IP Initialization

The IP module is initialized when `ip_init()` is called from the last few lines of `inet_init()`. This function is defined in `net/ipv4/ip_output.c` and performs three major functions:

- The call to `ip_rt_init()` initializes both the route cache in which a hash structure provides fast access by destination IP address to the routing entities describing the next hop and the FIB (Forwarding Information Base) which is the internal representation of the routing table.

- The `ip_rt_init()` function also calls `ip_fib_init()` to initialize the upper level routing structures.

- The call to `inet_initpeers()` initializes the AVL tree used to keep track of IP peers, hosts with which this host has recently exchanged packets.

```c
1409 void __init ip_init(void)  
1410 {  
1411     ip_rt_init();  
1412     inet_initpeers();  
1413     #if defined(CONFIG_IP_MULTICAST) && defined(CONFIG_PROC_FS)  
1414         igmp_mc_proc_init();  
1415     #endif  
1416 } 
```
Routing overview

Routing in Linux is comprised of a two level system. The upper level is the FIB (Forwarding Information Base).

Entries in the FIB correspond roughly to entries in a the standard routing table.

For a host system is quite small and consists of three types of entry:
- host
- network
- default

<table>
<thead>
<tr>
<th>Destination</th>
<th>Gateway</th>
<th>Genmask</th>
<th>Flags</th>
<th>Metric</th>
<th>Ref</th>
<th>Use</th>
<th>Iface</th>
</tr>
</thead>
<tbody>
<tr>
<td>130.127.88.0</td>
<td>0.0.0.0</td>
<td>255.255.255.224</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>eth1</td>
</tr>
<tr>
<td>192.168.129.0</td>
<td>0.0.0.0</td>
<td>255.255.255.0</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>vmnet1</td>
</tr>
<tr>
<td>192.168.70.0</td>
<td>192.168.1.1</td>
<td>255.255.255.0</td>
<td>UG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>eth0</td>
</tr>
<tr>
<td>192.168.80.0</td>
<td>192.168.1.1</td>
<td>255.255.255.0</td>
<td>UG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>eth0</td>
</tr>
<tr>
<td>192.168.110.0</td>
<td>0.0.0.0</td>
<td>255.255.255.0</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>eth0</td>
</tr>
<tr>
<td>192.168.122.0</td>
<td>0.0.0.0</td>
<td>255.255.255.0</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>virbr0</td>
</tr>
<tr>
<td>192.168.8.0</td>
<td>0.0.0.0</td>
<td>255.255.255.0</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>vmnet8</td>
</tr>
<tr>
<td>169.254.0.0</td>
<td>0.0.0.0</td>
<td>255.255.0.0</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>eth0</td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>130.127.88.1</td>
<td>0.0.0.0</td>
<td>UG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>eth1</td>
</tr>
</tbody>
</table>

In the above table the default entry shown in blue is used to reach all addresses outside Clemson.

The lower layer is the route cache there is a single route cache entry for every remote destination that this host has recently accessed.

When routing is performed, the cache is searched first, and if the cache search fails:

- the FIB is searched
- a new route cache entry is created.
Creating the Routing Cache

The `ip_rt_init()` function defined in `net/ipv4/route.c` is called from `ip_init()` function. This function is used to initialize the IP route cache.

Linux keeps a record of every specific destination that is currently in use or has been used recently in a hash table. For example, a host actively being used in web surfing might have recently accessed 100 different destinations with all destinations using the default route. These 100 destinations would be represented by 100 different route cache entries. In contrast, the default route is represented by a single FIB (routing) table entry.

Each distinct active destination address is described by an instance of `struct rtable`. This structure is defined in `include/net/route.h`. 
The struct rtable

At first, it might seem that it is a wasteful exercise to have a route cache entry per remote destination. If all that is needed is a next hop MAC address why not just keep that in a conventional tree structured routing table. As we will see, the route cache element holds much more than just the next hop.

Instances of the structure are queued in a dynamically allocated hash table with the look up key being a function of (src, dst, iif, oif, tos and scope) fields. Note the odd use of the union that permits pointers of type *rtable and *dst_entry to be used interchangeably. Fields of type __u32 are IP addresses.

```c
52 struct rtable
53 { 
54        union
55        { 
56             struct dst_entry dst;
57         } u;
58
59         /* Cache lookup keys */
60         struct flowi fl;   /* used to be rt_key */
61
62         struct in_device *idev;
63
64         int rt_genid;
65         unsigned rt_flags;
66         __u16 rt_type;
67
68         __be32 rt_dst;    /* Path destination */
69         __be32 rt_src;    /* Path source */
70         int rt_iif;
71
72         /* Info on neighbour */
73         __be32 rt_gateway;
74
75         /* Miscellaneous cached information */
76         __be32 rt_spec_dst; /* RFC1122 specific destination */
77         struct inet_peer *peer; /* long-living peer info */
78     };
79
```
The embedded *dst_entry* structure

An instance of the destination cache (*dst_entry*) structure which is defined in *include/net/dst.h* is embedded in each *struct rtable* and contains pointers to destination-specific input and output functions and data.

```c
38 struct dst_entry
39 {
40     struct rcu_head rcu_head;
41     struct dst_entry *child;
42     struct net_device *dev;
43     short error;
44     short obsolete;
45     int flags;
46 #define DST_HOST 1
47 #define DST_NOXFRM 2
48 #define DST_NOPOLICY 4
49 #define DST_NOHASH 8
50     unsigned long expires;
51
52     unsigned short header_len; /* more space at head required */
53     unsigned short trailer_len; /* space to reserve at tail */
54
55     unsigned int rate_tokens;
56     unsigned long rate_last; /* rate limiting for ICMP */
57
58     struct dst_entry *path;
59
60     struct neighbour *neighbour;
61     struct hh_cache *hh;
62     struct xfrm_state *xfrm;
63
64     int (*input)(struct sk_buff*);
65     int (*output)(struct sk_buff*);
66
```
Some elements of this structure which are presently understood include:

- **dev**  
  output device for this route

- **neighbour**  
  This is a pointer to the ARP cache neighbour structure for this route.

- **hh**  
  A pointer to a hardware header cache element; All routes with a common first hop would use the same hh cache element. The structure **contains the link header** to be used and **a function pointer to be called when a packet is to be physically transmitted.**

- **input**  
  A pointer to the post-routing input function for this route. This function is set during the routing process.

- **output**  
  A pointer to the output function for this route. This function is called just after routing and is **not the same as the function** pointed to by the hh_cache structure.

- **ops**  
  pointer to a statically allocated structure containing family, protocol, and check, reroute and destroy functions for this route (really all IPV4 routes).
The route key structure

```c
13 struct flowi {
14    int oif;
15    int iif;
16
17    union {
18        struct {
19            __u32 daddr;
20            __u32 saddr;
21            __u32 fwmark;
22            __u8 tos;
23            __u8 scope;
24        } ip4_u;
25
26        struct {
27            struct in6_addr daddr;
28            struct in6_addr saddr;
29            __u32 flowlabel;
30        } ip6_u;
31
32        struct {
33            __le16 daddr;
34            __le16 saddr;
35            __u32 fwmark;
36            __u8 scope;
37        } dn_u;
38    } nl_u;
39 #define fld_dst         nl_u.dn_u.daddr
40 #define fld_src         nl_u.dn_u.saddr
41 #define fld_fwmark      nl_u.dn_u.fwmark
42 #define fld_scope       nl_u.dn_u.scope
43 #define fl6_dst         nl_u.ip6_u.daddr
44 #define fl6_src         nl_u.ip6_u.saddr
45 #define fl6_flowlabel   nl_u.ip6_u.flowlabel
46 #define fl4_dst         nl_u.ip4_u.daddr
47 #define fl4_src         nl_u.ip4_u.saddr
48 #define fl4_flowmark    nl_u.ip4_u.fwmark
49 #define fl4_tos         nl_u.ip4_u.tos
50 #define fl4_scope       nl_u.ip4_u.scope
51```

__u8 proto;
__u8 flags;
#define FLOWI_FLAG_MULTIPATHOLDROUTE 0x01
union {
    struct {
        __u16   sport;
        __u16   dport;
    } ports;
    struct {
        __u8    type;
        __u8    code;
    } icmpt;
    struct {
        __le16  sport;
        __le16  dport;
        __u8    objnum;
        __u8    objnamel; /* Not 16 bits since max val is 16 */
        __u8    objname[16]; /* Not zero terminated */
    } dnports;
    __u32    spi;
} uli_u;
define fl_ip_sport    uli_u.ports.sport
define fl_ip_dport    uli_u.ports.dport
define fl_icmp_type   uli_u.icmpt.type
define fl_icmp_code   uli_u.icmpt.code
define fl_ipsec_spi   uli_u.spi
Initializing a route key

The following code can be used to correctly initialize a route key given that a socket is already connected.

```c
struct flowi fl =
    { .oif = sk ? sk->sk_bound_dev_if : 0,
      .nl_u = { .ip4_u =
        { .daddr = inet_sk(sk)->daddr,
          .saddr = inet_sk(sk)->saddr,
          .tos = 0} },
      .proto = IPPROTO_COP,
      .uli_u = { .ports =
        { .sport = inet_sk(sk)->sport,
          .dport = inet_sk(sk)->dport } } };
```
The hash queues

The data structures used in managing the hash queues of struct rtable are shown here. The hash table is accessed via the static pointer rt_hash_table. Each element of the table used to be 8 bytes long and contain a lock in addition to the queue pointer. This provided a very fine granularity of locking when updating the table on a multiprocessor. Now that scheme has been replaced by rcu protection. The number of buckets is determined by the number of physical pages present in memory and is reflected in the variable rt_hash_mask. The number of spinlocks is a function of the number of CPUs.

191/* The locking scheme is rather straight forward:
192 *
193 * 1) Read-Copy Update protects the buckets of the central route hash.
194 * 2) Only writers remove entries, and they hold the lock as they look at rtable reference counts.
196 * 3) Only readers acquire references to rtable entries, they do so with atomic increments and with the lock held.
199 */
200
201 struct rt_hash_bucket {
202 struct rtable *chain;
203 }
204 #if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK) || \
205 defined(CONFIG_PROVE_LOCKING)
206 /*
207 * Instead of using one spinlock for each rt_hash_bucket, we use a table of spinlocks
208 * The size of this table is a power of two and depends on the number of CPUs.
209 * (on lockdep we have a quite big spinlock_t, so keep the size down there)
210 */

211 #ifdef CONFIG_LOCKDEP
212 # define RT_HASH_LOCK_SZ 256
213 #else
214 # if NR_CPUS >= 32
215 # define RT_HASH_LOCK_SZ 4096
216 # elif NR_CPUS >= 16
217 # define RT_HASH_LOCK_SZ 2048
218 # elif NR_CPUS >= 8
219 # define RT_HASH_LOCK_SZ 1024
220 # elif NR_CPUS >= 4
221 # define RT_HASH_LOCK_SZ 512
222 # else
223 # define RT_HASH_LOCK_SZ 256
224 # endif
225 # endif
226
227static spinlock_t *rt_hash_locks;
Route cache management

The `struct dst_ops` is the root structure used in the management of a route cache entries. In addition to function pointers for querying link state and updating the cache because of changes in link state, it contains a pointer to the slab allocator cache of `struct rtable` elements in the element `kmem_cachep`.

```c
83 struct dst_ops
84 {
85        unsigned short       family;
86        unsigned short       protocol;
87        unsigned             gc_thresh;
88
89        int                  (*gc)(void);
90        struct dst_entry *   (*check)(struct dst_entry *,
                               __u32 cookie);
91        void                 (*destroy)(struct dst_entry *);
92        void                 (*ifdown)(struct dst_entry *,
                               struct net_device *dev,
                               int how);
93        struct dst_entry *   (*negative_advice)(struct
                               dst_entry *);
94        void                 (*link_failure)(struct sk_buff *);
95        void                 (*update_pmtu)(struct dst_entry
                               *dst, u32 mtu);
96
97        int                  entry_size;
98
99        atomic_t             entries;
100       kmem_cache_t         *kmem_cachep;
101    }
```
The *ipv4* struct dst_ops

This structure is defined in `net/ipv4/route.c` and also contains the statically initialized elements shown below.

```c
157 static struct dst_ops ipv4_dst_ops = {
158     .family = AF_INET,
159     .protocol = __constant_htons(ETH_P_IP),
160     .gc = rt_garbage_collect,
161     .check = ipv4_dst_check,
162     .destroy = ipv4_dst_destroy,
163     .ifdown = ipv4_dst_ifdown,
164     .negative_advice = ipv4_negative_advice,
165     .link_failure = ipv4_link_failure,
166     .update_pmtu = ip_rt_update_pmtu,
167     .entry_size = sizeof(struct rtable),
168 };
```

These static constants declared in `route.c` identify the location and size of the route cache.

```c
246 static struct rt_hash_bucket *rt_hash_table;
247 static unsigned rt_hash_mask;
248 static int rt_hash_log;
249 static unsigned int rt_hash_rnd;
250```
The `ip_rt_init()` function

```c
3126int __init ip_rt_init(void)
3127{
3128    int rc = 0;
3129    rt_hash_rnd = (int) ((num_physpages ^ (num_physpages>>8)) ^
3130        (jiffies ^ (jiffies >> 7)));
3131
This call creates the cache from which elements of `struct rtable` are allocated.

3145ipv4_dst_ops.kmem_cachep = kmem_cache_create("ip_dst_cache",
3146    sizeof(struct rtable),
3147    0, SLAB_HWCACHE_ALIGN,
3148    NULL, NULL);
3149
3150    if (!ipv4_dst_ops.kmem_cachep)
3151        panic("IP: failed to allocate ip_dst_cache\n");
3152
This call creates the hash table through which active elements are accessed.

3154    rt_hash_table = (struct rt_hash_bucket *)
3155     alloc_large_system_hash("IP route cache",
3156     sizeof(struct rt_hash_bucket),
3157     rhash_entries,
3158     (num_physpages >= 128 * 1024) ?
3159     15 : 17,
3160     0,
3161     &rt_hash_log,
3162     &rt_hash_mask,
3163     0);
3164    memset(rt_hash_table, 0, (rt_hash_mask + 1) * sizeof(struct
3165     rt_hash_bucket));
```

This call creates the cache from which elements of `struct rtable` are allocated.

```c
3145ipv4_dst_ops.kmem_cachep = kmem_cache_create("ip_dst_cache",
3146    sizeof(struct rtable),
3147    0, SLAB_HWCACHE_ALIGN,
3148    NULL, NULL);
3149
3150    if (!ipv4_dst_ops.kmem_cachep)
3151        panic("IP: failed to allocate ip_dst_cache\n");
3152
This call creates the hash table through which active elements are accessed.

```c
3154    rt_hash_table = (struct rt_hash_bucket *)
3155     alloc_large_system_hash("IP route cache",
3156     sizeof(struct rt_hash_bucket),
3157     rhash_entries,
3158     (num_physpages >= 128 * 1024) ?
3159     15 : 17,
3160     0,
3161     &rt_hash_log,
3162     &rt_hash_mask,
3163     0);
3164    memset(rt_hash_table, 0, (rt_hash_mask + 1) * sizeof(struct
3165     rt_hash_bucket));
```
rt_hash_lock_init();

ipv4_dst_ops.gc_thresh = (rt_hash_mask + 1);
ip_rt_max_size = (rt_hash_mask + 1) * 16;

devinet_init();
ip_fib_init();<---------- Init fib table

init_timer(&rt_flush_timer);
rt_flush_timer.function = rt_run_flush;
init_timer(&rt_periodic_timer);
rt_periodic_timer.function = rt_check_expire;
init_timer(&rt_secret_timer);
rt_secret_timer.function = rt_secret_rebuild;

/* All the timers, started at system startup tend to synchronize. Perturb it a bit. */
rt_periodic_timer.expires = jiffies + net_random() % ip_rt_gc_interval + ip_rt_gc_interval;
add_timer(&rt_periodic_timer);

rt_secret_timer.expires = jiffies + net_random() % ip_rt_secret_interval + ip_rt_secret_interval;
add_timer(&rt_secret_timer);

return rc;
Setting up the Forwarding Information Base (FIB).

The FIB is the internal representation of the routing table. The routing table, and thus the contents of the FIB, may be viewed by running the `/sbin/route` command. This complex and important routing structure contains the routing information needed to reach any valid IP address via its network address and netmask.

When an outgoing packet is to be routed, the IP layer:

- first checks to see if the destination address is in the routing cache (discussed earlier in this section).
- If not, the FIB must be searched for a (destination, netmask) combination that matches the target destination address.
- The table is searched using the standard strategy that longest matching mask wins. If a match is found, the routing cache is updated and the packet is sent on its way.

Initialization of the FIB is performed a call to

```
2507 ip_fib_init();
```
The `ip_fib_init()` function

The `ip_fib_init()` function is defined in `net/ipv4/fib_frontend.c`. In the standard configuration, this function references two global variables `struct fib_table *local_table, *main_table` defined in `net/ipv4/fib_frontend.c`. These pointers are set up to point to a dynamically allocated area of kernel memory that contains a fixed size structure of type `struct fib_table` followed a hash table with a single entry for each possible number of bits in a network mask.

- The contents of the `main_table` represent the remote IP addresses defined in routing table and may be viewed in `/proc/net/route` (as well as by using `/sbin/route`).

- The contents of the `local_table` reflect those IP addresses that exist on this computer.

```c
void __init ip_fib_init(void)
{
    #ifndef CONFIG_IP_MULTIPLE_TABLES
    ip_fib_local_table = fib_hash_init(RT_TABLE_LOCAL);
    ip_fib_main_table  = fib_hash_init(RT_TABLE_MAIN);
    #else
    fib_rules_init();
    #endif

    register_netdevice_notifier(&fib_netdev_notifier);
    register_inetaddr_notifier(&fib_inetaddr_notifier);
    nl_fib_lookup_init();
}
```
The *fib_table*

The *fib_table* structure defined in *include/net/ip_fib.h*. Like the *struct dst_ops()* the system is designed to support polymorphic behavior in which table management functions are user definable and or replaceable.

For example, the *tb_lookup()* element points to the that will actually be used to search the table.

```
116 struct fib_table
117 {
118     unsigned char  tb_id; /* local / main */
119     unsigned      tb_stamp;
120     int (*tb_lookup)(struct fib_table *tb,
121                         const struct rt_key *key,
122                         struct fib_result *res);
121     int (*tb_insert)(struct fib_table *table,
122                         struct rtmsg *r, struct kern_rta *rta,
123                         struct nlmsghdr *n, struct netlink_skb_parms *req);
124     int (*tb_delete)(struct fib_table *table,
125                         struct rtmsg *r, struct kern_rta *rta,
126                         struct nlmsghdr *n, struct netlink_skb_parms *req);
127     int (*tb_dump)(struct fib_table *table,
128                         struct sk_buff *skb, struct netlink_callback *cb);
129     int (*tb_flush)(struct fib_table *table);
130     int (*tb_get_info)(struct fib_table *table,
131                         char *buf, int first, int count);
132     void (*tb_select_default)(struct fib_table *table,
133                         const struct rt_key *key,
134                         struct fib_result *res);
135     unsigned char  tb_data[0];
136     }
```
This structure contains pointers to table functions such as lookup, delete, insert etc.

\[tb_id:\] Table identifier; 255 for \textit{local\_table}, 254 for \textit{main\_table}

\[(*tb...)(\)\] Function pointers to the routines that perform the service indicated by the function name.

\[tb\_data[0]:\] Place holder for the associated FIB hash table (\textit{fn\_hash} structure defined in \texttt{net/ipv4/fib\_hash.c}. When the entire structure is dynamically allocated space for both the fixed size elements shown above and the hash table will be provided.
The *fn_hash* structure

The variable size area represented by the `tb_data[0]` placeholder is a hash type table area comprised of elements of type `struct fn_hash`. The *fn_hash* data structure contains pointers to *fn_zone* structures.

Each zone structure describes the routing data associated with a netmask having \( n \) leading 1 bits. Since netmasks in IPV4 are 32 bits in length the 33 zones correspond to netmasks having 0, 1, ..., 32 leading 1 bits. The all zero netmask matches any address and thus corresponds to the default routing entry.

```c
104 struct fn_hash
105 {
106    struct fn_zone  *fn_zones[33];
107    struct fn_zone  *fn_zone_list;
108  };
```

*fn_zones[33]*: Pointers to zone entries (one zone for each bit in the mask);

*fn_zones[0]* points to zone for netmask 0x00000000, *fn_zones[1]* points to zone for 0x80000000, ..., *fn_zone[32] points to zone for 0xFFFFFFFF.

*fn_zone_list*: Pointer to *most specific non-empty zone* in the list. Since the number of non-empty zones is typically small, the non-empty zones are themselves linked together to expedite lookup. This pointer serves as the base to the non-empty zone chain.
The \textit{fn_zone} structure

There is one \textit{fn_zone} structure for each non-empty prefix length that is present in the route table.

The \textit{fn_zone} structure contains hashing information and pointers to a hash table of pointers to \textit{fib_node} structures. There is a single \textit{fn_zone} structure for each prefix length \{0, 1, \ldots, 32\}. There will be multiple \textit{fib_nodes} associated with a single \textit{fn_zone} if and only if the routing table has multiple entries with the same number of leading 1 bits in the network mask.

There is a single zone structure for each \textit{active prefix length}. There will be multiple \textit{fib_nodes} in a zone if and only if only two destinations have the same prefix length (netmask). There will be multiple \textit{fib_nodes} on a single hash queue if and only if at least two destinations in the routing table hash to the same queue.

The size of the hash tables are variable and can be as small as one entry.

```c
85 struct fn_zone
86 {
87     struct fn_zone  *fz_next;  /* Next not empty zone */
88     struct fib_node **fz_hash; /* Hash table ptr */
89     int             fz_nent;   /* Number of entries */
90     int             fz_divisor; /* Hash divisor */
91     u32             fz_hashmask; /*(1<<fz_divisor) - 1 */
92     #define FZ_HASHMASK(fz) ((fz)->fz_hashmask)
93     int             fz_order;  /* Zone order */
94     u32             fz_mask;
95     #define FZ_MASK(fz) ((fz)->fz_mask)
96 }
```

- \textit{fz_next}: pointer to next most specific non-empty zone
- \textit{fz_hash}: pointer to hash table of nodes for this zone
- \textit{fz_divisor}: number of buckets in the hash table for this zone
- \textit{fz_hashmask}: number of buckets - 1
- \textit{fz_order}: number of leading 1 bits in the netmask (i.e. prefix length)
- \textit{fz_mask}: zone netmask, defined as \(-(1<<32-fz_order))-1\)
The fib_node structure

There is a fib_node structure for each host, network, or default route address represented in the routing table.

Since many routes may all be assigned to a single outgoing interface, a pointer to information relating to common features of the routes is maintained in fib_info structure. The fn_key element contains the destination IP address and is used as the hash table key.

Information on how to reach the next hop on the way to a particular destination resides in the associated fib_info structure. This partitioning is done because multiple remote hosts or networks may be reachable through a common gateway.

```c
68 struct fib_node
69 {
70     struct fib_node     *fn_next;
71     struct fib_info     *fn_info;
72 #define FIB_INFO(f)     ((f)->fn_info)
73     fn_key_t            fn_key;
74     u8                  fn_tos;
75     u8                  fn_type;
76     u8                  fn_scope;
77     u8                  fn_state;
78 }
```

```c
60 typedef struct { 
61         u32     datum;
62 } fn_key_t;
63```

- `fn_next`: pointer to next fib_node in this hash_queue
- `fn_info`: pointer to fib_info structure containing next hop data
- `fn_key`: Dest IP address associated with this routing table entry
- `fn_tos, etc:` Route attributes
The \texttt{fib_info} structure

There is a \texttt{fib_info} structure for each \textit{next hop gateway} that is defined in the routing table.

The \texttt{fib_info} structure defined in \texttt{include/net/ip_fib.h} contains protocol and hardware information that are specific to an interface.

\begin{verbatim}
57 struct fib_info
58 {
59    struct fib_info    *fib_next;
60    struct fib_info    *fib_prev;
61    int                 fib_treeref;
62    atomic_t            fib_clntref;
63    int                 fib_dead;
64    unsigned           fib_flags;
65    int                 fib_protocol;
66    u32                 fib_prefsrc;
67    u32                 fib_priority;
68    unsigned           fib_metrics[RTAX_MAX];
69    #define fib_mtu fib_metrics[RTAX_MTU-1]
70    #define fib_window fib_metrics[RTAX_WINDOW-1]
71    #define fib_rtt fib_metrics[RTAX_RTT-1]
72    #define fib_advmss fib_metrics[RTAX_ADVMSS-1]
73    int                 fib_nhs;
74    #ifdef CONFIG_IP_ROUTE_MULTIPATH
75        int             fib_power;
76    #endif
77    struct fib_nh       fib_nh[0];
78    #define fib_dev fib_nh[0].nh_dev
79    };
\end{verbatim}

\texttt{fib_protocol:} Identifies the source of the route. This must be a legitimate RTPROT value (definitions shown below)

\texttt{fib_nh[0]:} A place holder for a table of eligible device characteristic structures for devices used for sending or receiving packets for this route
The protocol field identifies the entity that created the route.

121 /* rtm_protocol */
122
123#define RTPROT_UNSPEC 0
124#define RTPROT_REDIRECT 1 /* Route installed by ICMP redirects;  
125             not used by current IPv4 */
126#define RTPROT_KERNEL 2 /* Route installed by kernel */
127#define RTPROT_BOOT 3 /* Route installed during boot */
128#define RTPROT_STATIC 4 /* Route installed by adminor */
129#define RTPROT_GATED 8 /* Apparently, GateD */
130#define RTPROT_RA 9 /* RDISC/ND router advertisments */
131#define RTPROT_MRT 10 /* Merit MRT */
132#define RTPROT_ZEBRA 11 /* Zebra */
The **fib_nh** structure

The **fib_nh** structure contains a pointer to the *net_device* or devices that represent the eligible outgoing interfaces along with data associated with the suitability of the route for traffic of various characteristics.

```
37 struct fib_nh
38 {
39     struct net_device *nh_dev;
40     unsigned nh_flags;
41     unsigned char nh_scope;
42 #ifdef CONFIG_IP_ROUTE_MULTIPATH
43     int nh_weight;
44     int nh_power;
45 #endif
46 #ifdef CONFIG_NET_CLS_ROUTE
47     __u32 nh_tclassid;
48 #endif
49     int nh_oif;
50     u32 nh_gw;
51 };
```

- **nh_dev:** Pointer to the local *net_device* structure associated with the interface.
- **nh_flags:** These flags (RTNH_F_DEAD, RTNH_F_PERVASIVE, RTNH_F_ONLINK) characterize the state of the route, and appear to be primarily related to managing multipath routes.
- **nh_scope:** The scope of this route (RT_SCOPE_HOST, RT_SCOPE_LINK, RT_SCOPE_UNIVERSE).
- **nh_weight:** Used for multipath routing in which traffic for a single destination is distributed across multiple outgoing links.
- **nh_power:** Details of how weight, power, and classid all work also need to become understood.
- **nh_tclassid:** Use for class-based routing in which traffic is partitioned according to class.
- **nh_oif:** Index of the interface (*nh->nh_oif = nhp->rtnh_ifindex*)
- **nh_gw:** IP address of the next hop gateway on this route.
Creation of a *fib_table* and the *fib_node* cache

The actual creation of the cache of the *fib_node* cache and the allocation and initialization of the *struct fib_table* is done in the function *fib_hash_init()*.

Note that allocation of *fn_zone* structures, their associated hash tables, and the *fib_node* structures *is done as routes are added*.

```c
struct fib_table * fib_hash_init(int id)
{
    struct fib_table *tb;

    if (fn_hash_kmem == NULL)
        fn_hash_kmem = kmalloc(sizeof(struct fib_table) +
                               sizeof(struct fn_hash), GFP_KERNEL);

    if (tb == NULL)
        return NULL;

    return NULL;
}
```

This test is necessary because multiple *fib_tables* will be created but we want only a single *fib_node* cache.
The functions used in table lookup and management operations are typically accessed indirectly through the `tb` pointer. Here are the actual bindings:

```c
908   tb->tb_id = id;
909   tb->tb_lookup = fn_hash_lookup;
910   tb->tb_insert = fn_hash_insert;
911   tb->tb_delete = fn_hash_delete;
912   tb->tb_flush = fn_hash_flush;
913   tb->tb_select_default = fn_hash_select_default;
914   tb->tb_dump = fn_hash_dump;
915 #ifdef CONFIG_PROC_FS
916   tb->tb_get_info = fn_hash_get_info;
917 #endif
918   memset(tb->tb_data, 0, sizeof(struct fn_hash));
919   return tb;
920 }
```

Routes are added and deleted either by the system administrator using the `route` command and by dynamic routing protocols.
IP Peer Initialization

The `inet_initpeers()` function resides in `net/ipv4/inetpeer.c` and is called by `ip_init()` to initialize a data structure used by the kernel to maintain long lived information about its peers. A peer is any remote system with which this system has exchanged data. This information consists only of the sequence number to be used in the IP header ID for the next outgoing packet to each destination and timestamp information for TCP connections.

An AVL tree data structure is used instead of a hash table for storing this information. It is said that this is done “to prevent easy and efficient DoS attacks by creating hash collisions.” Each node of the tree is of `struct inet_peer` type.

Each routing table cache element of type `struct rtable` contains a member which points to its corresponding `struct inet_peer` node in this tree.
The *struct inet_peer* is defined in *include/net/inetpeer.h*.

```c
17 18 struct inet_peer
19 {
20    struct inet_peer        *avl_left, *avl_right;
21    struct inet_peer        *unused_next, **unused_prevp;
22    unsigned long           dtime;  /* the time of last
23                                referenced entries */
24    atomic_t                refcnt;
25    __u32                   v4daddr; /* peer's address */
26    __u16                   avl_height;
27    __u16                   ip_id_count; /* IP ID for
28                                the next packet */
29    atomic_t                rid;  /* Frag reception count*/
30    __u32                   tcp_ts;
31    unsigned long          tcp_ts_stamp;
32 }
```

Functions of structure elements:

*ip_id_count:* ID for the next outgoing IP packet to this destination. This ID is carried in the IP header and is used in reassembly of fragmented packets. These values should be assigned on a *(source, dest)* pair basis rather than a per connection basis. Why?

*v4daddr:* Unsigned IP address of peer.

*dtime:* Time of last use.

*refcnt:* Number of entities holding a pointer to this structure.
Removal of tree nodes:

A node may be removed from the tree in the following three cases:

- When its reference counter reaches zero.
- It has not been used for a sufficiently long time.
- The node pool is overloaded and it happens to be the least recently used entry.

(The node pool is overloaded when the number of nodes in it is \( \geq \) inet_peer_threshold.)

The actual code for the initialization function is as follows:

```c
void __init inet_initpeers(void)
{
    struct sysinfo si;

    /* Use the interface to information about memory. */
    si_meminfo(&si);

    /* The values below were suggested by Alexey Kuznetsov
     * <kuznet@ms2.inr.ac.ru>. I don't have any opinion
     * about the values myself. --SAW
     */
    if (si.totalram <= (32768*1024)/PAGE_SIZE)
        inet_peer_threshold >>= 1; /* max pool size
                                     about 1MB on IA32 */
    if (si.totalram <= (16384*1024)/PAGE_SIZE)
        inet_peer_threshold >>= 1; /* about 512KB */
    if (si.totalram <= (8192*1024)/PAGE_SIZE)
        inet_peer_threshold >>= 2; /* about 128KB */

    peer_cachep = kmalloc(sizeof(union inet_peer),
                          0, SLAB_HWCACHE_ALIGN,
                          NULL, NULL);
```
if (!peer_cachep)
    panic("cannot create inet_peer_cache");

/* All the timers, started at system startup tend
to synchronize. Perturb it a bit. */
peer_periodic_timer.expires = jiffies
    + net_random() % inet_peer_gc_maxtime
    + inet_peer_gc_maxtime;
add_timer(&peer_periodic_timer);
Selecting the next IP packet identifier

1083  void __ip_select_ident(struct iphdr *iph, 
1084         struct dst_entry *dst, int more)
1085  {
1086       struct rtable *rt = (struct rtable *) dst;
1087       if (rt) {
1088             if (rt->peer == NULL)
1089                 rt_bind_peer(rt, 1);
1090       /* If peer is attached to destination, it is never detached,
1091          so that we need not to grab a lock to dereference it.
1092          */
1093       if (rt->peer) {
1094               iph->id = htons(inet_getid(rt->peer, more));
1095               return;
1096         }
1097       } else
1098           printk(KERN_DEBUG "rt_bind_peer(0) @%p\n",
1099                  __builtin_return_address(0));
1100       ip_select_fb_ident(iph);
1101  }
1103  
1104
static void ip_select_fb_ident(struct iphdr *iph)
{
    static DEFINE_SPINLOCK(ip_fb_id_lock);
    static u32 ip_fallback_id;
    u32 salt;

    spin_lock_bh(&ip_fb_id_lock);
    salt = secure_ip_id(ip_fallback_id ^ iph->daddr);
    iph->id = htons(salt & 0xFFFF);
    ip_fallback_id = salt;
    spin_unlock_bh(&ip_fb_id_lock);
}

/* The code below is shamelessly stolen from
   secure_tcp_sequence_number().
   All blames to Andrey V. Savochkin <saw@msu.ru>.
*/
__u32 secure_ip_id(__u32 daddr)
{
    struct keydata *keyptr;
    __u32 hash[4];

    keyptr = get_keyptr();

    /* Pick a unique starting offset for each IP dest.
       The dest ip address is placed in the start vector,
       which is then hashed with random data.
    */
    hash[0] = daddr;
    hash[1] = keyptr->secret[9];
    hash[2] = keyptr->secret[10];

    return half_md4_transform(hash, keyptr->secret);
Timers

*struct timer_list* is defined in `include/linux/timer.h`.

```c
struct timer_list {
    struct list_head list;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};
```

Functions of structure elements:

- `expires`: Desired expiration time of timer in jiffies.
- `function`: Function to be called with `data` as its argument, when timer expires. If the same function is managing several timers, the argument may be used to distinguish which one expired.

---

1 Man page for *add_timer, del_timer, init_timer* by Linus Torvalds available at http://man-pages.net/linux/man9/del_timer.9.html
peer_periodic_timer is of struct timer_list type and is declared as below. Since, it is statically initialised via the declaration, a call to init_timer is not necessary.

```c
100 static struct timer_list peer_periodic_timer =
101     { { NULL, NULL }, 0, 0, &peer_check_expire };
```

The function inet_initpeers() concludes by establishing the periodic timer used in the removal of inactive nodes.

```c
133       peer_periodic_timer.expires = jiffies
134           + net_random() % inet_peer_gc_maxtime
135           + inet_peer_gc_maxtime;
```

add_timer schedules an event, adding it to a linked list of events maintained by the kernel.

```c
136           add_timer(&peer_periodic_timer);
137 }
```
This section is technically not a part of initialization and should find a home elsewhere at some point in the future.

Removal of old entries from the inet_peer AVL tree.

The following variables are statically declared in net/ipv4/inetpeer.c. and used in the removal process. The total number of active nodes in the AVL tree is maintained in peer_total.

86 static volatile int peer_total;

TTL denotes time to live for a peer entry from time of its last use. inet_peer_minttl and inet_peer_maxttl hold the min and max values for TTL respectively. The system clock ticks 100 times per second in standard x86 kernels and thus HZ is normally set to 100.

/* TTL under high load: 120 sec */
90 int inet_peer_minttl = 120 * HZ;
/* usual time to live: 10 min */
91 int inet_peer_maxttl = 10 * 60 * HZ;

The peer_check_expire() function is called when peer_periodic_timer timer expires.

432 static void peer_check_expire(unsigned long dummy)
433 {
434     int i;
435     int ttl;

If the AVL tree is overloaded, TTL is set to its minimum. Otherwise, its value is scaled between inet_peer_minttl and inet_peer_maxttl based on the number of nodes in the tree.

437     if (peer_total >= inet_peer_threshold)
438         ttl = inet_peer_minttl;
439     else
440         ttl = inet_peer_maxttl
441             - (inet_peer_maxttl - inet_peer_minttl) / HZ * 
442             peer_total / inet_peer_threshold * HZ;
cleanup_once returns -1 if there is no unused node. Otherwise, it returns 0 after considering the node for removal -- what does this mean?? removing the node??

443 for (i = 0; i < PEER_MAX_CLEANUP_WORK && !cleanup_once(ttl); i++);

PEER_MAX_CLEANUP_WORK is defined as below,

97 #define PEER_MAX_CLEANUP_WORK 30

The following comment clearly describes how and when the next timeout event to occur is chosen.

445 /* Trigger the timer after inet_peer_gc_mintime .. inet_peer_gc_maxtime interval depending on the total number of entries (more entries, less interval). */

448 peer_periodic_timer.expires = jiffies + inet_peer_gc_maxtime - (inet_peer_gc_maxtime - inet_peer_gc_mintime) / HZ * peer_total / inet_peer_threshold * HZ;

The updated timer is added to the list of timers managed by the kernel.

452 add_timer(&peer_periodic_timer);
453 }
After IP Peer initialisation, \texttt{ip_init} creates a proc entry /proc/net/igmp, if IP multicasting is enabled.

```
1010  #ifdef CONFIG_IP_MULTICAST
1011     proc_net_create("igmp", 0, ip_mc_procinfo);
1012  #endif
1013 }
```

On 822 systems in our lab IP Multicasting is enabled and the following is the output from this proc reader.

```
[root@stephen net]# cat /proc/net/igmp
Idx Device : Count Querier Group    Users Timer Reporter
 1  lo     :  0      V2    010000E0    1 0:FB410A0D        0
 2  lec0   :  1      V2    010000E0    1 0:FB410A0D        0
 5  eth0   :  0      V2
```