# Sheet 6

# FIB structures and lookup code

**Longest match – and how we do it in Linux. We achieve what we want using the Forwarding Information Base (FIB), which is a complex structure in the kernel, containing the routing information we need indexed on its network mask. BTW all routes with the same network mask are said to be in the same 'zone'.**

**First a look at some structures, then a look at the code that manipulates them.**

## struct fib_table

**This is include/net/ip_fib.h::fib_table. It's the starting point for FIB traversal and is instantiated with data appropriate to the type of network we are using at any given point in time.**

```
struct fib_table
{
    unsigned char  tb_id;
```

**Timestamp**

```
    unsigned       tb_stamp;
```

**Routines for lookup. These are set in net/ipv4/fib_hash.c::fib_hash_init to be net/ipv4/fib_hash.c::[fn_hash_lookup, fn_hash_insert, fn_hash_delete, etc.]**

```
    int          (*tb_lookup)(struct fib_table *tb, const struct rt_key *key,
                     struct fib_result *res);
    int          (*tb_insert)(struct fib_table *table, struct rtmsg *r,
                     struct kern_rta *rta, struct nlmsghdr *n,
                     struct netlink_skb_parms *req);
    int          (*tb_delete)(struct fib_table *table, struct rtmsg *r,
                     struct kern_rta *rta, struct nlmsghdr *n,
```

```
                         struct netlink_skb_parms *req);
    int           (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
                         struct netlink_callback *cb);
    int           (*tb_flush)(struct fib_table *table);
    int           (*tb_get_info)(struct fib_table *table, char *buf, int first,
                         int count);
    void          (*tb_select_default)(struct fib_table *table,
                         const struct rt_key *key, struct fib_result *res);

```

**And this is set to be a net/ipv4/fib_hash.c::fn_hash, described below.**

```
    unsigned char  tb_data[0];
};

```

## struct fib_hash

**This is net/ipv4/fib_hash.c::fn_hash and is the data we're talking about above. As you can see, we split the table into zones at a very high level. There are 33 possible netmasks (0x0000 to 0xFFFF) and a zone is defined by one of these. Also, the zones are all linked together, and the second field here points to the head of the list of zones.**

```
struct fn_hash
{
    struct fn_zone   *fn_zones[33];
    struct fn_zone   *fn_zone_list;
};
```

## struct fib_zone

**This is net/ipv4/fib_hash.c::fn_zone. It defines some housekeeping things about the hash table associated with each zone.**

```
struct fn_zone
{
```

**Pointer to next non-empty zone in the hash structure where the netmask is *less* restrictive (= shorter) than this**

```
    struct fn_zone    *fz_next;         /* Next not empty zone    */
```

**This is a pointer to the hash table.**

```
    struct fib_node   **fz_hash;        /* Hash table pointer */
```

**The number of entries in this zone**

```
    int             fz_nent;            /* Number of entries  */
```

**The number of buckets in the hash table associated with this zone (initially set to 16 for all zones but zone 0 in net/ipv4/fib_hash.c::fn_new_zone, but reset to 256 or 1024 in net/ipv4/fib_hash.c::fn_rehash_zone if no. entries grows)**

```
    int             fz_divisor;         /* Hash divisor       */
```

**Used so we can ensure that a hash value lies in the range [0, fz_divisor-1] – see net/ipv4/fib_hash.c::fn_hash**

```
    u32             fz_hashmask;        /* (1<<fz_divisor) - 1    */
```

**The index in the parent fn_hash structure (i.e. 0 to 32)**

```
    int             fz_order;           /* Zone order         */
```

**This is the netmask for fz_order = 0, fz_mask = 0x0000, for for fz_order = 1, fz_mask = 0x8000, for fz_order = 2, fz_mask = 0xC000, ... , for fz_order = 32, fz_mask = 0xFFFF,**

```
    u32             fz_mask;

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

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

```

## struct fib_node

**This is net/ipv4/fib_hash.c::fib_node. It's an entry in an open hash table that contains details about this particular route.**

```
struct fib_node
{
```

**It's an open hash table, so this is the link to the next item on the chain.**

```
    struct fib_node *fn_next;

```

**Key is set to be the network part of an IP address against which addresses (masked with fz_mask from above) will be tested for equality.**

```
    fn_key_t         fn_key;

```

**When we have something that matches the key, the details about this route are held in fn_info. Since many routes will have the the same next hop, this is a pointer to a shared structure**

```
    struct fib_info *fn_info;

    u8               fn_tos;
    u8               fn_type;
    u8               fn_scope;
    u8               fn_state;

#define FIB_INFO(f)   ((f)->fn_info)
};
```

## struct fib_info

**This is include/net/ip_fib.h::fib_info This structure contains data specific to an interface and, therefore, common to many zones.**

```
struct fib_info
{
    struct fib_info   *fib_next;
    struct fib_info   *fib_prev;
```

**Index to network protocol (e.g. IP) used for this route.**

```
    int               fib_protocol;
```

**Pointer to next hop information**

```
    struct fib_nh     fib_nh[0];

```

**Housekeeping stuff. Since this is a shared structure, we care about reference counting carefully, amongst other things.**

```
    int               fib_treeref;
    atomic_t          fib_clntref;
    int               fib_dead;
    unsigned          fib_flags;
    u32               fib_prefsrc;
    u32               fib_priority;
    unsigned          fib_metrics[RTAX_MAX];
    int               fib_nhs;
```

**<Multipath stuff deleted>**

**<some #defines deleted>**

```
};

```

## struct fib_nh

**This is [include/net/ip_fib.h::fib_nh]() Next hop structure – defined in terms of the output device or the IP address of the next hop gateway.**

```
struct fib_nh
{
    struct net_device *nh_dev;
    unsigned          nh_flags;
    unsigned char     nh_scope;
    <Multipath, class stuff deleted>
    int               nh_oif;
    u32               nh_gw;
};
```

## fib_lookup

**This is net/ipv4/fib_rules.c::fib_lookup and was called from ip_route_input_slow**

**Different rules can be applied to forwarding to different destinations. There might be a rule prohibiting output, or one saying that we use NAT, but the simplest one and that with which we're really concerned is simple unicast (RTN_UNICAST below)**

```
int fib_lookup(const struct rt_key *key, struct fib_result *res)
{
    int err;
    struct fib_rule *r, *policy;
    struct fib_table *tb;

    u32 daddr = key->dst;
    u32 saddr = key->src;

FRprintk("Lookup: %u.%u.%u.%u <- %u.%u.%u.%u ",
    NIPQUAD(key->dst), NIPQUAD(key->src));
    read_lock(&fib_rules_lock);
```

**Look for the relevant rule associated with this dest. By default we'll unicast.**

```
    for (r = fib_rules; r; r=r->r_next) {
        if (((saddr^r->r_src) & r->r_srcmask) ||
```

```
                ((daddr^r->r_dst) & r->r_dstmask) ||
#ifdef CONFIG_IP_ROUTE_TOS
                (r->r_tos && r->r_tos != key->tos) ||
#endif
#ifdef CONFIG_IP_ROUTE_FWMARK
                (r->r_fwmark && r->r_fwmark != key->fwmark) ||
#endif
                (r->r_ifindex && r->r_ifindex != key->iif))
                continue;

FRprintk("tb %d r %d ", r->r_table, r->r_action);

```

**This is where we decide whether we're going to do something or return an error.**

```
           switch (r->r_action) {
                case RTN_UNICAST:
                case RTN_NAT:
                     policy = r;
                     break;
                case RTN_UNREACHABLE:
                     read_unlock(&fib_rules_lock);
                     return -ENETUNREACH;
                default:
                case RTN_BLACKHOLE:
                     read_unlock(&fib_rules_lock);
                     return -EINVAL;
                case RTN_PROHIBIT:
```

```
                read_unlock(&fib_rules_lock);
                return -EACCES;
        }

```

**Given that we've decided we're going to do something, get a handle on the correct FIB.**

```
        if ((tb = fib_get_table(r->r_table)) == NULL)
            continue;

```

**And perform the lookup. In our case, this is set to be net/ipv4/fib_hash.c::fn_hash_lookup – see below.**

```
        err = tb->tb_lookup(tb, key, res);

        if (err == 0) {
            res->r = policy;
            if (policy)
                atomic_inc(&policy->r_clntref);
            read_unlock(&fib_rules_lock);
            return 0;
        }
        if (err < 0 && err != -EAGAIN) {
            read_unlock(&fib_rules_lock);
            return err;
        }
    }
FRprintk("FAILURE\n");
    read_unlock(&fib_rules_lock);
```

```
        return -ENETUNREACH;
}

```

## fn_hash_lookup

**This is net/ipv4/fib_hash.c::fn_hash_lookup and, actually, relatively straightforward. There are a number of static inline functions used (all in net/ipv4/fib_hash.c)**

| | | |
|---|---|---|
| **fn_key_t** | **fz_key(u32 dst, struct fn_zone *fz)** | **returns an address that has been masked by the netmask for a given zone** |
| **struct fib_node *** | **fz_chain(fn_key_t key, struct fn_zone *fz)** | **hashes the key and returns the head of the chain of node structures that match for this zone** |
| **int** | **fn_key_eq(fn_key_t a, fn_key_t b)** | **Compare keys and say if they're equal** |
| **int** | **fn_key_leq(fn_key_t a, fn_key_t b)** | **Same but for leq** |

**As a matter of interest, the top two routines are defined thus:**

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

fz_chain(fn_key_t key, struct fn_zone *fz) {
    return fz->fz_hash[fn_hash(key, fz).datum]; }
```

**And the all important hash function is defined thus: N.B. in C, ^ is XOR**

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

**The algorithm used for lookup is a simple linear search on a series of open hash tables, rather than anything massively sophisticated. Note that the elements of each chain in a hash table entry are held ordered by key value.**

```
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;

    read_lock(&fib_hash_lock);

```

**Start with the most restrictive zone and interate over zones with smaller and smaller netmasks**

```
    for (fz = t->fn_zone_list; fz; fz = fz->fz_next) {
        struct fib_node *f;
```

**Mask the destination appropriately to produce the lookup key for this zone**

```
        fn_key_t k = fz_key(key->dst, fz);
```

**Now, do a hash (implicit in fz_chain) and walk down the open hash table chain returned looking for a match. As a matter of interest, the hash is defined as:**

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

**Did we find it, or did we go past it? If neither, then keep chaining down.**

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

**If we come here, we've found something where the keys match. However, we have to be careful, make sure that it's a proper match**

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

```
#endif
            f->fn_state |= FN_S_ACCESSED;

            if (f->fn_state&FN_S_ZOMBIE)
                continue;
            if (f->fn_scope < key->scope)
                continue;

```

**net/ipv4/fib_semantics.c::fib_semantic_match is a routine to make really sure we're allowed to use this interface for this packet; it also fills in some fields in res, notably res->fi which points to the fib_info structure passed as the second arg. If we are allowed to proceed, then fill in a result structure with info about this node and return.**

```
            err = fib_semantic_match(f->fn_type, FIB_INFO(f), key, res);
            if (err == 0) {
                res->type      = f->fn_type;
                res->scope     = f->fn_scope;
                res->prefixlen = fz->fz_order;

                goto out;
            }
            if (err < 0)
                goto out;
        }
    }
    err = 1;
out:
```

```
    read_unlock(&fib_hash_lock);
    return err;
}
```