IP Routing

The *ip_route_output_slow()* function, defined in net/ipv4/route.c is the major route resolver. Given a ''routing key'' as an input parameter, this routine builds a new route cache entry and stores a pointer to it in the parameter ***rp.* A Linux route is defined by (*dst*, *src*, *tos*).

```
int ip_route_output_slow(struct rtable **rp, const
                               struct rt_key *oldkey)
{
        struct rt_key key;
        struct fib_result res;
        unsigned flags = 0;
        struct rtable *rth;
        struct net_device *dev_out = NULL;
        unsigned hash;
        int free_res = 0;
        int err;
        u32 tos;
```

The function uses two important local variables: *key* is of *struct rt_key*, derived from the values pointed to by *oldkey* and is used to specify the characteristics of the desired route;

```
struct rt_key
{
        __u32      dst;       /* Destination IP address */
        __u32      src;       /* Source IP address      */
        int        iif;       /* Input interface index  */
        int        oif;       /* Output interface index */
#ifdef CONFIG_IP_ROUTE_FWMARK
        __u32      fwmark;
#endif
        __u8       tos;       /* Requested type of service */
        __u8       scope;     /* Host, LAN, site, universe */
};
```

The variable res has type struct fib_result and is later used in building the new routing cache entry.

```
struct fib_result
{
    unsigned char   prefixlen;
    unsigned char   nh_sel;
    unsigned char   type;
    unsigned char   scope;
    struct fib_info *fi;
#ifdef CONFIG_IP_MULTIPLE_TABLES
    struct fib_rule *r;
#endif
};
```

The elements of the fib_result structure include:

prefixlen	prefix length or equivalently the number of leading 1 bits in the subnet mask
nh_sel	next hop (output dev index). This actually appears under grep –r to be one of the ever popular write only variables!
scope	An indication of the distance to the destination IP address (e.g. host, local network, site, universe). Higher scope values are more specific.
type	type of address (LOCAL, UNICAST, BROADCAST, MULTICAST)
fi	Pointer to the fib_info structure that contains protocol and hardware information specific to an interface.
r	Pointer tof a fib_rule structure used for policy based routing.

The fib_rule structure is defined in net/ipv4/fib_rules.c. This structure is the key element defining the existence of a route with a given class of service between a specific source and destination address. It is not used unless CONFIG_IP_MULTIPLE_TABLES has been defined.

```
struct fib_rule
{
    struct fib_rule *r_next;
    atomic_t        r_clntref;
    u32             r_preference;
    unsigned char   r_table;
    unsigned char   r_action;
    unsigned char   r_dst_len;
    unsigned char   r_src_len;
    u32             r_src;
    u32             r_srcmask;
    u32             r_dst;
    u32             r_dstmask;
    u32             r_srcmap;
    u8              r_flags;
    u8              r_tos;
#ifdef CONFIG_IP_ROUTE_FWMARK
    u32             r_fwmark;
#endif
    int             r_ifindex;
#ifdef CONFIG_NET_CLS_ROUTE
    __u32           r_tclassid;
#endif
    char            r_ifname[IFNAMSIZ];
    int             r_dead;
};
```

The function ip_route_ouput_slow() begins by constructing the new routing key structure. Manipulation of the tos field is somewhat strange. TOS related constants are defined as follows:

```
#define IPTOS_TOS_MASK          0x1E
#define IPTOS_TOS(tos)          ((tos)&IPTOS_TOS_MASK)
#define IPTOS_LOWDELAY          0x10
#define IPTOS_THROUGHPUT        0x08
#define IPTOS_RELIABILITY       0x04
#define IPTOS_MINCOST           0x02

#define IPTOS_RT_MASK           (IPTOS_TOS_MASK & ~3)
#define RTO_ONLINK              0x01
```

RTO_ONLINK is a flag that indicates the destination is no more than one hop away and reachable via a link layer protocol. Thus, in line 1702 the input value of tos is being and'ed with 0x1d. Then in line it is 1705 is and'ed with 0x1c. Presumably there is a reason for distinguishing tos and key.tos.

```
1702     tos = oldkey->tos & (IPTOS_RT_MASK | RTO_ONLINK);
1703     key.dst = oldkey->dst;
1704     key.src = oldkey->src;
1705     key.tos = tos & IPTOS_RT_MASK;
```

The input interface identifier is forced to that of the loopback device. The variable loopback_dev is an instance of struct net_device and is globally defined in drivers/net/Space.c. The value of the ifindex field is a unique identifier assigned to the interface at initialization time. The output interface identifier is copied from the oldkey structure.

```
    key.iif = loopback_dev.ifindex;
    key.oif = oldkey->oif;
```

CONFIG_IP_ROUTE_FWMARK is an option to specify different route for packets with different (netfilter) mark values.

```
#ifdef CONFIG_IP_ROUTE_FWMARK
    key.fwmark = oldkey->fwmark;
#endif
```

The value of key.scope is an indication of the distance from the destination. Here there are only two possible choices, and they depend on the setting of RTO_ONLINK If RTO_ONLINK is set then the scope must be RT_SCOPE_LINK. Otherwise it is RT_SCOPE_UNIVERSE. Thus the scope attribute of the new key does reflect the setting of the RTO_ONLINK bit in the tos field of the old key.

```
    key.scope = (tos & RTO_ONLINK) ? RT_SCOPE_LINK:
                   RT_SCOPE_UNIVERSE;
```

As described more fully in the kernel comments below and subsequent data definitions it is clear that a wider range of possible scopes is intended and that the higher the value of scope the more specific the target routing domain.

```
/* rtm_scope

Really not a scope, but sort of distance to the destination.
NOWHERE are reserved for non-existing  dests, HOST is our
local addresses, LINK are dests on directly attached
link and UNIVERSE is everywhere in the Universe.

Intermediate values are also possible f.e.  interior routes
could be assigned a value between UNIVERSE and LINK.
*/
```

RT_SCOPE_LINK, RT_SCOPE_UNIVERSE stand for on−link routes and global routes respectively and are defined in include/linux/rtnetlink.h.

```
enum rt_scope_t
{
    RT_SCOPE_UNIVERSE=0,
/* User defined values  */
    RT_SCOPE_SITE=200,
    RT_SCOPE_LINK=253,
    RT_SCOPE_HOST=254,
    RT_SCOPE_NOWHERE=255
};
```

CONFIG_IP_MULTIPLE_TABLES is an option that allows the Linux router to be able to take the packet's source address into account. (Normally, a router decides what to do with a received packet based solely on the packet's final destination address.)

The routing tables are referred to as "classes". Currently, the number of classes is limited to 255, of which three classes are builtin[1]:

RT_CLASS_LOCAL	= 255 – local interface addresses, broadcasts, nat addresses
RT_CLASS_MAIN	= 254 – all normal routes are put here by default.
RT_CLASS_DEFAULT	= 253 – If the ip_fib_model == 1, then normal default routes are put there. If the ip_fib_model == 2, all gateway routes are put there .

```
        res.fi             = NULL;
```

```
#ifdef CONFIG_IP_MULTIPLE_TABLES
        res.r              = NULL;
#endif
```

Check if the source address is defined, and if so determine its type. If the source address is a MULTICAST, BADCLASS and ZERONET address (these macros are defined in include/linux/in.h), return error.

```
#define LOOPBACK(x)
        (((x) & htonl(0xff000000)) == htonl(0x7f000000))
#define MULTICAST(x)
        (((x) & htonl(0xf0000000)) == htonl(0xe0000000))
#define BADCLASS(x)
        (((x) & htonl(0xf0000000)) == htonl(0xf0000000))
#define ZERONET(x)
        (((x) & htonl(0xff000000)) == htonl(0x00000000))
#define LOCAL_MCAST(x)
        (((x) & htonl(0xFFFFFF00)) == htonl(0xE0000000))
```

```
        if (oldkey->src) {
             err = -EINVAL;
             if (MULTICAST(oldkey->src) ||
                 BADCLASS(oldkey->src)  ||
                  ZERONET(oldkey->src))
                  goto out;
```

1 http://lxr.linux.no/source/Documentation/networking/policy-routing.txt

The ip_dev_find() function looks up the IP source address in the local table and returns a pointer to the struct net_device associated with the source address. This function is defined in net/ipv4/fib_frontend.c.

```
/* It is equivalent to inet_addr_type(saddr) ==  RTN_LOCAL */
          dev_out = ip_dev_find(oldkey->src);
```

The input parameter here is the source IP address associated with the route being setup. Note that the address is subsequently put into the dst element of the new key structure that is built.

```
struct net_device * ip_dev_find(u32 addr)
{
    struct rt_key key;
    struct fib_result res;
    struct net_device *dev = NULL;

    memset(&key, 0, sizeof(key));
    key.dst = addr;
#ifdef CONFIG_IP_MULTIPLE_TABLES
    res.r = NULL;
#endif

```

The variable local_table is a reference to the statically defined local table.

```
ip_fib.h:
#define local_table (fib_tables[RT_TABLE_LOCAL])
```

```
    if (!local_table ||
         local_table->tb_lookup(local_table, &key, &res)) {
         return NULL;
}
```

The call to local_table−>tb_lookup() is a reference to the fn_hash_lookup() function. This function is used to determine if the destination entity identified by key exists in the specified table. All the fib_tables are searched by zone where a routing zone is the set of routing destinations that have the same length prefix (or equivalently netmask). The fn_hash_lookup() searches the specified table, starting with the most specific zone netmask looking for a match. The most specific existing zone is pointed by the fn_zone_list variable.

```
static int
fn_hash_lookup (struct fib_table *tb,
        const struct rt_key *key, struct fib_result *res)
{
        int err;
        struct fn_zone *fz;
        struct fn_hash *t = (struct fn_hash*)tb->tb_data;

```

This outer loop processes every non−empty zone associated with the fib_table in longest prefix first order.

```
        read_lock(&fib_hash_lock);
        for (fz = t->fn_zone_list; fz; fz = fz->fz_next) {
             struct fib_node *f;
             fn_key_t k = fz_key(key->dst, fz);
```

The fz_key() function, defined in fib_hash.c, builds a test key by and−ing the address with the zone's netmask. The structures fn_key_t and fn_hash_idx_t are simply unsigned integers representing IP prefixes and hash table indices respectively.

```
typedef struct {
        u32 datum;
} fn_key_t;

typedef struct {
        u32 datum;
} fn_hash_idx_t;

static __inline__ fn_key_t fz_key(u32 dst, struct
                            fn_zone *fz)
{
        fn_key_t k;
        k.datum = dst & FZ_MASK(fz);
        return k;
}
```

FZ_MASK is a macro defined in fib_hash.c

```
#define FZ_MASK(fz)      ((fz)->fz_mask)
```

On returning to fn_hash_lookup(), this inner loop traverses the list of fib_node structures associated with the hash bucket of the routing key searching for the first key match. To initiate this process fz_chain() is called to retrieve the address of the first fib_node in the chain. It performs the hash function fn_hash() and ANDs this value with the zone's fz_hashmask to get an index into the zone's hash table of nodes. The syntax of this function is a bit dense. Note that fn_hash returns fn_hash_idx_t which was shown above to be a "structure" consisting of a single unsigned int member called datum. That value is used as an index into the hash table structure associated with the routing zone yielding the required pointer to the struct fib_node.

```
             for (f = fz_chain(k, fz); f; f = f->fn_next) {
```

```
static __inline__ struct fib_node * fz_chain(fn_key_t
                              key, struct fn_zone *fz)
{
     return fz->fz_hash[fn_hash(key, fz).datum];
}
static __inline__ fn_hash_idx_t fn_hash(fn_key_t key,
             struct fn_zone *fz)
{
     u32 h = ntohl(key.datum)>>(32 - fz->fz_order);
     h ^= (h>>20);
     h ^= (h>>10);
     h ^= (h>>5);
     h &= FZ_HASHMASK(fz);
```

FZ_HASHMASK is a macro defined in fib_hash.c

```
#define FZ_HASHMASK(fz) ((fz)->fz_hashmask)
```

fn_hash_idx_t is a structure containing the address as its element.

```
typedef struct {
     u32     datum;
} fn_hash_idx_t;
```

```
     return *(fn_hash_idx_t*)&h;
}
```

The first action of the inner loop is to compare search key with the key of the struct fib_node. Recall that the variable k is an instance of fn_key_t, a structure of the single element datum, whose value was previously set to the target IP address anded with the netmask associated with the zone. From this we can infer that the value of f->fn_key is the network address or CIDR network prefix associated with the routing table entity associated with this node. The nodes on any hash queue are sorted in decreasing order by prefix. Therefore, if they do not match and if the search key value is greater than that of the node key, the search continues on to the next node.

```
                if (!fn_key_eq(k, f->fn_key))
                {
                     if (fn_key_leq(k, f->fn_key))
                           break;
                     else
                          continue;
                }
```

Arriving here implies that there has been a match. CONFIG_IP_ROUTE_TOS makes use of TOS value as routing key and so if there is a tos associated with the fib_node and it is not equal to the tos of the key, the match is discarded and the search continues.

```
#ifdef CONFIG_IP_ROUTE_TOS
                if (f->fn_tos && f->fn_tos != key->tos)
                     continue;
#endif
```

Update and test the state information of the fib_node. Zombie nodes are considered non–usable and likely relate to deleted routes or dead interfaces. Very little state information is present in fib_notes. Only 2 bits are defined:

```
80 #define FN_S_ZOMBIE      1
81 #define FN_S_ACCESSED    2
```

```
                f->fn_state |= FN_S_ACCESSED;

                if (f->fn_state & FN_S_ZOMBIE)
                     continue;
```

Recall that higher values of scope means more specific or constrained routing. Thus the node scope is required to be at least as specific as the requested route scope. If the fib_node scope is less than that of the scope of the key, then this node is also not usable.

```
                if (f->fn_scope < key->scope)
                     continue;

```

Finally the fib_semantic_match() function is called to ensure that this fib_node is usable within the semantic constraints imposed by the route key.

```
                 err = fib_semantic_match(f->fn_type,
                           FIB_INFO(f), key, res);
```

The fib_semantic_match() function is defined in net/ipv4/fib_semantics.c. Its mission is to ensure that the candidate fib_node appears to represent an acceptable route. The tests include ensuring that the associated fib_info's view of the next hop is that it is alive, the fib_nh's view of the next hop is that its alive, and that if the output interface is specified in the routing key, it is the same interface as the one associated with the next hop structure.

```
int
fib_semantic_match (int type, struct fib_info *fi, const
                 struct rt_key *key, struct fib_result *res)
{
        int err = fib_props[type].error;

        if (err == 0) {
```

If the fib_info structure indicates that the next hop is dead, then failure is returned.

```
            if (fi->fib_flags & RTNH_F_DEAD)
                 return 1;

```

The fib_info structure is connected to the results structure.

```
            res->fi = fi;

            switch (type) {
#ifdef CONFIG_IP_ROUTE_NAT
            case RTN_NAT:
                 FIB_RES_RESET(*res);
                 atomic_inc(&fi->fib_clntref);
                 return 0;
#endif
```

Only the NAT type route is distinguished for the purposes of route semantics.

```
            case RTN_UNICAST:
            case RTN_LOCAL:
            case RTN_BROADCAST:
            case RTN_ANYCAST:
            case RTN_MULTICAST:
```

Check if a next hop is feasible from this node. The macros used in this loop depend upon whether or not multipath routing is enabled. If not, there can be only one next hop associated with a fib_info structure.

```
#ifdef CONFIG_IP_ROUTE_MULTIPATH

#define for_nexthops(fi) { int nhsel; const struct fib_nh * nh; \
for (nhsel=0, nh = (fi)->fib_nh; nhsel < (fi)->fib_nhs; nh++, nhsel++)

#else /* CONFIG_IP_ROUTE_MULTIPATH */

/* Hope, that gcc will optimize it to get rid of dummy loop */

#define for_nexthops(fi) {int nhsel=0;const struct fib_nh *nh = (fi)->fib_nh; \
for (nhsel=0; nhsel < 1; nhsel++)

#endif /* CONFIG_IP_RO
```

```
                for_nexthops(fi) {
                        if  (nh->nh_flags & RTNH_F_DEAD)
                                continue;
```

If the route key requires a specific output interface and that is not the output interface associated with this fib_nh then the route is not usable. The break is taken if the route is usable.

```
                        if (!key->oif || key->oif == nh->nh_oif)
                                break;
                }
```

The CONFIG_IP_ROUTE_MULTIPATH option allows the routing tables to specify alternative paths to travel for a given packet. The router considers all these paths to be of equal "cost" and chooses one of them in a non–deterministic fashion when selecting a route. How is this done??

```
#ifdef CONFIG_IP_ROUTE_MULTIPATH
                if (nhsel < fi->fib_nhs) {
                        res->nh_sel = nhsel;
                        atomic_inc(&fi->fib_clntref);
                              return 0;
                }
#else
```

For non multi–path routing, this is the success return point. The loop will have been exited via the break and so nhsel will remain 0. The reference counter of the fib_info structure is incremented here.

```
                if (nhsel < 1) {
                atomic_inc(&fi->fib_clntref);
                        return 0;
                }
#endif
```

This endfor is misleading. The actual loop ended at line 597. This closes the block in which the local variables preceeding the for loop are declared.

```
610            endfor_nexthops(fi);
```

Falling out of the loop implies no fib_nh with acceptable semantics was found.

```
611            res->fi = NULL;
612                 return 1;
613            default:
614                 res->fi = NULL;
615            printk(KERN_DEBUG "impossible 102\n");
616                 return -EINVAL;
617           }
618       }
619       return err;
620 }
```

This is the point of return from fib_semantic_match() to fn_hash_lookup(). If the the source address was found to be acceptable, the res structure was filled with the type and scope elements copied from the fib_node structure and the prefix length is copied from the fn_zone structure.

```
299                 if (err == 0) {
300                      res->type = f->fn_type;
301                      res->scope = f->fn_scope;
302                      res->prefixlen = fz->fz_order;
303                      goto out;
304                 }
305                 if (err < 0)
306                      goto out;
307            }
308       }
309       err = 1;
310 out:
311       read_unlock(&fib_hash_lock);
312       return err;
313 }
314
```

Here control returns to ip_find_dev(), since the source address is being processed, it is necessary that the returned route type be RTN_LOCAL. This seems like one convoluted way to find if a host owns a particular IP address. If the route is not RTN_LOCAL, a jump is made to the tag out bypassing the code which normally sets up the return value, dev. The value of dev was initialized to NULL, and a return value of NULL will cause ip_route_output_slow to return failure.

```
    if (res.type != RTN_LOCAL)
        goto out;
```

FIB_RES_DEV, a macro defined in include/net/ip_fib.h, extracts the struct netdevice pointer from the fib_info pointer contained in the results structure. Note that dev−>refcnt is incremented here. Where the corresponding decrement occurs is not clear at present.

```
#define FIB_RES_DEV(res) (FIB_RES_NH(res).nh_dev)
#define FIB_RES_NH(res)  ((res).fi->fib_nh[0])
```

```
    dev = FIB_RES_DEV(res);
    if (dev)
        atomic_inc(&dev->refcnt);

```

The fib_res_put() function triggers a set of events that is not well understood at present.

```
out:
    fib_res_put(&res);
    return dev;
}
```

What is not well understood here is how routes dynamically become ‘‘dead’’ or come to have reference counts of 0. The best guess at the moment is that the fib_info structure is held by all but its creator for a very short interval of time. Nevertheless, it would be possible that whatever owned and normally keeps the reference count at 1 tried to delete the route while we owned it here. Thus when we release it, it really should go away, but qui sait.

```
static inline void fib_res_put(struct fib_result *res)
{
    if (res->fi)
        fib_info_put(res->fi);
#ifdef CONFIG_IP_MULTIPLE_TABLES
    if (res->r)
        fib_rule_put(res->r);
#endif
}
```

The fib_clntref is a reference counter and when its value reaches zero, the struct fib_info is deleted. In this context fib_clntref was incremented in the function fib_semantic_match(). The atomic_dec_and_test() function returns true if the value is zero.

```
static inline void fib_info_put(struct fib_info *fi)
{
    if (atomic_dec_and_test(&fi->fib_clntref))
         free_fib_info(fi);
}

void free_fib_info(struct fib_info *fi)
{
    if (fi->fib_dead == 0) {
         printk("Freeing alive fib_info %p\n", fi);
         return;
    }
```

Unless multipath routing is enabled, change_nexthops() will cause the enclosed block to be executed exactly one time and this fib_info structure's claim on the net_device will be dropped.

```
    change_nexthops(fi) {
         if (nh->nh_dev)
              dev_put(nh->nh_dev);
         nh->nh_dev = NULL;
    } endfor_nexthops(fi);
    fib_info_cnt--;
    kfree(fi);
}
```

Release a fib_rule structure.

```
void fib_rule_put(struct fib_rule *r)
{
    if (atomic_dec_and_test(&r->r_clntref)) {
         if (r->r_dead)
              kfree(r);
    else
         printk("Freeing alive rule %p\n", r);
    }
}
```

Finally return is made from ip_find_dev() to ip_route_output_slow(). Recall that we only embarked upon this path if the source IP address was not NULL. If the value of dev_out is NULL, then there is no usable network interface associated with the source IP address. The comment below discusses why it is not necessary that the device found here actually map to the output interface specified by the caller. He actually probably means key.oif == dev_out->oif.

```
              if (dev_out == NULL)
                   goto out;

 /*   I removed check for oif == dev_out->oif here.
      It was wrong by three reasons:
      1. ip_dev_find(saddr) can return wrong iface, if saddr
           is assigned to multiple interfaces.
      2. Moreover, we are allowed to send packets with saddr
           of another iface. --ANK
 */
```

Since oif == 0 means unspecified, what is happening here is a coerced conversion of a multicast and broadcast destination addresses to use the output interface associated with the device that was returned. In addition to the factors discussion below, it is also the case that proper multicast addresses must be associated with a specific interface.

```
              if (oldkey->oif == 0
                   && (MULTICAST(oldkey->dst) ||
                   oldkey->dst == 0xFFFFFFFF)) {
         /*   Special hack: user can direct multicasts
              and limited broadcast via necessary interface
              without fiddling with IP_MULTICAST_IF or IP_PKTINFO.
              This hack is not just for fun, it allows
              vic,vat and friends to work.
              They bind socket to loopback, set ttl to zero
              and expect that it will work.
              From the viewpoint of routing cache they are broken,
              because we are not allowed to build multicast path
              with loopback source addr (look, routing cache
              cannot know, that ttl is zero, so that packet
              will not leave this host and route is valid).
              Luckily, this hack is good workaround.
         */

                   key.oif = dev_out->ifindex;
                   goto make_route;
              }
```

Release the device by invoking the dev_put() function defined in include/linux/netdevice.h

```
              if (dev_out)
                   dev_put(dev_out);
              dev_out = NULL;
         } /* end if (oldkey->src) */
```

If an output interface index is specified, attempt to retrieve a pointer to the associated struct net_device. A return value of NULL indicates the device is not found. If the device exists, its reference count is incremented, and the pointer is safe until dev_put is called to release it.

```
    if (oldkey->oif) {
         dev_out = dev_get_by_index(oldkey->oif);
         err = -ENODEV;
         if (dev_out == NULL)
              goto out;
```

The IPV4 specific data is retrieved by the in_dev_get() function which is defined in include/linux/inetdevice.h. This call returns the void *ip_ptr element of the net_device structure. This pointer points to an instance of struct in_device. Each net_device that supports IPV4 also has an associated struct in_device that carries the IPV4 dependencies of the device layer. An important element of the in_device is the ifa_list pointer. This pointer is the root of a list of struct ifa_list elements.

```
         if (__in_dev_get(dev_out) == NULL) {
              dev_put(dev_out);
              goto out;       /* Wrong error code */
         }
```

```
__in_dev_get(const struct net_device *dev)
{
    return (struct in_device*)dev->ip_ptr;
}

```

```
struct in_device
{
    struct net_device   *dev;
    atomic_t            refcnt;
    rwlock_t            lock;
    int                 dead;
    struct in_ifaddr   *ifa_list;  /* IP ifaddr chain   */
    struct ip_mc_list  *mc_list;   /* IP mcst filter chain */
    unsigned  long      mr_v1_seen;
    struct neigh_parms *arp_parms;
    struct ipv4_devconf cnf;
};
```

Each physical net_device may be assigned alias IP addresses and labels (eth0:1 eth0:2, .. etc). Each alias is represented by an instance of the struct in_ifaddr . The distinction between ifa_local and ifa_address is not well understood. Empirical analysis of ''normal'' network configurations fails to disclose any instances in which ifa_local and ifa_address differ.

```
struct in_ifaddr
{
    struct in_ifaddr    *ifa_next;
    struct in_device    *ifa_dev;
    u32                 ifa_local;
    u32                 ifa_address;
    u32                 ifa_mask;
    u32                 ifa_broadcast;
    u32                 ifa_anycast;
    unsigned char       ifa_scope;
    unsigned char       ifa_flags;
    unsigned char       ifa_prefixlen;
    char                ifa_label[IFNAMSIZ];
};
```

When a new interface is created by the inet_rtm_newaddr(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg) function in net/ipv4/devinet, the two addresses are set to the values passed in via the netlinks protocol message (don't ask).

```
    if (rta[IFA_ADDRESS-1] == NULL)
         rta[IFA_ADDRESS-1] = rta[IFA_LOCAL-1];
    memcpy(&ifa->ifa_local, RTA_DATA(rta[IFA_LOCAL-1]), 4);
    memcpy(&ifa->ifa_address,RTA_DATA(rta[IFA_ADDRESS-1]),4);
    ifa->ifa_prefixlen = ifm->ifa_prefixlen;
```

Continuing along in the code block in which the oif index was explicitly specified, if the destination address is a LOCAL multicast address or broadcast address, retreive the IP address of the output device. Recall that dev_out is a pointer to the struct net_device associated with the explicitly specified output interface. The call to inet_select_address() will return the ifa_local associated with the first interface that is found associated with the net_device that has scope no more restrictive (numerically less than or equal to) than LINK. The use of RT_SCOPE_LINK seems a bit unusual here. It will turn out that the scope is used only for LOCAL MCAST and BCAST. For UCAST destinations the scope will be set to RT_SCOPE_HOST when inet_select_address() is called.

```
        if (LOCAL_MCAST(oldkey->dst)||oldkey->dst ==
                  0xFFFFFFFF) {
             if (!key.src)
                  key.src = inet_select_addr(dev_out, 0,
                            RT_SCOPE_LINK);
             goto make_route;
        }
```

```
/* Some random defines to make it easier in the kernel.. */
#define LOOPBACK(x)     (((x) & htonl(0xff000000)) == htonl(0x7f000000))
#define MULTICAST(x)    (((x) & htonl(0xf0000000)) == htonl(0xe0000000))
#define BADCLASS(x)     (((x) & htonl(0xf0000000)) == htonl(0xf0000000))
#define ZERONET(x)      (((x) & htonl(0xff000000)) == htonl(0x00000000))
#define LOCAL_MCAST(x)  (((x) & htonl(0xFFFFFF00)) == htonl(0xE0000000))
```

When the destination address is LOCAL multcast or broadcast, the inet_select_addr() function, defined in net/ipv4/devinet.c, returns the local address associated with the specified output device. In this case dev points to the output device, the dst address is NULL and the scope is RT_SCOPE_LINK. The return value is the selected IP address or is NULL upon failure.

```
u32 inet_select_addr(const struct net_device *dev, u32
                     dst, int scope)
{
     u32 addr = 0;
     struct in_device *in_dev;

     read_lock(&inetdev_lock);
     in_dev = __in_dev_get(dev);
     if (in_dev == NULL) {
          read_unlock(&inetdev_lock);
          return 0;
     }

```

At this point in_dev points to a valid in_device structure. The for_primary_ifa macro runs the interface address chain associated with the in_device. Recall that routing scope values are ordered with the most specific scope (i.e. this host) having the highest value. The scope passed in was RT_SCOPE_LINK. Thus interfaces having a more specific address scope (HOST or NOWHERE) are rejected (for reasons, yet unknown). The address matching logic is with respect to the network mask associated with the in_ifaddr structure. In practice it would appear that only 2 distinct values of scope are assigned to interfaces. Scope 0 (UNIVERSE) is assigned to physical interfaces and scope 254 (HOST) to the loopback interface, lo. Thus in the scope matching logic below, physical interfaces are always acceptable and the loopback interface is acceptable only if the input scope is also HOST.

```
        read_lock(&in_dev->lock);
        for_primary_ifa(in_dev) {
             if (ifa->ifa_scope > scope)
                  continue;
```

The value of dst that was passed in was 0. Therefore !dst is true and the value of addr is set to the ifa_local field of the interface. Note that the address matching test is against ifa_address, but if a match occurs addr is set to ifa_local.

```
static __inline__ int inet_ifa_match(u32 addr, struct
                       in_ifaddr *ifa)
{
        return !((addr^ifa->ifa_address)&ifa->ifa_mask);
}
```

```
             if (!dst || inet_ifa_match(dst, ifa)) {
                  addr = ifa->ifa_local;
                  break;
             }
             if (!addr)
                  addr = ifa->ifa_local;
        } endfor_ifa(in_dev);
        read_unlock(&in_dev->lock);
        read_unlock(&inetdev_lock);

```

For the control path we are investingating it appears that addr should always be non–zero here and thus a return should take place.

```
        if (addr)
             return addr;
```

If control should reach here, it indicates that dst was non−zero and didn't match the ifa_address field of any interface address structure associated with the device. dev_base is a global variable pointing to the list of all instances of struct net_device. Here the selection criterion appears to be finding an interface whose scope is not LINK and whose scope is numerically less than or equal to the scope that was passed in.

```

/*  Not loopback addresses on loopback should  be preferred
    in this case. It is importnat that lo is the 1st intf
    in dev_base list.
 */
    read_lock(&dev_base_lock);
    read_lock(&inetdev_lock);
    for (dev = dev_base; dev; dev = dev->next) {
         if ((in_dev=__in_dev_get(dev)) == NULL)
              continue;

         read_lock(&in_dev->lock);
         for_primary_ifa(in_dev) {
              if (ifa->ifa_scope != RT_SCOPE_LINK &&
                   ifa->ifa_scope <= scope) {
                   read_unlock(&in_dev->lock);
                   read_unlock(&inetdev_lock);
                   read_unlock(&dev_base_lock);
                   return ifa->ifa_local;
              }
         } endfor_ifa(in_dev);
         read_unlock(&in_dev->lock);
    }
    read_unlock(&inetdev_lock);
    read_unlock(&dev_base_lock);

```

Return failure if an acceptable address cannot be found.

```
    return 0;
}

```

Well wasn't that an interesting excursion! Recall that this code block was executed only if the routing key specified an output interface and that the objective was to find an IP source address that is in some sense compatible with previously selected output device. We just dispensed with local multicast and broadcast source addresses. If the destination is general MULTICAST, then the address is selected from the output device using the key's scope. If the destination is unspecified, the scope RT_SCOPE_HOST is passed in.

```
               if (!key.src) {
                    if (MULTICAST(oldkey->dst))
                         key.src = inet_select_addr(dev_out, 0,
                                        key.scope);
               else if (!oldkey->dst)
                         key.src = inet_select_addr(dev_out, 0,
                                   RT_SCOPE_HOST);
               }
          } /* if (oldkey->oif) */

```

If the destination address is unspecified, the destination is set to the source address (which is presumably on this machine) . If the source is also NULL then they are both set to the loopback address.

```
          if (!key.dst) {
               key.dst = key.src;

               if (!key.dst)
                    key.dst= key.src = htonl(INADDR_LOOPBACK);
               if (dev_out)
                    dev_put(dev_out);
```

Use loopback device for sending packet to this machine.

```
               dev_out = &loopback_dev;
               dev_hold(dev_out);
               key.oif = loopback_dev.ifindex;
               res.type = RTN_LOCAL;
               flags |= RTCF_LOCAL;
               goto make_route;
          }
```

Finally, the function fib_lookup() defined in include/net/ip_fib.h is invoked to try to resolve the destination address.

```
    if (fib_lookup(&key, &res)) {
         res.fi = NULL;
```

Since the destination may be on this host as well as elsewhere in the Internet, the fib_lookup() function calls tb_lookup() on both the local table and the main table. Both tb_lookup functions resolve to fn_hash_lookup which was encountered earlier. Since fn_hash_lookup() returns 0 on success and non−zero on failure. The operation fails only if both lookups fail. Theoretically, at least, the lookup should not succeed in both tables but if it does, it would appear that the main table has precedence.

```
static inline int fib_lookup(const struct rt_key *key,
             struct fib_result *res)
{
    if (local_table->tb_lookup(local_table, key, res) &&
         main_table->tb_lookup(main_table, key, res))
              return -ENETUNREACH;
        return 0;
}
```

Falling into this implies that the fib_lookup failed. Check to see if an output interface was specified (haven't we been here before??) and, if so, get the source address from the device.

```
         if (oldkey->oif) {

/*  Apparently, routing tables are wrong. Assume,
    that the destination is on link.

    WHY? DW.
    Because we are allowed to send to iface
    even if it has NO routes and NO assigned
    addresses. When oif is specified, routing
    tables are looked up with only one purpose:
    to catch if destination is gatewayed, rather than
    direct. Moreover, if MSG_DONTROUTE is set,
    we send packet, ignoring both routing tables
    and ifaddr state. --ANK

    We could make it even if oif is unknown,
    likely IPv6, but we do not.
*/

              if (key.src == 0)
                   key.src = inet_select_addr(dev_out, 0,
                                  RT_SCOPE_LINK);
              res.type = RTN_UNICAST;
              goto make_route;
         }
```

```
        if (dev_out)
            dev_put(dev_out);
        err = -ENETUNREACH;
        goto out;
    }
    free_res = 1;
```

probably some error occurred during lookup ??

```
    if (res.type == RTN_NAT)
        goto e_inval;
```

This packet is routed locally (RTN_LOCAL), so destination and source are the same.

```
    if (res.type == RTN_LOCAL) {
        if (!key.src)
            key.src = key.dst;
        if (dev_out)
            dev_put(dev_out);
        dev_out = &loopback_dev;
        dev_hold(dev_out);
        key.oif = dev_out->ifindex;
```

Release reference into FIB table by calling fib_info_put().

```
        if (res.fi)
            fib_info_put(res.fi);
        res.fi = NULL;
        flags |= RTCF_LOCAL;
        goto make_route;
    }

#ifdef CONFIG_IP_ROUTE_MULTIPATH
    if (res.fi->fib_nhs > 1 && key.oif == 0)
        fib_select_multipath(&key, &res);
    else
#endif
```

If the prefix length is 0 (implying default route), and the type is UNICAST, and no output interface index was specified then its necessary to select among (possibly multiple) default routes.

```
    if(!res.prefixlen && res.type == RTN_UNICAST && !key.oif)
        fib_select_default(&key, &res);
```

The fib_select_default() function is defined in include/net/ip_fib.h. It appears that it may be a no operation if the if conditions are false. The FIB_RES_GW() macro will return the nh_gw of the next hop structure. As noted earlier three different entities, the node, the next hop, and the interface all have scope values. The value of nh_scope appears to be the most specific, having the value 254 for all local interface entries and local net entries in the main table. It does appear to have a 253 value for those table entries that do specify routing through a gateway either to a remote net or the default route.

```
static inline void fib_select_default(const struct rt_key
        *key, struct fib_result *res)
{
    if (FIB_RES_GW(*res) &&
            FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
        main_table->tb_select_default(main_table, key, res);
}
```

This function calls the main table's tb_select_default() which is a reference to the function fn_hash_select_default() defined in net/ipv4/fib_hash.c.

```
static void
fn_hash_select_default(struct fib_table *tb,
        const struct rt_key *key, struct fib_result *res)
{
    int order, last_idx;
    struct fib_node *f;
    struct fib_info *fi = NULL;
    struct fib_info *last_resort;
    struct fn_hash *t = (struct fn_hash*)tb->tb_data;
    struct fn_zone *fz = t->fn_zones[0];
```

fz points to the default netmask (fn_zones[0]). If that zone list is empty, there are no default routes and there is no more that can be done.

```

    if (fz == NULL)
        return;

    last_idx = -1;
    last_resort = NULL;
    order = -1;

    read_lock(&fib_hash_lock);
```

Iterate through all the nodes for the order zero zone. Needless to say this implies the existence of more than one default route. To successfully find something here would require finding a nh_scope of RT_SCOPE_LINK which we have not seen in our examples.

```
    for (f = fz->fz_hash[0]; f; f = f->fn_next) {
         struct fib_info *next_fi = FIB_INFO(f);

         if ((f->fn_state & FN_S_ZOMBIE) ||
              f->fn_scope != res->scope ||
              f->fn_type != RTN_UNICAST)
              continue;

         if (next_fi->fib_priority > res->fi->fib_priority)
              break;
         if (!next_fi->fib_nh[0].nh_gw ||
              next_fi->fib_nh[0].nh_scope!= RT_SCOPE_LINK)
              continue;
         f->fn_state |= FN_S_ACCESSED;

         if (fi == NULL) {
              if (next_fi != res->fi)
                   break;
         } else if (!fib_detect_death
              (fi,order,&last_resort, &last_idx)) {
```

fib_detect_death() checks whether the route (checking all nexthops) contains alive paths and whether the route can be used as last resort if there are no valid alternative routes in the group.

```
static int fib_detect_death(struct fib_info *fi, int order,
         struct fib_info **last_resort, int *last_idx)
{
    struct neighbour *n;
    int state = NUD_NONE;

    n = neigh_lookup(&arp_tbl,&fi->fib_nh[0].nh_gw,
              fi->fib_dev);
```

neigh_lookup() function is defined in net/core/neighbour.c. This function gets the physical address of the neighbour by matching the IP address of the search key with the one in the table.

```
struct neighbour *neigh_lookup (struct neigh_table
                   *tbl, const void *pkey,
                              struct net_device *dev)
{
    struct neighbour *n;
    u32 hash_val;
    int key_len = tbl->key_len;

    hash_val = tbl->hash(pkey, dev);
```

tbl->hash() is a reference to arp_hash() function defined in net/ipv4/arp.c. This function takes the IP address of the neighbor and the associated device and returns a hash value that is used as an index for the hash buckets of the arp table.

```
static u32 arp_hash(const void *pkey, const struct net_device *dev)
{
    u32 hash_val;

    hash_val = *(u32*)pkey;
    hash_val ^= (hash_val>>16);
    hash_val ^= hash_val>>8;
    hash_val ^= hash_val>>3;
    hash_val = (hash_val^dev->ifindex)&NEIGH_HASHMASK;

    return hash_val;
}
```

On return to neigh_lookup the search proceeds.

```

    read_lock_bh(&tbl->lock);
    for (n = tbl->hash_buckets[hash_val]; n; n = n->next) {
         if (dev == n->dev &&
              memcmp(n->primary_key, pkey, key_len) == 0) {
                   neigh_hold(n);
                   break;
              }
         }
         read_unlock_bh(&tbl->lock);
         return n;
    }
```

After returning from neigh_lookup() that returns a pointer to an entry in the neighbour table, the state of the neighbor is checked.

```
    if (n) {
         state = n->nud_state;
         neigh_release(n);
    }
    if (state==NUD_REACHABLE)
         return 0;
    if ((state & NUD_VALID) && order != fn_hash_last_dflt)
         return 0;
    if ((state & NUD_VALID) || (*last_idx<0 && order > fn_hash_last_dflt)) {
         *last_resort = fi;
         *last_idx = order;
    }
    return 1;
}
```

We return from fib_detect_death() into fib_select_default()

```
                        if (res->fi)
                                fib_info_put(res->fi);
                        res->fi = fi;
                        atomic_inc(&fi->fib_clntref);
                        fn_hash_last_dflt = order;
                        goto out;
                }
                fi = next_fi;
                order++;
        }

        if (order<=0 || fi==NULL) {
                fn_hash_last_dflt = -1;
                goto out;
        }

        if (!fib_detect_death(fi, order, &last_resort,
                &last_idx)) {
                if (res->fi)
                        fib_info_put(res->fi);
                res->fi = fi;
                atomic_inc(&fi->fib_clntref);
                fn_hash_last_dflt = order;
                goto out;
        }

        if (last_idx >= 0) {
                if (res->fi)
                        fib_info_put(res->fi);
                res->fi = last_resort;
                if (last_resort)
                        atomic_inc(&last_resort->fib_clntref);
        }
        fn_hash_last_dflt = last_idx;
out:
        read_unlock(&fib_hash_lock);
}
```

Finally, back in the main line of ip_route_output_slow() a check is made to see if the source IP address remains NULL.

```

        if (!key.src)
```

If so an attempt is made to derive the source address from the fib_prefsrc field of the fib_info structure. If that field is also NULL then our old friend inet_select_addr() is asked to recover it from the net_device and nh_gw parameters. This makes no sense to me because the value of nh_gw should be an IP address that is owned by a different host!

```
                key.src = FIB_RES_PREFSRC(res);
```

FIB_RES_PREFSRC is a macro defined in include/net/ip_fib.h

```
#define FIB_RES_PREFSRC(res)((res).fi->fib_prefsrc ? :
                       __fib_res_prefsrc(&res))
```

```
u32 __fib_res_prefsrc(struct fib_result *res)
{
        return inet_select_addr(FIB_RES_DEV(*res),
                       FIB_RES_GW(*res), res->scope);
}
```

If a net device is held in dev_out, release it here.

```

        if (dev_out)
                dev_put(dev_out);
```

Set the value of key.oif from the net_device pointed to by the fib_info structure than lives in the res structure.

```
        dev_out = FIB_RES_DEV(res);
        dev_hold(dev_out);
        key.oif = dev_out->ifindex;
```

Now we are ready to create the route and add it to the route cache. After filling in the appropriate data, we determine the hash id and install the new route in the cache.

First ensure that if the source address is a loopback address then the selected output device carries the IFF_LOOPBACK flag. Couldn't this have been done earlier???

```
make_route:
    if (LOOPBACK(key.src) &&
              !(dev_out->flags&IFF_LOOPBACK))
         goto e_inval;

    if (key.dst == 0xFFFFFFFF)
         res.type = RTN_BROADCAST;
    else if (MULTICAST(key.dst))
         res.type = RTN_MULTICAST;
    else if (BADCLASS(key.dst) || ZERONET(key.dst))
         goto e_inval;

    if (dev_out->flags & IFF_LOOPBACK)
         flags |= RTCF_LOCAL;

```

If the result type is BROADCAST, then any fib_info structure that is held is released.

```
    if (res.type == RTN_BROADCAST) {
         flags |= RTCF_BROADCAST | RTCF_LOCAL;
         if (res.fi) {
              fib_info_put(res.fi);
              res.fi = NULL;
         }
    } else if (res.type == RTN_MULTICAST) {
         flags |= RTCF_MULTICAST|RTCF_LOCAL;
         read_lock(&inetdev_lock);
         if (!__in_dev_get(dev_out) ||
              !ip_check_mc(__in_dev_get(dev_out),
                        oldkey->dst))
              flags &= ~RTCF_LOCAL;
         read_unlock(&inetdev_lock);
         /* If multicast route do not exist use
              default one, but do not gateway in
              this case. Yes, it is hack.
          */
         if (res.fi && res.prefixlen < 4) {
                        fib_info_put(res.fi);
                        res.fi = NULL;
         }
    }
```

Allocate memory from the slab allocator for a route cache entry.

```
    rth = dst_alloc(&ipv4_dst_ops);
    if (!rth)
         goto e_nobufs;

    atomic_set(&rth->u.dst.__refcnt, 1);
```

Copy (most of) the elements of the key structure that was used to create the route to the key structure embedded the rth.

```
    rth->u.dst.flags= DST_HOST;
    rth->key.dst    = oldkey->dst;
    rth->key.tos    = tos;
    rth->key.src    = oldkey->src;
    rth->key.iif    = 0;
    rth->key.oif    = oldkey->oif;
#ifdef CONFIG_IP_ROUTE_FWMARK
    rth->key.fwmark = oldkey->fwmark;
#endif
```

Copy the elements used to route the packet to the rt_ fields of the route cache element.

```
    rth->rt_dst     = key.dst;
    rth->rt_src     = key.src;
#ifdef CONFIG_IP_ROUTE_NAT
    rth->rt_dst_map = key.dst;
    rth->rt_src_map = key.src;
#endif
    rth->rt_iif= oldkey->oif ? : dev_out->ifindex;
    rth->u.dst.dev   = dev_out;
    dev_hold(dev_out);
    rth->rt_gateway = key.dst;
    rth->rt_spec_dst= key.src;

```

Setup the function that will be used to transmit the packet.

```
    rth->u.dst.output=ip_output;

    rt_cache_stat[smp_processor_id()].out_slow_tot++;

```

If the flags indicate that this route terminates on this machine, then the input handler is set to ip_local_deliver.

```
    if (flags & RTCF_LOCAL) {
        rth->u.dst.input = ip_local_deliver;
        rth->rt_spec_dst = key.dst;
    }
    if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
        rth->rt_spec_dst = key.src;
        if (flags & RTCF_LOCAL &&
                  !(dev_out->flags & IFF_LOOPBACK)) {
            rth->u.dst.output = ip_mc_output;
            rt_cache_stat
                      [smp_processor_id()].out_slow_mc++;
        }
```

CONFIG_IP_MROUTE option is used if you want your machine to act as a router for IP packets that have multicast destination addresses.

```
#ifdef CONFIG_IP_MROUTE
        if (res.type == RTN_MULTICAST) {
            struct in_device *in_dev =
                      in_dev_get(dev_out);
                if (in_dev) {
                    if (IN_DEV_MFORWARD(in_dev) &&
                    !LOCAL_MCAST(oldkey->dst)) {
                    rth->u.dst.input =  ip_mr_input;
                    rth->u.dst.output = ip_mc_output;
                }
                in_dev_put(in_dev);
            }
        }
#endif
    }

```

Call rt_set_nexthop() defined in net/ipv4/route.c to set next neighbor parameters like pmtu and mss.

```
    rt_set_nexthop(rth, &res, 0);
```

```
static void rt_set_nexthop(struct rtable *rt, struct
                                fib_result *res, u32 itag)
{
    struct fib_info *fi = res->fi;

```

The bulk of this code seems to be attempting to address potential problems associated with missing or invalid elements in the fib_info structure.

```
    if (fi) {
        if (FIB_RES_GW(*res) &&
            FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
            rt->rt_gateway = FIB_RES_GW(*res);
            memcpy(&rt->u.dst.mxlock, fi->fib_metrics,
                        sizeof(fi->fib_metrics));
```

fib_mtu is actually a macro referencing the RTAX_MTU element of the fib_metrics array. If the value is zero it is copied from the net device. Oddly, it appears that rt->u.dst.pmtu has not been previously set in this module... so it is also set in the else clause!

```
        if (fi->fib_mtu == 0) {
            rt->u.dst.pmtu = rt->u.dst.dev->mtu;
            if (rt->u.dst.mxlock & (1 <<  RTAX_MTU) &&
                rt->rt_gateway != rt->rt_dst &&
                rt->u.dst.pmtu > 576)
                rt->u.dst.pmtu = 576;
        }
#ifdef CONFIG_NET_CLS_ROUTE
    rt->u.dst.tclassid = FIB_RES_NH(*res).nh_tclassid;
#endif
        } else
            rt->u.dst.pmtu  = rt->u.dst.dev->mtu;

        if (rt->u.dst.pmtu > IP_MAX_MTU)
            rt->u.dst.pmtu = IP_MAX_MTU;
        if (rt->u.dst.advmss == 0)
            rt->u.dst.advmss = max_t(unsigned int,
                rt->u.dst.dev->mtu - 40,
                ip_rt_min_advmss);
```

```
            if (rt->u.dst.advmss > 65535 - 40)
                 rt->u.dst.advmss = 65535 - 40;

#ifdef CONFIG_NET_CLS_ROUTE
#ifdef CONFIG_IP_MULTIPLE_TABLES
            set_class_tag(rt, fib_rules_tclass(res));
#endif
            set_class_tag(rt, itag);
#endif
            rt->rt_type = res->type;
       }
```

On return to ip_route_output_slow(), use the source address, destination address, and tos to determine and return a hash value by invoking the rt_hash_code() function defined in net/ipv4/route.c We had visited this function earlier in UDP connect and was called by the ip_route_output_key() function.

```
       rth->rt_flags = flags;
       hash = rt_hash_code(oldkey->dst, oldkey->src ^
                           (oldkey->oif << 5), tos);
```

The hash code returned is used by rt_intern_hash() function to search in the respective hash queue of routing cache (rt_hash_table) to find an entry that matches the entry that was just created. The rp parameter was passed in to ip_route_output_slow() as the location at which a pointer to the new route cache entry should be returned.

```
       err = rt_intern_hash(hash, rth, rp);

static int rt_intern_hash(unsigned hash, struct rtable
                          *rt, struct rtable **rp)
{
       struct rtable   *rth, **rthp;
       unsigned long   now = jiffies;
       int attempts = !in_softirq();

restart:
```

Recall that the route cache is based upon a table of structures. Each structure contains a pointer to the first struct rtable element in the hash queue and a lock for the queue. Here the queue is locked and the value of rthp is set to point to the chain header (as opposed to set to the chain header!) As the while loop continues rthp will be advanced.

```
        rthp = &rt_hash_table[hash].chain;

        write_lock_bh(&rt_hash_table[hash].lock);
```

This loop appears to be looking for the possible case that the route already exists! This could conceivably occur due to race conditions involving multiple callers of ip_route_output(). If an existing entry with the same key is found, the existing entry is used and the newly created one is dropped.

```
        while ((rth = *rthp) != NULL) {
             if (memcmp(&rth->key, &rt->key,
                       sizeof(rt->key)) == 0) {
                            /* Put it first */
                  *rthp = rth->u.rt_next;
                  rth->u.rt_next = rt_hash_table[hash].chain;
                  rt_hash_table[hash].chain = rth;
```

Update the reference count and the last use of the existing entry.

```
                  rth->u.dst.__use++;
                  dst_hold(&rth->u.dst);
                  rth->u.dst.lastuse = now;

                  write_unlock_bh(&rt_hash_table
                                         [hash].lock);

                  rt_drop(rt);
                  *rp = rth;
                  return 0;
             }

             rthp = &rth->u.rt_next;
        }
```

```
    /* Try to bind route to arp only if it is output
         route or unicast forwarding path.
    */
    if (rt->rt_type == RTN_UNICAST || rt->key.iif== 0) {
         int err = arp_bind_neighbour(&rt->u.dst);
```

The arp_bind_neighbour() function defined in net/ipv4/arp.c is invoked. This function tries to locate an entry in the ARP table for the destination address, if one exists.

```
int arp_bind_neighbour(struct dst_entry *dst)
{
    struct net_device *dev = dst->dev;
    struct neighbour *n = dst->neighbour;

    if (dev == NULL)
         return -EINVAL;
    if (n == NULL) {
         u32 nexthop = ((struct rtable*)dst)->rt_gateway;
         if (dev->flags&(IFF_LOOPBACK|IFF_POINTOPOINT))
              nexthop = 0;
         n = __neigh_lookup_errno(
#ifdef CONFIG_ATM_CLIP
         dev->type == ARPHRD_ATM ? &clip_tbl :
#endif
         &arp_tbl, &nexthop, dev);
```

neigh_lookup_errno() is defined in include/net/neighbour.h

```
static inline struct neighbour *
_neigh_lookup_errno(struct neigh_table *tbl,const void *pkey,
  struct net_device *dev)
{
    struct neighbour *n = neigh_lookup(tbl, pkey, dev);

    if (n)
         return n;

    return neigh_create(tbl, pkey, dev);
}
```

```
         if (IS_ERR(n))
              return PTR_ERR(n);
         dst->neighbour = n;
    }
    return 0;
}
```

```
         if (err) {
              write_unlock_bh(&rt_hash_table[hash].lock);

              if (err != -ENOBUFS) {
                   rt_drop(rt);
                   return err;
              }

/* Neighbour tables are full and nothing
   can be released. Try to shrink route cache,
   it is most likely it holds some neighbour records.
*/

              if (attempts-- > 0) {
                   int saved_elasticity =
                        ip_rt_gc_elasticity;
                   int saved_int = ip_rt_gc_min_interval;
                   ip_rt_gc_elasticity  = 1;
                   ip_rt_gc_min_interval   = 0;
                        rt_garbage_collect();
                   ip_rt_gc_min_interval =  saved_int;
                   ip_rt_gc_elasticity = saved_elasticity;
                   goto restart;
              }


              if (net_ratelimit())
                   printk("Neighbour table overflow.\n");
              rt_drop(rt);
              return -ENOBUFS;
         }
    }
```

Here the new route is inserted at the head of the hash queue.

```
            rt->u.rt_next = rt_hash_table[hash].chain;
#if RT_CACHE_DEBUG >= 2
            if (rt->u.rt_next) {
                 struct rtable *trt;
                 printk("rt_cache @%02x: %u.%u.%u.%u", hash,
                             NIPQUAD(rt->rt_dst));
                 for (trt = rt->u.rt_next; trt;
                                trt = trt->u.rt_next)
                 printk(" . %u.%u.%u.%u", NIPQUAD(trt->rt_dst));
                 printk("\n");
            }
#endif
            rt_hash_table[hash].chain = rt;
            write_unlock_bh(&rt_hash_table[hash].lock);
            *rp = rt;
            return 0;
       }
```

Release reference to FIB table and the device, if holding.

```
done:
     if (free_res)
          fib_res_put(&res);
     if (dev_out)
          dev_put(dev_out);
out: return err;

e_inval:
     err = -EINVAL;
     goto done;
e_nobufs:
     err = -ENOBUFS;
     goto done;
}
```

To summarize, the ip_route_output_slow() function does the following:

Creates a routing table cache key
If the source address is specified, calls ip_dev_find() to determine the output device.
If the oif is specified, use dev_get_by_index to retrieve output device and select source addr (if the dest address was not NULL (p.22).
If the destination address is not known, set up loopback
Calls fib_lookup() to find route to destination.
Allocates memory for new routing cache entry and initializes it.
Calls rt_set_nexthop() to find next destination.
Returns rt_intern_hash(), which creates a new route in the routing cache.