);
1101         }
1102 }
1103 EXPORT_SYMBOL_GPL(ipv4_redirect);
1104
1105 void ipv4_sk_redirect(struct sk_buff *skb, struct sock *sk)
1106 {
1107         const struct iphdr *iph = (const struct iphdr *) skb->data;
1108         struct flowi4 fl4;
1109         struct rtable *rt;
1110
1111         __build_flow_key(&fl4, sk, iph, 0, 0, 0, 0, 0);
1112         rt = __ip_route_output_key(sock_net(sk), &fl4);
1113         if (!IS_ERR(rt)) {
1114                 __ip_do_redirect(rt, skb, &fl4, false);
1115                 ip_rt_put(rt);
1116         }
1117 }
1118 EXPORT_SYMBOL_GPL(ipv4_sk_redirect);
1119
1120 static struct dst_entry *ipv4_dst_check(struct dst_entry *dst, u32 cookie)
1121 {
1122         struct rtable *rt = (struct rtable *) dst;
1123
1124         /* All IPV4 dsts are created with ->obsolete set to the value
```

```
1125          * DST_OBSOLETE_FORCE_CHK which forces validation calls down
1126          * into this function always.
1127          *
1128          * When a PMTU/redirect information update invalidates a route,
1129          * this is indicated by setting obsolete to DST_OBSOLETE_KILL or
1130          * DST_OBSOLETE_DEAD by dst_free().
1131          */
1132         if (dst->obsolete != DST_OBSOLETE_FORCE_CHK || rt_is_expired(rt))
1133                 return NULL;
1134         return dst;
1135 }
1136
1137 static void ipv4_link_failure(struct sk_buff *skb)
1138 {
1139         struct rtable *rt;
1140
1141         icmp_send(skb, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH, 0);
1142
1143         rt = skb_rtable(skb);
1144         if (rt)
1145                 dst_set_expires(&rt->dst, 0);
1146 }
1147
1148 static int ip_rt_bug(struct sock *sk, struct sk_buff *skb)
1149 {
1150         pr_debug("%s: %pI4 -> %pI4, %s\n",
1151                  __func__, &ip_hdr(skb)->saddr, &ip_hdr(skb)->daddr,
1152                  skb->dev ? skb->dev->name : "?");
1153         kfree_skb(skb);
1154         WARN_ON(1);
1155         return 0;
1156 }
1157
1158 /*
1159    We do not cache source address of outgoing interface,
1160    because it is used only by IP RR, TS and SRR options,
1161    so that it out of fast path.
1162
1163    BTW remember: "addr" is allowed to be not aligned
1164    in IP options!
1165  */
1166
1167 void ip_rt_get_source(u8 *addr, struct sk_buff *skb, struct rtable *rt)
1168 {
1169         __be32 src;
1170
1171         if (rt_is_output_route(rt))
1172                 src = ip_hdr(skb)->saddr;
1173         else {
1174                 struct fib_result res;
1175                 struct flowi4 fl4;
1176                 struct iphdr *iph;
1177
1178                 iph = ip_hdr(skb);
1179
1180                 memset(&fl4, 0, sizeof(fl4));
1181                 fl4.daddr = iph->daddr;
1182                 fl4.saddr = iph->saddr;
1183                 fl4.flowi4_tos = RT_TOS(iph->tos);
1184                 fl4.flowi4_oif = rt->dst.dev->ifindex;
1185                 fl4.flowi4_iif = skb->dev->ifindex;
1186                 fl4.flowi4_mark = skb->mark;
1187
1188                 rcu_read_lock();
1189                 if (fib_lookup(dev_net(rt->dst.dev), &fl4, &res, 0) == 0)
1190                         src = FIB_RES_PREFSRC(dev_net(rt->dst.dev), res);
1191                 else
1192                         src = inet_select_addr(rt->dst.dev,
1193                                                rt_nexthop(rt, iph->daddr),
1194                                                RT_SCOPE_UNIVERSE);
1195                 rcu_read_unlock();
1196         }
1197         memcpy(addr, &src, 4);
1198 }
1199
1200 #ifdef CONFIG_IP_ROUTE_CLASSID
1201 static void set_class_tag(struct rtable *rt, u32 tag)
1202 {
1203         if (!(rt->dst.tclassid & 0xFFFF))
1204                 rt->dst.tclassid |= tag & 0xFFFF;
1205         if (!(rt->dst.tclassid & 0xFFFF0000))
1206                 rt->dst.tclassid |= tag & 0xFFFF0000;
1207 }
1208 #endif
1209
1210 static unsigned int ipv4_default_advmss(const struct dst_entry *dst)
1211 {
1212         unsigned int advmss = dst_metric_raw(dst, RTAX_ADVMSS);
1213
1214         if (advmss == 0) {
1215                 advmss = max_t(unsigned int, dst->dev->mtu - 40,
1216                                ip_rt_min_advmss);
1217                 if (advmss > 65535 - 40)
1218                         advmss = 65535 - 40;
1219         }
1220         return advmss;
1221 }
1222
1223 static unsigned int ipv4_mtu(const struct dst_entry *dst)
1224 {
1225         const struct rtable *rt = (const struct rtable *) dst;
1226         unsigned int mtu = rt->rt_pmtu;
1227
1228         if (!mtu || time_after_eq(jiffies, rt->dst.expires))
```

```
1229                 mtu = dst_metric_raw(dst, RTAX_MTU);
1230
1231         if (mtu)
1232                 return mtu;
1233
1234         mtu = dst->dev->mtu;
1235
1236         if (unlikely(dst_metric_locked(dst, RTAX_MTU))) {
1237                 if (rt->rt_uses_gateway && mtu > 576)
1238                         mtu = 576;
1239         }
1240
1241         return min_t(unsigned int, mtu, IP_MAX_MTU);
1242 }
1243
1244 static struct fib_nh_exception *find_exception(struct fib_nh *nh, __be32 daddr)
1245 {
1246         struct fnhe_hash_bucket *hash = rcu_dereference(nh->nh_exceptions);
1247         struct fib_nh_exception *fnhe;
1248         u32 hval;
1249
1250         if (!hash)
1251                 return NULL;
1252
1253         hval = fnhe_hashfun(daddr);
1254
1255         for (fnhe = rcu_dereference(hash[hval].chain); fnhe;
1256              fnhe = rcu_dereference(fnhe->fnhe_next)) {
1257                 if (fnhe->fnhe_daddr == daddr)
1258                         return fnhe;
1259         }
1260         return NULL;
1261 }
1262
1263 static bool rt_bind_exception(struct rtable *rt, struct fib_nh_exception *fnhe,
1264                               __be32 daddr)
1265 {
1266         bool ret = false;
1267
1268         spin_lock_bh(&fnhe_lock);
1269
1270         if (daddr == fnhe->fnhe_daddr) {
1271                 struct rtable __rcu **porig;
1272                 struct rtable *orig;
1273                 int genid = fnhe_genid(dev_net(rt->dst.dev));
1274
1275                 if (rt_is_input_route(rt))
1276                         porig = &fnhe->fnhe_rth_input;
1277                 else
1278                         porig = &fnhe->fnhe_rth_output;
1279                 orig = rcu_dereference(*porig);
1280
1281                 if (fnhe->fnhe_genid != genid) {
1282                         fnhe->fnhe_genid = genid;
1283                         fnhe->fnhe_gw = 0;
1284                         fnhe->fnhe_pmtu = 0;
1285                         fnhe->fnhe_expires = 0;
1286                         fnhe_flush_routes(fnhe);
1287                         orig = NULL;
1288                 }
1289                 fill_route_from_fnhe(rt, fnhe);
1290                 if (!rt->rt_gateway)
1291                         rt->rt_gateway = daddr;
1292
1293                 if (!(rt->dst.flags & DST_NOCACHE)) {
1294                         rcu_assign_pointer(*porig, rt);
1295                         if (orig)
1296                                 rt_free(orig);
1297                         ret = true;
1298                 }
1299
1300                 fnhe->fnhe_stamp = jiffies;
1301         }
1302         spin_unlock_bh(&fnhe_lock);
1303
1304         return ret;
1305 }
1306
1307 static bool rt_cache_route(struct fib_nh *nh, struct rtable *rt)
1308 {
1309         struct rtable *orig, *prev, **p;
1310         bool ret = true;
1311
1312         if (rt_is_input_route(rt)) {
1313                 p = (struct rtable **)&nh->nh_rth_input;
1314         } else {
1315                 p = (struct rtable **)raw_cpu_ptr(nh->nh_pcpu_rth_output);
1316         }
1317         orig = *p;
1318
1319         prev = cmpxchg(p, orig, rt);
1320         if (prev == orig) {
1321                 if (orig)
1322                         rt_free(orig);
1323         } else
1324                 ret = false;
1325
1326         return ret;
1327 }
1328
1329 struct uncached_list {
1330         spinlock_t              lock;
1331         struct list_head        head;
1332 };
```

```
1333
1334 static DEFINE_PER_CPU_ALIGNED(struct uncached_list, rt_uncached_list);
1335
1336 static void rt_add_uncached_list(struct rtable *rt)
1337 {
1338         struct uncached_list *ul = raw_cpu_ptr(&rt_uncached_list);
1339
1340         rt->rt_uncached_list = ul;
1341
1342         spin_lock_bh(&ul->lock);
1343         list_add_tail(&rt->rt_uncached, &ul->head);
1344         spin_unlock_bh(&ul->lock);
1345 }
1346
1347 static void ipv4_dst_destroy(struct dst_entry *dst)
1348 {
1349         struct rtable *rt = (struct rtable *) dst;
1350
1351         if (!list_empty(&rt->rt_uncached)) {
1352                 struct uncached_list *ul = rt->rt_uncached_list;
1353
1354                 spin_lock_bh(&ul->lock);
1355                 list_del(&rt->rt_uncached);
1356                 spin_unlock_bh(&ul->lock);
1357         }
1358 }
1359
1360 void rt_flush_dev(struct net_device *dev)
1361 {
1362         struct net *net = dev_net(dev);
1363         struct rtable *rt;
1364         int cpu;
1365
1366         for_each_possible_cpu(cpu) {
1367                 struct uncached_list *ul = &per_cpu(rt_uncached_list, cpu);
1368
1369                 spin_lock_bh(&ul->lock);
1370                 list_for_each_entry(rt, &ul->head, rt_uncached) {
1371                         if (rt->dst.dev != dev)
1372                                 continue;
1373                         rt->dst.dev = net->loopback_dev;
1374                         dev_hold(rt->dst.dev);
1375                         dev_put(dev);
1376                 }
1377                 spin_unlock_bh(&ul->lock);
1378         }
1379 }
1380
1381 static bool rt_cache_valid(const struct rtable *rt)
1382 {
1383         return  rt &&
1384                 rt->dst.obsolete == DST_OBSOLETE_FORCE_CHK &&
1385                 !rt_is_expired(rt);
1386 }
1387
1388 static void rt_set_nexthop(struct rtable *rt, __be32 daddr,
1389                            const struct fib_result *res,
1390                            struct fib_nh_exception *fnhe,
1391                            struct fib_info *fi, u16 type, u32 itag)
1392 {
1393         bool cached = false;
1394
1395         if (fi) {
1396                 struct fib_nh *nh = &FIB_RES_NH(*res);
1397
1398                 if (nh->nh_gw && nh->nh_scope == RT_SCOPE_LINK) {
1399                         rt->rt_gateway = nh->nh_gw;
1400                         rt->rt_uses_gateway = 1;
1401                 }
1402                 dst_init_metrics(&rt->dst, fi->fib_metrics, true);
1403 #ifdef CONFIG_IP_ROUTE_CLASSID
1404                 rt->dst.tclassid = nh->nh_tclassid;
1405 #endif
1406                 if (unlikely(fnhe))
1407                         cached = rt_bind_exception(rt, fnhe, daddr);
1408                 else if (!(rt->dst.flags & DST_NOCACHE))
1409                         cached = rt_cache_route(nh, rt);
1410                 if (unlikely(!cached)) {
1411                         /* Routes we intend to cache in nexthop exception or
1412                          * FIB nexthop have the DST_NOCACHE bit clear.
1413                          * However, if we are unsuccessful at storing this
1414                          * route into the cache we really need to set it.
1415                          */
1416                         rt->dst.flags |= DST_NOCACHE;
1417                         if (!rt->rt_gateway)
1418                                 rt->rt_gateway = daddr;
1419                         rt_add_uncached_list(rt);
1420                 }
1421         } else
1422                 rt_add_uncached_list(rt);
1423
1424 #ifdef CONFIG_IP_ROUTE_CLASSID
1425 #ifdef CONFIG_IP_MULTIPLE_TABLES
1426         set_class_tag(rt, res->tclassid);
1427 #endif
1428         set_class_tag(rt, itag);
1429 #endif
1430 }
1431
1432 static struct rtable *rt_dst_alloc(struct net_device *dev,
1433                                    bool nopolicy, bool noxfrm, bool will_cache)
1434 {
1435         return dst_alloc(&ipv4_dst_ops, dev, 1, DST_OBSOLETE_FORCE_CHK,
1436                          (will_cache ? 0 : (DST_HOST | DST_NOCACHE)) |
```

```
1437                        (nopolicy ? DST_NOPOLICY : 0) |
1438                        (noxfrm ? DST_NOXFRM : 0));
1439 }
1440
1441 /* called in rcu_read_lock() section */
1442 static int ip_route_input_mc(struct sk_buff *skb, __be32 daddr, __be32 saddr,
1443                              u8 tos, struct net_device *dev, int our)
1444 {
1445         struct rtable *rth;
1446         struct in_device *in_dev = __in_dev_get_rcu(dev);
1447         u32 itag = 0;
1448         int err;
1449
1450         /* Primary sanity checks. */
1451
1452         if (!in_dev)
1453                 return -EINVAL;
1454
1455         if (ipv4_is_multicast(saddr) || ipv4_is_lbcast(saddr) ||
1456             skb->protocol != htons(ETH_P_IP))
1457                 goto e_inval;
1458
1459         if (likely(!IN_DEV_ROUTE_LOCALNET(in_dev)))
1460                 if (ipv4_is_loopback(saddr))
1461                         goto e_inval;
1462
1463         if (ipv4_is_zeronet(saddr)) {
1464                 if (!ipv4_is_local_multicast(daddr))
1465                         goto e_inval;
1466         } else {
1467                 err = fib_validate_source(skb, saddr, 0, tos, 0, dev,
1468                                           in_dev, &itag);
1469                 if (err < 0)
1470                         goto e_err;
1471         }
1472         rth = rt_dst_alloc(dev_net(dev)->loopback_dev,
1473                            IN_DEV_CONF_GET(in_dev, NOPOLICY), false, false);
1474         if (!rth)
1475                 goto e_nobufs;
1476
1477 #ifdef CONFIG_IP_ROUTE_CLASSID
1478         rth->dst.tclassid = itag;
1479 #endif
1480         rth->dst.output = ip_rt_bug;
1481
1482         rth->rt_genid   = rt_genid_ipv4(dev_net(dev));
1483         rth->rt_flags   = RTCF_MULTICAST;
1484         rth->rt_type    = RTN_MULTICAST;
1485         rth->rt_is_input= 1;
1486         rth->rt_iif     = 0;
1487         rth->rt_pmtu    = 0;
1488         rth->rt_gateway = 0;
1489         rth->rt_uses_gateway = 0;
1490         INIT_LIST_HEAD(&rth->rt_uncached);
1491         if (our) {
1492                 rth->dst.input= ip_local_deliver;
1493                 rth->rt_flags |= RTCF_LOCAL;
1494         }
1495
1496 #ifdef CONFIG_IP_MROUTE
1497         if (!ipv4_is_local_multicast(daddr) && IN_DEV_MFORWARD(in_dev))
1498                 rth->dst.input = ip_mr_input;
1499 #endif
1500         RT_CACHE_STAT_INC(in_slow_mc);
1501
1502         skb_dst_set(skb, &rth->dst);
1503         return 0;
1504
1505 e_nobufs:
1506         return -ENOBUFS;
1507 e_inval:
1508         return -EINVAL;
1509 e_err:
1510         return err;
1511 }
1512
1513
1514 static void ip_handle_martian_source(struct net_device *dev,
1515                                      struct in_device *in_dev,
1516                                      struct sk_buff *skb,
1517                                      __be32 daddr,
1518                                      __be32 saddr)
1519 {
1520         RT_CACHE_STAT_INC(in_martian_src);
1521 #ifdef CONFIG_IP_ROUTE_VERBOSE
1522         if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit()) {
1523                 /*
1524                  *      RFC1812 recommendation, if source is martian,
1525                  *      the only hint is MAC header.
1526                  */
1527                 pr_warn("martian source %pI4 from %pI4, on dev %s\n",
1528                         &daddr, &saddr, dev->name);
1529                 if (dev->hard_header_len && skb_mac_header_was_set(skb)) {
1530                         print_hex_dump(KERN_WARNING, "ll header: ",
1531                                        DUMP_PREFIX_OFFSET, 16, 1,
1532                                        skb_mac_header(skb),
1533                                        dev->hard_header_len, true);
1534                 }
1535         }
1536 #endif
1537 }
1538
1539 /* called in rcu_read_lock() section */
1540 static int __mkroute_input(struct sk_buff *skb,
```

```
1541                            const struct fib_result *res,
1542                            struct in_device *in_dev,
1543                            __be32 daddr, __be32 saddr, u32 tos)
1544 {
1545         struct fib_nh_exception *fnhe;
1546         struct rtable *rth;
1547         int err;
1548         struct in_device *out_dev;
1549         unsigned int flags = 0;
1550         bool do_cache;
1551         u32 itag = 0;
1552
1553         /* get a working reference to the output device */
1554         out_dev = __in_dev_get_rcu(FIB_RES_DEV(*res));
1555         if (!out_dev) {
1556                 net_crit_ratelimited("Bug in ip_route_input_slow(). Please report.\n");
1557                 return -EINVAL;
1558         }
1559
1560         err = fib_validate_source(skb, saddr, daddr, tos, FIB_RES_OIF(*res),
1561                                   in_dev->dev, in_dev, &itag);
1562         if (err < 0) {
1563                 ip_handle_martian_source(in_dev->dev, in_dev, skb, daddr,
1564                                          saddr);
1565
1566                 goto cleanup;
1567         }
1568
1569         do_cache = res->fi && !itag;
1570         if (out_dev == in_dev && err && IN_DEV_TX_REDIRECTS(out_dev) &&
1571             skb->protocol == htons(ETH_P_IP) &&
1572             (IN_DEV_SHARED_MEDIA(out_dev) ||
1573              inet_addr_onlink(out_dev, saddr, FIB_RES_GW(*res))))
1574                 IPCB(skb)->flags |= IPSKB_DOREDIRECT;
1575
1576         if (skb->protocol != htons(ETH_P_IP)) {
1577                 /* Not IP (i.e. ARP). Do not create route, if it is
1578                  * invalid for proxy arp. DNAT routes are always valid.
1579                  *
1580                  * Proxy arp feature have been extended to allow, ARP
1581                  * replies back to the same interface, to support
1582                  * Private VLAN switch technologies. See arp.c.
1583                  */
1584                 if (out_dev == in_dev &&
1585                     IN_DEV_PROXY_ARP_PVLAN(in_dev) == 0) {
1586                         err = -EINVAL;
1587                         goto cleanup;
1588                 }
1589         }
1590
1591         fnhe = find_exception(&FIB_RES_NH(*res), daddr);
1592         if (do_cache) {
1593                 if (fnhe)
1594                         rth = rcu_dereference(fnhe->fnhe_rth_input);
1595                 else
1596                         rth = rcu_dereference(FIB_RES_NH(*res).nh_rth_input);
1597
1598                 if (rt_cache_valid(rth)) {
1599                         skb_dst_set_noref(skb, &rth->dst);
1600                         goto out;
1601                 }
1602         }
1603
1604         rth = rt_dst_alloc(out_dev->dev,
1605                            IN_DEV_CONF_GET(in_dev, NOPOLICY),
1606                            IN_DEV_CONF_GET(out_dev, NOXFRM), do_cache);
1607         if (!rth) {
1608                 err = -ENOBUFS;
1609                 goto cleanup;
1610         }
1611
1612         rth->rt_genid = rt_genid_ipv4(dev_net(rth->dst.dev));
1613         rth->rt_flags = flags;
1614         rth->rt_type = res->type;
1615         rth->rt_is_input = 1;
1616         rth->rt_iif     = 0;
1617         rth->rt_pmtu    = 0;
1618         rth->rt_gateway = 0;
1619         rth->rt_uses_gateway = 0;
1620         INIT_LIST_HEAD(&rth->rt_uncached);
1621         RT_CACHE_STAT_INC(in_slow_tot);
1622
1623         rth->dst.input = ip_forward;
1624         rth->dst.output = ip_output;
1625
1626         rt_set_nexthop(rth, daddr, res, fnhe, res->fi, res->type, itag);
1627         skb_dst_set(skb, &rth->dst);
1628 out:
1629         err = 0;
1630  cleanup:
1631         return err;
1632 }
1633
1634 static int ip_mkroute_input(struct sk_buff *skb,
1635                             struct fib_result *res,
1636                             const struct flowi4 *fl4,
1637                             struct in_device *in_dev,
1638                             __be32 daddr, __be32 saddr, u32 tos)
1639 {
1640 #ifdef CONFIG_IP_ROUTE_MULTIPATH
1641         if (res->fi && res->fi->fib_nhs > 1)
1642                 fib_select_multipath(res);
1643 #endif
1644
```

```
1645         /* create a routing cache entry */
1646         return __mkroute_input(skb, res, in_dev, daddr, saddr, tos);
1647 }
1648
1649 /*
1650  *      NOTE. We drop all the packets that has local source
1651  *      addresses, because every properly looped back packet
1652  *      must have correct destination already attached by output routine.
1653  *
1654  *      Such approach solves two big problems:
1655  *      1. Not simplex devices are handled properly.
1656  *      2. IP spoofing attempts are filtered with 100% of guarantee.
1657  *      called with rcu_read_lock()
1658  */
1659
1660 static int ip_route_input_slow(struct sk_buff *skb, __be32 daddr, __be32 saddr,
1661                                u8 tos, struct net_device *dev)
1662 {
1663         struct fib_result res;
1664         struct in_device *in_dev = __in_dev_get_rcu(dev);
1665         struct flowi4   f14;
1666         unsigned int    flags = 0;
1667         u32             itag = 0;
1668         struct rtable   *rth;
1669         int             err = -EINVAL;
1670         struct net    *net = dev_net(dev);
1671         bool do_cache;
1672
1673         /* IP on this device is disabled. */
1674
1675         if (!in_dev)
1676                 goto out;
1677
1678         /* Check for the most weird martians, which can be not detected
1679            by fib_lookup.
1680          */
1681
1682         if (ipv4_is_multicast(saddr) || ipv4_is_lbcast(saddr))
1683                 goto martian_source;
1684
1685         res.fi = NULL;
1686         if (ipv4_is_lbcast(daddr) || (saddr == 0 && daddr == 0))
1687                 goto brd_input;
1688
1689         /* Accept zero addresses only to limited broadcast;
1690          * I even do not know to fix it or not. Waiting for complains :-)
1691          */
1692         if (ipv4_is_zeronet(saddr))
1693                 goto martian_source;
1694
1695         if (ipv4_is_zeronet(daddr))
1696                 goto martian_destination;
1697
1698         /* Following code try to avoid calling IN_DEV_NET_ROUTE_LOCALNET(),
1699          * and call it once if daddr or/and saddr are loopback addresses
1700          */
1701         if (ipv4_is_loopback(daddr)) {
1702                 if (!IN_DEV_NET_ROUTE_LOCALNET(in_dev, net))
1703                         goto martian_destination;
1704         } else if (ipv4_is_loopback(saddr)) {
1705                 if (!IN_DEV_NET_ROUTE_LOCALNET(in_dev, net))
1706                         goto martian_source;
1707         }
1708
1709         /*
1710          *      Now we are ready to route packet.
1711          */
1712         f14.flowi4_oif = 0;
1713         f14.flowi4_iif = dev->ifindex;
1714         f14.flowi4_mark = skb->mark;
1715         f14.flowi4_tos = tos;
1716         f14.flowi4_scope = RT_SCOPE_UNIVERSE;
1717         f14.daddr = daddr;
1718         f14.saddr = saddr;
1719         err = fib_lookup(net, &f14, &res, 0);
1720         if (err != 0) {
1721                 if (!IN_DEV_FORWARD(in_dev))
1722                         err = -EHOSTUNREACH;
1723                 goto no_route;
1724         }
1725
1726         if (res.type == RTN_BROADCAST)
1727                 goto brd_input;
1728
1729         if (res.type == RTN_LOCAL) {
1730                 err = fib_validate_source(skb, saddr, daddr, tos,
1731                                           0, dev, in_dev, &itag);
1732                 if (err < 0)
1733                         goto martian_source_keep_err;
1734                 goto local_input;
1735         }
1736
1737         if (!IN_DEV_FORWARD(in_dev)) {
1738                 err = -EHOSTUNREACH;
1739                 goto no_route;
1740         }
1741         if (res.type != RTN_UNICAST)
1742                 goto martian_destination;
1743
1744         err = ip_mkroute_input(skb, &res, &f14, in_dev, daddr, saddr, tos);
1745 out:    return err;
1746
1747 brd_input:
1748         if (skb->protocol != htons(ETH_P_IP))
```

```
1749                    goto e_inval;
1750
1751            if (!ipv4_is_zeronet(saddr)) {
1752                    err = fib_validate_source(skb, saddr, 0, tos, 0, dev,
1753                                              in_dev, &itag);
1754                    if (err < 0)
1755                            goto martian_source_keep_err;
1756            }
1757            flags |= RTCF_BROADCAST;
1758            res.type = RTN_BROADCAST;
1759            RT_CACHE_STAT_INC(in_brd);
1760
1761 local_input:
1762            do_cache = false;
1763            if (res.fi) {
1764                    if (!itag) {
1765                            rth = rcu_dereference(FIB_RES_NH(res).nh_rth_input);
1766                            if (rt_cache_valid(rth)) {
1767                                    skb_dst_set_noref(skb, &rth->dst);
1768                                    err = 0;
1769                                    goto out;
1770                            }
1771                            do_cache = true;
1772                    }
1773            }
1774
1775            rth = rt_dst_alloc(net->loopback_dev,
1776                               IN_DEV_CONF_GET(in_dev, NOPOLICY), false, do_cache);
1777            if (!rth)
1778                    goto e_nobufs;
1779
1780            rth->dst.input= ip_local_deliver;
1781            rth->dst.output= ip_rt_bug;
1782 #ifdef CONFIG_IP_ROUTE_CLASSID
1783            rth->dst.tclassid = itag;
1784 #endif
1785
1786            rth->rt_genid = rt_genid_ipv4(net);
1787            rth->rt_flags   = flags|RTCF_LOCAL;
1788            rth->rt_type    = res.type;
1789            rth->rt_is_input = 1;
1790            rth->rt_iif     = 0;
1791            rth->rt_pmtu    = 0;
1792            rth->rt_gateway = 0;
1793            rth->rt_uses_gateway = 0;
1794            INIT_LIST_HEAD(&rth->rt_uncached);
1795            RT_CACHE_STAT_INC(in_slow_tot);
1796            if (res.type == RTN_UNREACHABLE) {
1797                    rth->dst.input= ip_error;
1798                    rth->dst.error= -err;
1799                    rth->rt_flags   &= ~RTCF_LOCAL;
1800            }
1801            if (do_cache) {
1802                    if (unlikely(!rt_cache_route(&FIB_RES_NH(res), rth))) {
1803                            rth->dst.flags |= DST_NOCACHE;
1804                            rt_add_uncached_list(rth);
1805                    }
1806            }
1807            skb_dst_set(skb, &rth->dst);
1808            err = 0;
1809            goto out;
1810
1811 no_route:
1812            RT_CACHE_STAT_INC(in_no_route);
1813            res.type = RTN_UNREACHABLE;
1814            res.fi = NULL;
1815            goto local_input;
1816
1817            /*
1818             *      Do not cache martian addresses: they should be logged (RFC1812)
1819             */
1820 martian_destination:
1821            RT_CACHE_STAT_INC(in_martian_dst);
1822 #ifdef CONFIG_IP_ROUTE_VERBOSE
1823            if (IN_DEV_LOG_MARTIANS(in_dev))
1824                    net_warn_ratelimited("martian destination %pI4 from %pI4, dev %s\n",
1825                                         &daddr, &saddr, dev->name);
1826 #endif
1827
1828 e_inval:
1829            err = -EINVAL;
1830            goto out;
1831
1832 e_nobufs:
1833            err = -ENOBUFS;
1834            goto out;
1835
1836 martian_source:
1837            err = -EINVAL;
1838 martian_source_keep_err:
1839            ip_handle_martian_source(dev, in_dev, skb, daddr, saddr);
1840            goto out;
1841 }
1842
1843 int ip_route_input_noref(struct sk_buff *skb, __be32 daddr, __be32 saddr,
1844                          u8 tos, struct net_device *dev)
1845 {
1846            int res;
1847
1848            rcu_read_lock();
1849
1850            /* Multicast recognition logic is moved from route cache to here.
1851               The problem was that too many Ethernet cards have broken/missing
1852               hardware multicast filters :-( As result the host on multicasting
```

```
1853            network acquires a lot of useless route cache entries, sort of
1854            SDR messages from all the world. Now we try to get rid of them.
1855            Really, provided software IP multicast filter is organized
1856            reasonably (at least, hashed), it does not result in a slowdown
1857            comparing with route cache reject entries.
1858            Note, that multicast routers are not affected, because
1859            route cache entry is created eventually.
1860          */
1861         if (ipv4_is_multicast(daddr)) {
1862                 struct in_device *in_dev = __in_dev_get_rcu(dev);
1863
1864                 if (in_dev) {
1865                         int our = ip_check_mc_rcu(in_dev, daddr, saddr,
1866                                                   ip_hdr(skb)->protocol);
1867                         if (our
1868 #ifdef CONFIG_IP_MROUTE
1869                                 ||
1870                             (!ipv4_is_local_multicast(daddr) &&
1871                              IN_DEV_MFORWARD(in_dev))
1872 #endif
1873                             ) {
1874                                 int res = ip_route_input_mc(skb, daddr, saddr,
1875                                                             tos, dev, our);
1876                                 rcu_read_unlock();
1877                                 return res;
1878                         }
1879                 }
1880                 rcu_read_unlock();
1881                 return -EINVAL;
1882         }
1883         res = ip_route_input_slow(skb, daddr, saddr, tos, dev);
1884         rcu_read_unlock();
1885         return res;
1886 }
1887 EXPORT_SYMBOL(ip_route_input_noref);
1888
1889 /* called with rcu_read_lock() */
1890 static struct rtable *__mkroute_output(const struct fib_result *res,
1891                                        const struct flowi4 *fl4, int orig_oif,
1892                                        struct net_device *dev_out,
1893                                        unsigned int flags)
1894 {
1895         struct fib_info *fi = res->fi;
1896         struct fib_nh_exception *fnhe;
1897         struct in_device *in_dev;
1898         u16 type = res->type;
1899         struct rtable *rth;
1900         bool do_cache;
1901
1902         in_dev = __in_dev_get_rcu(dev_out);
1903         if (!in_dev)
1904                 return ERR_PTR(-EINVAL);
1905
1906         if (likely(!IN_DEV_ROUTE_LOCALNET(in_dev)))
1907                 if (ipv4_is_loopback(fl4->saddr) && !(dev_out->flags & IFF_LOOPBACK))
1908                         return ERR_PTR(-EINVAL);
1909
1910         if (ipv4_is_lbcast(fl4->daddr))
1911                 type = RTN_BROADCAST;
1912         else if (ipv4_is_multicast(fl4->daddr))
1913                 type = RTN_MULTICAST;
1914         else if (ipv4_is_zeronet(fl4->daddr))
1915                 return ERR_PTR(-EINVAL);
1916
1917         if (dev_out->flags & IFF_LOOPBACK)
1918                 flags |= RTCF_LOCAL;
1919
1920         do_cache = true;
1921         if (type == RTN_BROADCAST) {
1922                 flags |= RTCF_BROADCAST | RTCF_LOCAL;
1923                 fi = NULL;
1924         } else if (type == RTN_MULTICAST) {
1925                 flags |= RTCF_MULTICAST | RTCF_LOCAL;
1926                 if (!ip_check_mc_rcu(in_dev, fl4->daddr, fl4->saddr,
1927                                      fl4->flowi4_proto))
1928                         flags &= ~RTCF_LOCAL;
1929                 else
1930                         do_cache = false;
1931                 /* If multicast route do not exist use
1932                  * default one, but do not gateway in this case.
1933                  * Yes, it is hack.
1934                  */
1935                 if (fi && res->prefixlen < 4)
1936                         fi = NULL;
1937         }
1938
1939         fnhe = NULL;
1940         do_cache &= fi != NULL;
1941         if (do_cache) {
1942                 struct rtable __rcu **prth;
1943                 struct fib_nh *nh = &FIB_RES_NH(*res);
1944
1945                 fnhe = find_exception(nh, fl4->daddr);
1946                 if (fnhe)
1947                         prth = &fnhe->fnhe_rth_output;
1948                 else {
1949                         if (unlikely(fl4->flowi4_flags &
1950                                      FLOWI_FLAG_KNOWN_NH &&
1951                                      !(nh->nh_gw &&
1952                                        nh->nh_scope == RT_SCOPE_LINK))) {
1953                                 do_cache = false;
1954                                 goto add;
1955                         }
1956                         prth = raw_cpu_ptr(nh->nh_pcpu_rth_output);
```

```
1957                 }
1958                 rth = rcu_dereference(*prth);
1959                 if (rt_cache_valid(rth)) {
1960                         dst_hold(&rth->dst);
1961                         return rth;
1962                 }
1963         }
1964
1965 add:
1966         rth = rt_dst_alloc(dev_out,
1967                            IN_DEV_CONF_GET(in_dev, NOPOLICY),
1968                            IN_DEV_CONF_GET(in_dev, NOXFRM),
1969                            do_cache);
1970         if (!rth)
1971                 return ERR_PTR(-ENOBUFS);
1972
1973         rth->dst.output = ip_output;
1974
1975         rth->rt_genid = rt_genid_ipv4(dev_net(dev_out));
1976         rth->rt_flags   = flags;
1977         rth->rt_type    = type;
1978         rth->rt_is_input = 0;
1979         rth->rt_iif     = orig_oif ? : 0;
1980         rth->rt_pmtu    = 0;
1981         rth->rt_gateway = 0;
1982         rth->rt_uses_gateway = 0;
1983         INIT_LIST_HEAD(&rth->rt_uncached);
1984
1985         RT_CACHE_STAT_INC(out_slow_tot);
1986
1987         if (flags & RTCF_LOCAL)
1988                 rth->dst.input = ip_local_deliver;
1989         if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
1990                 if (flags & RTCF_LOCAL &&
1991                     !(dev_out->flags & IFF_LOOPBACK)) {
1992                         rth->dst.output = ip_mc_output;
1993                         RT_CACHE_STAT_INC(out_slow_mc);
1994                 }
1995 #ifdef CONFIG_IP_MROUTE
1996                 if (type == RTN_MULTICAST) {
1997                         if (IN_DEV_MFORWARD(in_dev) &&
1998                             !ipv4_is_local_multicast(fl4->daddr)) {
1999                                 rth->dst.input = ip_mr_input;
2000                                 rth->dst.output = ip_mc_output;
2001                         }
2002                 }
2003 #endif
2004         }
2005
2006         rt_set_nexthop(rth, fl4->daddr, res, fnhe, fi, type, 0);
2007
2008         return rth;
2009 }
2010
2011 /*
2012  * Major route resolver routine.
2013  */
2014
2015 struct rtable *__ip_route_output_key(struct net *net, struct flowi4 *fl4)
2016 {
2017         struct net_device *dev_out = NULL;
2018         __u8 tos = RT_FL_TOS(fl4);
2019         unsigned int flags = 0;
2020         struct fib_result res;
2021         struct rtable *rth;
2022         int orig_oif;
2023
2024         res.tclassid    = 0;
2025         res.fi          = NULL;
2026         res.table       = NULL;
2027
2028         orig_oif = fl4->flowi4_oif;
2029
2030         fl4->flowi4_iif = LOOPBACK_IFINDEX;
2031         fl4->flowi4_tos = tos & IPTOS_RT_MASK;
2032         fl4->flowi4_scope = ((tos & RTO_ONLINK) ?
2033                              RT_SCOPE_LINK : RT_SCOPE_UNIVERSE);
2034
2035         rcu_read_lock();
2036         if (fl4->saddr) {
2037                 rth = ERR_PTR(-EINVAL);
2038                 if (ipv4_is_multicast(fl4->saddr) ||
2039                     ipv4_is_lbcast(fl4->saddr) ||
2040                     ipv4_is_zeronet(fl4->saddr))
2041                         goto out;
2042
2043                 /* I removed check for oif == dev_out->oif here.
2044                    It was wrong for two reasons:
2045                    1. ip_dev_find(net, saddr) can return wrong iface, if saddr
2046                       is assigned to multiple interfaces.
2047                    2. Moreover, we are allowed to send packets with saddr
2048                       of another iface. --ANK
2049                  */
2050
2051                 if (fl4->flowi4_oif == 0 &&
2052                     (ipv4_is_multicast(fl4->daddr) ||
2053                      ipv4_is_lbcast(fl4->daddr))) {
2054                         /* It is equivalent to inet_addr_type(saddr) == RTN_LOCAL */
2055                         dev_out = __ip_dev_find(net, fl4->saddr, false);
2056                         if (!dev_out)
2057                                 goto out;
2058
2059                         /* Special hack: user can direct multicasts
2060                            and limited broadcast via necessary interface
```

```
2061                         without fiddling with IP_MULTICAST_IF or IP_PKTINFO.
2062                         This hack is not just for fun, it allows
2063                         vic,vat and friends to work.
2064                         They bind socket to loopback, set ttl to zero
2065                         and expect that it will work.
2066                         From the viewpoint of routing cache they are broken,
2067                         because we are not allowed to build multicast path
2068                         with loopback source addr (look, routing cache
2069                         cannot know, that ttl is zero, so that packet
2070                         will not leave this host and route is valid).
2071                         Luckily, this hack is good workaround.
2072                       */
2073
2074                 fl4->flowi4_oif = dev_out->ifindex;
2075                 goto make_route;
2076         }
2077
2078         if (!(fl4->flowi4_flags & FLOWI_FLAG_ANYSRC)) {
2079                 /* It is equivalent to inet_addr_type(saddr) == RTN_LOCAL */
2080                 if (!__ip_dev_find(net, fl4->saddr, false))
2081                         goto out;
2082         }
2083 }
2084
2085
2086 if (fl4->flowi4_oif) {
2087         dev_out = dev_get_by_index_rcu(net, fl4->flowi4_oif);
2088         rth = ERR_PTR(-ENODEV);
2089         if (!dev_out)
2090                 goto out;
2091
2092         /* RACE: Check return value of inet_select_addr instead. */
2093         if (!(dev_out->flags & IFF_UP) || !__in_dev_get_rcu(dev_out)) {
2094                 rth = ERR_PTR(-ENETUNREACH);
2095                 goto out;
2096         }
2097         if (ipv4_is_local_multicast(fl4->daddr) ||
2098             ipv4_is_lbcast(fl4->daddr) ||
2099             fl4->flowi4_proto == IPPROTO_IGMP) {
2100                 if (!fl4->saddr)
2101                         fl4->saddr = inet_select_addr(dev_out, 0,
2102                                                       RT_SCOPE_LINK);
2103                 goto make_route;
2104         }
2105         if (!fl4->saddr) {
2106                 if (ipv4_is_multicast(fl4->daddr))
2107                         fl4->saddr = inet_select_addr(dev_out, 0,
2108                                                       fl4->flowi4_scope);
2109                 else if (!fl4->daddr)
2110                         fl4->saddr = inet_select_addr(dev_out, 0,
2111                                                       RT_SCOPE_HOST);
2112         }
2113 }
2114
2115 if (!fl4->daddr) {
2116         fl4->daddr = fl4->saddr;
2117         if (!fl4->daddr)
2118                 fl4->daddr = fl4->saddr = htonl(INADDR_LOOPBACK);
2119         dev_out = net->loopback_dev;
2120         fl4->flowi4_oif = LOOPBACK_IFINDEX;
2121         res.type = RTN_LOCAL;
2122         flags |= RTCF_LOCAL;
2123         goto make_route;
2124 }
2125
2126 if (fib_lookup(net, fl4, &res, 0)) {
2127         res.fi = NULL;
2128         res.table = NULL;
2129         if (fl4->flowi4_oif) {
2130                 /* Apparently, routing tables are wrong. Assume,
2131                    that the destination is on link.
2132
2133                    WHY? DW.
2134                    Because we are allowed to send to iface
2135                    even if it has NO routes and NO assigned
2136                    addresses. When oif is specified, routing
2137                    tables are looked up with only one purpose:
2138                    to catch if destination is gatewayed, rather than
2139                    direct. Moreover, if MSG_DONTROUTE is set,
2140                    we send packet, ignoring both routing tables
2141                    and ifaddr state. --ANK
2142
2143
2144                    We could make it even if oif is unknown,
2145                    likely IPv6, but we do not.
2146                  */
2147
2148                 if (fl4->saddr == 0)
2149                         fl4->saddr = inet_select_addr(dev_out, 0,
2150                                                       RT_SCOPE_LINK);
2151                 res.type = RTN_UNICAST;
2152                 goto make_route;
2153         }
2154         rth = ERR_PTR(-ENETUNREACH);
2155         goto out;
2156 }
2157
2158 if (res.type == RTN_LOCAL) {
2159         if (!fl4->saddr) {
2160                 if (res.fi->fib_prefsrc)
2161                         fl4->saddr = res.fi->fib_prefsrc;
2162                 else
2163                         fl4->saddr = fl4->daddr;
2164         }
```

```
2165                 dev_out = net->loopback_dev;
2166                 fl4->flowi4_oif = dev_out->ifindex;
2167                 flags |= RTCF_LOCAL;
2168                 goto make_route;
2169         }
2170
2171 #ifdef CONFIG_IP_ROUTE_MULTIPATH
2172         if (res.fi->fib_nhs > 1 && fl4->flowi4_oif == 0)
2173                 fib_select_multipath(&res);
2174         else
2175 #endif
2176         if (!res.prefixlen &&
2177             res.table->tb_num_default > 1 &&
2178             res.type == RTN_UNICAST && !fl4->flowi4_oif)
2179                 fib_select_default(fl4, &res);
2180
2181         if (!fl4->saddr)
2182                 fl4->saddr = FIB_RES_PREFSRC(net, res);
2183
2184         dev_out = FIB_RES_DEV(res);
2185         fl4->flowi4_oif = dev_out->ifindex;
2186
2187
2188 make_route:
2189         rth = __mkroute_output(&res, fl4, orig_oif, dev_out, flags);
2190
2191 out:
2192         rcu_read_unlock();
2193         return rth;
2194 }
2195 EXPORT_SYMBOL_GPL(__ip_route_output_key);
2196
2197 static struct dst_entry *ipv4_blackhole_dst_check(struct dst_entry *dst, u32 cookie)
2198 {
2199         return NULL;
2200 }
2201
2202 static unsigned int ipv4_blackhole_mtu(const struct dst_entry *dst)
2203 {
2204         unsigned int mtu = dst_metric_raw(dst, RTAX_MTU);
2205
2206         return mtu ? : dst->dev->mtu;
2207 }
2208
2209 static void ipv4_rt_blackhole_update_pmtu(struct dst_entry *dst, struct sock *sk,
2210                                           struct sk_buff *skb, u32 mtu)
2211 {
2212 }
2213
2214 static void ipv4_rt_blackhole_redirect(struct dst_entry *dst, struct sock *sk,
2215                                        struct sk_buff *skb)
2216 {
2217 }
2218
2219 static u32 *ipv4_rt_blackhole_cow_metrics(struct dst_entry *dst,
2220                                           unsigned long old)
2221 {
2222         return NULL;
2223 }
2224
2225 static struct dst_ops ipv4_dst_blackhole_ops = {
2226         .family                 =       AF_INET,
2227         .check                  =       ipv4_blackhole_dst_check,
2228         .mtu                    =       ipv4_blackhole_mtu,
2229         .default_advmss         =       ipv4_default_advmss,
2230         .update_pmtu            =       ipv4_rt_blackhole_update_pmtu,
2231         .redirect               =       ipv4_rt_blackhole_redirect,
2232         .cow_metrics            =       ipv4_rt_blackhole_cow_metrics,
2233         .neigh_lookup           =       ipv4_neigh_lookup,
2234 };
2235
2236 struct dst_entry *ipv4_blackhole_route(struct net *net, struct dst_entry *dst_orig)
2237 {
2238         struct rtable *ort = (struct rtable *) dst_orig;
2239         struct rtable *rt;
2240
2241         rt = dst_alloc(&ipv4_dst_blackhole_ops, NULL, 1, DST_OBSOLETE_NONE, 0);
2242         if (rt) {
2243                 struct dst_entry *new = &rt->dst;
2244
2245                 new->__use = 1;
2246                 new->input = dst_discard;
2247                 new->output = dst_discard_sk;
2248
2249                 new->dev = ort->dst.dev;
2250                 if (new->dev)
2251                         dev_hold(new->dev);
2252
2253                 rt->rt_is_input = ort->rt_is_input;
2254                 rt->rt_iif = ort->rt_iif;
2255                 rt->rt_pmtu = ort->rt_pmtu;
2256
2257                 rt->rt_genid = rt_genid_ipv4(net);
2258                 rt->rt_flags = ort->rt_flags;
2259                 rt->rt_type = ort->rt_type;
2260                 rt->rt_gateway = ort->rt_gateway;
2261                 rt->rt_uses_gateway = ort->rt_uses_gateway;
2262
2263                 INIT_LIST_HEAD(&rt->rt_uncached);
2264
2265                 dst_free(new);
2266         }
2267
2268         dst_release(dst_orig);
```

```
2269
2270         return rt ? &rt->dst : ERR_PTR(-ENOMEM);
2271 }
2272
2273 struct rtable *ip_route_output_flow(struct net *net, struct flowi4 *flp4,
2274                                     struct sock *sk)
2275 {
2276         struct rtable *rt = __ip_route_output_key(net, flp4);
2277
2278         if (IS_ERR(rt))
2279                 return rt;
2280
2281         if (flp4->flowi4_proto)
2282                 rt = (struct rtable *)xfrm_lookup_route(net, &rt->dst,
2283                                                         flowi4_to_flowi(flp4),
2284                                                         sk, 0);
2285
2286         return rt;
2287 }
2288 EXPORT_SYMBOL_GPL(ip_route_output_flow);
2289
2290 static int rt_fill_info(struct net *net,  __be32 dst, __be32 src,
2291                         struct flowi4 *fl4, struct sk_buff *skb, u32 portid,
2292                         u32 seq, int event, int nowait, unsigned int flags)
2293 {
2294         struct rtable *rt = skb_rtable(skb);
2295         struct rtmsg *r;
2296         struct nlmsghdr *nlh;
2297         unsigned long expires = 0;
2298         u32 error;
2299         u32 metrics[RTAX_MAX];
2300
2301         nlh = nlmsg_put(skb, portid, seq, event, sizeof(*r), flags);
2302         if (!nlh)
2303                 return -EMSGSIZE;
2304
2305         r = nlmsg_data(nlh);
2306         r->rtm_family    = AF_INET;
2307         r->rtm_dst_len  = 32;
2308         r->rtm_src_len  = 0;
2309         r->rtm_tos      = fl4->flowi4_tos;
2310         r->rtm_table    = RT_TABLE_MAIN;
2311         if (nla_put_u32(skb, RTA_TABLE, RT_TABLE_MAIN))
2312                 goto nla_put_failure;
2313         r->rtm_type     = rt->rt_type;
2314         r->rtm_scope    = RT_SCOPE_UNIVERSE;
2315         r->rtm_protocol = RTPROT_UNSPEC;
2316         r->rtm_flags    = (rt->rt_flags & ~0xFFFF) | RTM_F_CLONED;
2317         if (rt->rt_flags & RTCF_NOTIFY)
2318                 r->rtm_flags |= RTM_F_NOTIFY;
2319         if (IPCB(skb)->flags & IPSKB_DOREDIRECT)
2320                 r->rtm_flags |= RTCF_DOREDIRECT;
2321
2322         if (nla_put_in_addr(skb, RTA_DST, dst))
2323                 goto nla_put_failure;
2324         if (src) {
2325                 r->rtm_src_len = 32;
2326                 if (nla_put_in_addr(skb, RTA_SRC, src))
2327                         goto nla_put_failure;
2328         }
2329         if (rt->dst.dev &&
2330             nla_put_u32(skb, RTA_OIF, rt->dst.dev->ifindex))
2331                 goto nla_put_failure;
2332 #ifdef CONFIG_IP_ROUTE_CLASSID
2333         if (rt->dst.tclassid &&
2334             nla_put_u32(skb, RTA_FLOW, rt->dst.tclassid))
2335                 goto nla_put_failure;
2336 #endif
2337         if (!rt_is_input_route(rt) &&
2338             fl4->saddr != src) {
2339                 if (nla_put_in_addr(skb, RTA_PREFSRC, fl4->saddr))
2340                         goto nla_put_failure;
2341         }
2342         if (rt->rt_uses_gateway &&
2343             nla_put_in_addr(skb, RTA_GATEWAY, rt->rt_gateway))
2344                 goto nla_put_failure;
2345
2346         expires = rt->dst.expires;
2347         if (expires) {
2348                 unsigned long now = jiffies;
2349
2350                 if (time_before(now, expires))
2351                         expires -= now;
2352                 else
2353                         expires = 0;
2354         }
2355
2356         memcpy(metrics, dst_metrics_ptr(&rt->dst), sizeof(metrics));
2357         if (rt->rt_pmtu && expires)
2358                 metrics[RTAX_MTU - 1] = rt->rt_pmtu;
2359         if (rtnetlink_put_metrics(skb, metrics) < 0)
2360                 goto nla_put_failure;
2361
2362         if (fl4->flowi4_mark &&
2363             nla_put_u32(skb, RTA_MARK, fl4->flowi4_mark))
2364                 goto nla_put_failure;
2365
2366         error = rt->dst.error;
2367
2368         if (rt_is_input_route(rt)) {
2369 #ifdef CONFIG_IP_MROUTE
2370                 if (ipv4_is_multicast(dst) && !ipv4_is_local_multicast(dst) &&
2371                     IPV4_DEVCONF_ALL(net, MC_FORWARDING)) {
2372                         int err = ipmr_get_route(net, skb,
```

```
2373                                            fl4->saddr, fl4->daddr,
2374                                            r, nowait);
2375                        if (err <= 0) {
2376                                if (!nowait) {
2377                                        if (err == 0)
2378                                                return 0;
2379                                        goto nla_put_failure;
2380                                } else {
2381                                        if (err == -EMSGSIZE)
2382                                                goto nla_put_failure;
2383                                        error = err;
2384                                }
2385                        }
2386                } else
2387 #endif
2388                        if (nla_put_u32(skb, RTA_IIF, skb->dev->ifindex))
2389                                goto nla_put_failure;
2390        }
2391
2392        if (rtnl_put_cacheinfo(skb, &rt->dst, 0, expires, error) < 0)
2393                goto nla_put_failure;
2394
2395        nlmsg_end(skb, nlh);
2396        return 0;
2397
2398 nla_put_failure:
2399        nlmsg_cancel(skb, nlh);
2400        return -EMSGSIZE;
2401 }
2402
2403 static int inet_rtm_getroute(struct sk_buff *in_skb, struct nlmsghdr *nlh)
2404 {
2405        struct net *net = sock_net(in_skb->sk);
2406        struct rtmsg *rtm;
2407        struct nlattr *tb[RTA_MAX+1];
2408        struct rtable *rt = NULL;
2409        struct flowi4 fl4;
2410        __be32 dst = 0;
2411        __be32 src = 0;
2412        u32 iif;
2413        int err;
2414        int mark;
2415        struct sk_buff *skb;
2416
2417        err = nlmsg_parse(nlh, sizeof(*rtm), tb, RTA_MAX, rtm_ipv4_policy);
2418        if (err < 0)
2419                goto errout;
2420
2421        rtm = nlmsg_data(nlh);
2422
2423        skb = alloc_skb(NLMSG_GOODSIZE, GFP_KERNEL);
2424        if (!skb) {
2425                err = -ENOBUFS;
2426                goto errout;
2427        }
2428
2429        /* Reserve room for dummy headers, this skb can pass
2430           through good chunk of routing engine.
2431            */
2432        skb_reset_mac_header(skb);
2433        skb_reset_network_header(skb);
2434
2435        /* Bugfix: need to give ip_route_input enough of an IP header to not gag. */
2436        ip_hdr(skb)->protocol = IPPROTO_ICMP;
2437        skb_reserve(skb, MAX_HEADER + sizeof(struct iphdr));
2438
2439        src = tb[RTA_SRC] ? nla_get_in_addr(tb[RTA_SRC]) : 0;
2440        dst = tb[RTA_DST] ? nla_get_in_addr(tb[RTA_DST]) : 0;
2441        iif = tb[RTA_IIF] ? nla_get_u32(tb[RTA_IIF]) : 0;
2442        mark = tb[RTA_MARK] ? nla_get_u32(tb[RTA_MARK]) : 0;
2443
2444        memset(&fl4, 0, sizeof(fl4));
2445        fl4.daddr = dst;
2446        fl4.saddr = src;
2447        fl4.flowi4_tos = rtm->rtm_tos;
2448        fl4.flowi4_oif = tb[RTA_OIF] ? nla_get_u32(tb[RTA_OIF]) : 0;
2449        fl4.flowi4_mark = mark;
2450
2451        if (iif) {
2452                struct net_device *dev;
2453
2454                dev = __dev_get_by_index(net, iif);
2455                if (!dev) {
2456                        err = -ENODEV;
2457                        goto errout_free;
2458                }
2459
2460                skb->protocol   = htons(ETH_P_IP);
2461                skb->dev        = dev;
2462                skb->mark       = mark;
2463                local_bh_disable();
2464                err = ip_route_input(skb, dst, src, rtm->rtm_tos, dev);
2465                local_bh_enable();
2466
2467                rt = skb_rtable(skb);
2468                if (err == 0 && rt->dst.error)
2469                        err = -rt->dst.error;
2470        } else {
2471                rt = ip_route_output_key(net, &fl4);
2472
2473                err = 0;
2474                if (IS_ERR(rt))
2475                        err = PTR_ERR(rt);
2476        }
```

```
2477
2478         if (err)
2479                 goto errout_free;
2480
2481         skb_dst_set(skb, &rt->dst);
2482         if (rtm->rtm_flags & RTM_F_NOTIFY)
2483                 rt->rt_flags |= RTCF_NOTIFY;
2484
2485         err = rt_fill_info(net, dst, src, &fl4, skb,
2486                            NETLINK_CB(in_skb).portid, nlh->nlmsg_seq,
2487                            RTM_NEWROUTE, 0, 0);
2488         if (err < 0)
2489                 goto errout_free;
2490
2491         err = rtnl_unicast(skb, net, NETLINK_CB(in_skb).portid);
2492 errout:
2493         return err;
2494
2495 errout_free:
2496         kfree_skb(skb);
2497         goto errout;
2498 }
2499
2500 void ip_rt_multicast_event(struct in_device *in_dev)
2501 {
2502         rt_cache_flush(dev_net(in_dev->dev));
2503 }
2504
2505 #ifdef CONFIG_SYSCTL
2506 static int ip_rt_gc_timeout __read_mostly       = RT_GC_TIMEOUT;
2507 static int ip_rt_gc_interval __read_mostly  = 60 * HZ;
2508 static int ip_rt_gc_min_interval __read_mostly  = HZ / 2;
2509 static int ip_rt_gc_elasticity __read_mostly    = 8;
2510
2511 static int ipv4_sysctl_rtcache_flush(struct ctl_table *__ctl, int write,
2512                                      void __user *buffer,
2513                                      size_t *lenp, loff_t *ppos)
2514 {
2515         struct net *net = (struct net *)__ctl->extra1;
2516
2517         if (write) {
2518                 rt_cache_flush(net);
2519                 fnhe_genid_bump(net);
2520                 return 0;
2521         }
2522
2523         return -EINVAL;
2524 }
2525
2526 static struct ctl_table ipv4_route_table[] = {
2527         {
2528                 .procname       = "gc_thresh",
2529                 .data           = &ipv4_dst_ops.gc_thresh,
2530                 .maxlen         = sizeof(int),
2531                 .mode           = 0644,
2532                 .proc_handler   = proc_dointvec,
2533         },
2534         {
2535                 .procname       = "max_size",
2536                 .data           = &ip_rt_max_size,
2537                 .maxlen         = sizeof(int),
2538                 .mode           = 0644,
2539                 .proc_handler   = proc_dointvec,
2540         },
2541         {
2542                 /*  Deprecated. Use gc_min_interval_ms */
2543
2544                 .procname       = "gc_min_interval",
2545                 .data           = &ip_rt_gc_min_interval,
2546                 .maxlen         = sizeof(int),
2547                 .mode           = 0644,
2548                 .proc_handler   = proc_dointvec_jiffies,
2549         },
2550         {
2551                 .procname       = "gc_min_interval_ms",
2552                 .data           = &ip_rt_gc_min_interval,
2553                 .maxlen         = sizeof(int),
2554                 .mode           = 0644,
2555                 .proc_handler   = proc_dointvec_ms_jiffies,
2556         },
2557         {
2558                 .procname       = "gc_timeout",
2559                 .data           = &ip_rt_gc_timeout,
2560                 .maxlen         = sizeof(int),
2561                 .mode           = 0644,
2562                 .proc_handler   = proc_dointvec_jiffies,
2563         },
2564         {
2565                 .procname       = "gc_interval",
2566                 .data           = &ip_rt_gc_interval,
2567                 .maxlen         = sizeof(int),
2568                 .mode           = 0644,
2569                 .proc_handler   = proc_dointvec_jiffies,
2570         },
2571         {
2572                 .procname       = "redirect_load",
2573                 .data           = &ip_rt_redirect_load,
2574                 .maxlen         = sizeof(int),
2575                 .mode           = 0644,
2576                 .proc_handler   = proc_dointvec,
2577         },
2578         {
2579                 .procname       = "redirect_number",
2580                 .data           = &ip_rt_redirect_number,
```

```
2581                .maxlen         = sizeof(int),
2582                .mode           = 0644,
2583                .proc_handler   = proc_dointvec,
2584        },
2585        {
2586                .procname       = "redirect_silence",
2587                .data           = &ip_rt_redirect_silence,
2588                .maxlen         = sizeof(int),
2589                .mode           = 0644,
2590                .proc_handler   = proc_dointvec,
2591        },
2592        {
2593                .procname       = "error_cost",
2594                .data           = &ip_rt_error_cost,
2595                .maxlen         = sizeof(int),
2596                .mode           = 0644,
2597                .proc_handler   = proc_dointvec,
2598        },
2599        {
2600                .procname       = "error_burst",
2601                .data           = &ip_rt_error_burst,
2602                .maxlen         = sizeof(int),
2603                .mode           = 0644,
2604                .proc_handler   = proc_dointvec,
2605        },
2606        {
2607                .procname       = "gc_elasticity",
2608                .data           = &ip_rt_gc_elasticity,
2609                .maxlen         = sizeof(int),
2610                .mode           = 0644,
2611                .proc_handler   = proc_dointvec,
2612        },
2613        {
2614                .procname       = "mtu_expires",
2615                .data           = &ip_rt_mtu_expires,
2616                .maxlen         = sizeof(int),
2617                .mode           = 0644,
2618                .proc_handler   = proc_dointvec_jiffies,
2619        },
2620        {
2621                .procname       = "min_pmtu",
2622                .data           = &ip_rt_min_pmtu,
2623                .maxlen         = sizeof(int),
2624                .mode           = 0644,
2625                .proc_handler   = proc_dointvec,
2626        },
2627        {
2628                .procname       = "min_adv_mss",
2629                .data           = &ip_rt_min_advmss,
2630                .maxlen         = sizeof(int),
2631                .mode           = 0644,
2632                .proc_handler   = proc_dointvec,
2633        },
2634        { }
2635 };
2636
2637 static struct ctl_table ipv4_route_flush_table[] = {
2638        {
2639                .procname       = "flush",
2640                .maxlen         = sizeof(int),
2641                .mode           = 0200,
2642                .proc_handler   = ipv4_sysctl_rtcache_flush,
2643        },
2644        { },
2645 };
2646
2647 static __net_init int sysctl_route_net_init(struct net *net)
2648 {
2649        struct ctl_table *tbl;
2650
2651        tbl = ipv4_route_flush_table;
2652        if (!net_eq(net, &init_net)) {
2653                tbl = kmemdup(tbl, sizeof(ipv4_route_flush_table), GFP_KERNEL);
2654                if (!tbl)
2655                        goto err_dup;
2656
2657                /* Don't export sysctls to unprivileged users */
2658                if (net->user_ns != &init_user_ns)
2659                        tbl[0].procname = NULL;
2660        }
2661        tbl[0].extra1 = net;
2662
2663        net->ipv4.route_hdr = register_net_sysctl(net, "net/ipv4/route", tbl);
2664        if (!net->ipv4.route_hdr)
2665                goto err_reg;
2666        return 0;
2667
2668 err_reg:
2669        if (tbl != ipv4_route_flush_table)
2670                kfree(tbl);
2671 err_dup:
2672        return -ENOMEM;
2673 }
2674
2675 static __net_exit void sysctl_route_net_exit(struct net *net)
2676 {
2677        struct ctl_table *tbl;
2678
2679        tbl = net->ipv4.route_hdr->ctl_table_arg;
2680        unregister_net_sysctl_table(net->ipv4.route_hdr);
2681        BUG_ON(tbl == ipv4_route_flush_table);
2682        kfree(tbl);
2683 }
2684
```

```
2685 static __net_initdata struct pernet_operations sysctl_route_ops = {
2686         .init = sysctl_route_net_init,
2687         .exit = sysctl_route_net_exit,
2688 };
2689 #endif
2690
2691 static __net_init int rt_genid_init(struct net *net)
2692 {
2693         atomic_set(&net->ipv4.rt_genid, 0);
2694         atomic_set(&net->fnhe_genid, 0);
2695         get_random_bytes(&net->ipv4.dev_addr_genid,
2696                          sizeof(net->ipv4.dev_addr_genid));
2697         return 0;
2698 }
2699
2700 static __net_initdata struct pernet_operations rt_genid_ops = {
2701         .init = rt_genid_init,
2702 };
2703
2704 static int __net_init ipv4_inetpeer_init(struct net *net)
2705 {
2706         struct inet_peer_base *bp = kmalloc(sizeof(*bp), GFP_KERNEL);
2707
2708         if (!bp)
2709                 return -ENOMEM;
2710         inet_peer_base_init(bp);
2711         net->ipv4.peers = bp;
2712         return 0;
2713 }
2714
2715 static void __net_exit ipv4_inetpeer_exit(struct net *net)
2716 {
2717         struct inet_peer_base *bp = net->ipv4.peers;
2718
2719         net->ipv4.peers = NULL;
2720         inetpeer_invalidate_tree(bp);
2721         kfree(bp);
2722 }
2723
2724 static __net_initdata struct pernet_operations ipv4_inetpeer_ops = {
2725         .init   =       ipv4_inetpeer_init,
2726         .exit   =       ipv4_inetpeer_exit,
2727 };
2728
2729 #ifdef CONFIG_IP_ROUTE_CLASSID
2730 struct ip_rt_acct __percpu *ip_rt_acct __read_mostly;
2731 #endif /* CONFIG_IP_ROUTE_CLASSID */
2732
2733 int __init ip_rt_init(void)
2734 {
2735         int rc = 0;
2736         int cpu;
2737
2738         ip_idents = kmalloc(IP_IDENTS_SZ * sizeof(*ip_idents), GFP_KERNEL);
2739         if (!ip_idents)
2740                 panic("IP: failed to allocate ip_idents\n");
2741
2742         prandom_bytes(ip_idents, IP_IDENTS_SZ * sizeof(*ip_idents));
2743
2744         ip_tstamps = kcalloc(IP_IDENTS_SZ, sizeof(*ip_tstamps), GFP_KERNEL);
2745         if (!ip_tstamps)
2746                 panic("IP: failed to allocate ip_tstamps\n");
2747
2748         for_each_possible_cpu(cpu) {
2749                 struct uncached_list *ul = &per_cpu(rt_uncached_list, cpu);
2750
2751                 INIT_LIST_HEAD(&ul->head);
2752                 spin_lock_init(&ul->lock);
2753         }
2754 #ifdef CONFIG_IP_ROUTE_CLASSID
2755         ip_rt_acct = __alloc_percpu(256 * sizeof(struct ip_rt_acct), __alignof__(struct ip_rt_acct));
2756         if (!ip_rt_acct)
2757                 panic("IP: failed to allocate ip_rt_acct\n");
2758 #endif
2759
2760         ipv4_dst_ops.kmem_cachep =
2761                 kmem_cache_create("ip_dst_cache", sizeof(struct rtable), 0,
2762                                   SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
2763
2764         ipv4_dst_blackhole_ops.kmem_cachep = ipv4_dst_ops.kmem_cachep;
2765
2766         if (dst_entries_init(&ipv4_dst_ops) < 0)
2767                 panic("IP: failed to allocate ipv4_dst_ops counter\n");
2768
2769         if (dst_entries_init(&ipv4_dst_blackhole_ops) < 0)
2770                 panic("IP: failed to allocate ipv4_dst_blackhole_ops counter\n");
2771
2772         ipv4_dst_ops.gc_thresh = ~0;
2773         ip_rt_max_size = INT_MAX;
2774
2775         devinet_init();
2776         ip_fib_init();
2777
2778         if (ip_rt_proc_init())
2779                 pr_err("Unable to create route proc files\n");
2780 #ifdef CONFIG_XFRM
2781         xfrm_init();
2782         xfrm4_init();
2783 #endif
2784         rtnl_register(PF_INET, RTM_GETROUTE, inet_rtm_getroute, NULL, NULL);
2785
2786 #ifdef CONFIG_SYSCTL
2787         register_pernet_subsys(&sysctl_route_ops);
2788 #endif
```

```
2789         register_pernet_subsys(&rt_genid_ops);
2790         register_pernet_subsys(&ipv4_inetpeer_ops);
2791         return rc;
2792 }
2793
2794 #ifdef CONFIG_SYSCTL
2795 /*
2796  * We really need to sanitize the damn ipv4 init order, then all
2797  * this nonsense will go away.
2798  */
2799 void __init ip_static_sysctl_init(void)
2800 {
2801         register_net_sysctl(&init_net, "net/ipv4/route", ipv4_route_table);
2802 }
2803 #endif
2804
```