Linux Networking Subsystem

Desktop Companion to the Linux Source Code

by Ankit Jain

Linux-2.4.18,
Version 0.1, 31 May ‘02

Contents

Preface

This document attempts to describe how the networking subsystem is implemented in the Linux kernel. It is based on the Linux-2.4.18 kernel. The reader is assumed to have some knowledge of networking concepts. This document is best read with the kernel source by your side.

Acknowledgements

While preparing this document, I asked for reviewers on `#kernelnewbies` on irc.openprojects.net. I got a lot of response. The following individuals helped me with corrections, suggestions and material to improve this paper. They put in a big effort to help me get this document into its present shape. I would like to sincerely thank all of them. Naturally, all the mistakes you'll find in this book are mine.

Chapter 1

Introduction

This is the introduction. [fixme!]

1.1 Background

1.2 Document Conventions

1.3 Sample Network Example

Chapter 2

Initialization

This chapter presents the Network Subsystem's initialization on startup. It provides a walkthrough of the whole initialization process. The function that starts it all is sock_init(). [fixme: overview]

The call graph for the initialization process is given below, but only the top level functions are shown :

2.1 Function do_basic_setup()

File: `init/main.c`

This function does some setup work for the system including networking initialization. The code is quoted below with only the parts relevant to networking being shown.

```
static void __init do_basic_setup(void)
{
    .......

    /* Networking initialization needs a process context */
    sock_init();
    ...
    do_initcalls();

    .......
}
```

Both these functions call other functions to do the actual setup and initialization of the networking subsystem. All these functions are discussed below :

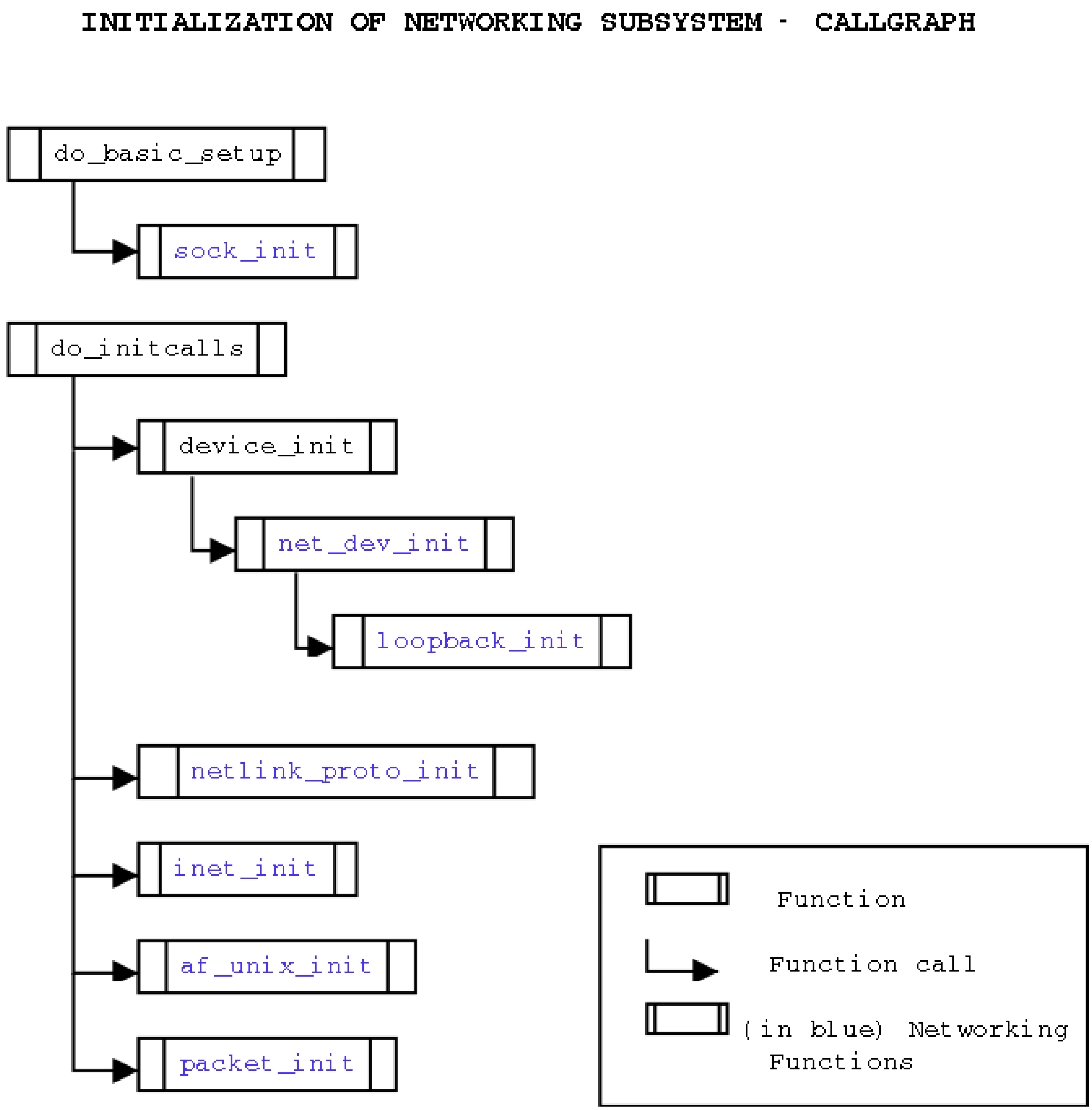

Figure 2.1: Initialization starts

2.2 Function sock_init()

File: net/socket.c
Prototype:

```
void sock_init(void)
```

This is the function which initializes the networking subsystem on bootup. Refer to figure 2.2 for its callgraph.

```
int i;

printk(KERN_INFO "Linux NET4.0 for Linux 2.4\n");
printk(KERN_INFO "Based upon Swansea University
                            Computer Society Net3.039\n");
```

These are the 2 lines printed by kernel which mark the start of the initialization of networking in the kernel. You can find these in the kernel ring buffer, by using the command 'dmesg'.

```
/*
 * Initialize all address (protocol) families.
 */
for (i=0; i < NPROTO; i++)
    net_families[i] = NULL;
```

The above for loop 'clears' the `net_families` table. `net_families[]` is an array of `struct net_proto_family` pointers, which is just a table where all the protocols families are registered. For example: INET, AppleTalk, ATM etc. Refer to 2.5.1.1 for more on this.

`NPROTO`[1] defines the maximum number of protocols that can be registered. Its set to 32 in kernel 2.4.18.

```
/*
 * Initialize sock SLAB cache.
 */
sk_init();
```

The above function allocates a memory cache for `struct sock` type objects. Refer to 2.2.1 for more on this function.

[1] include/linux/net.h

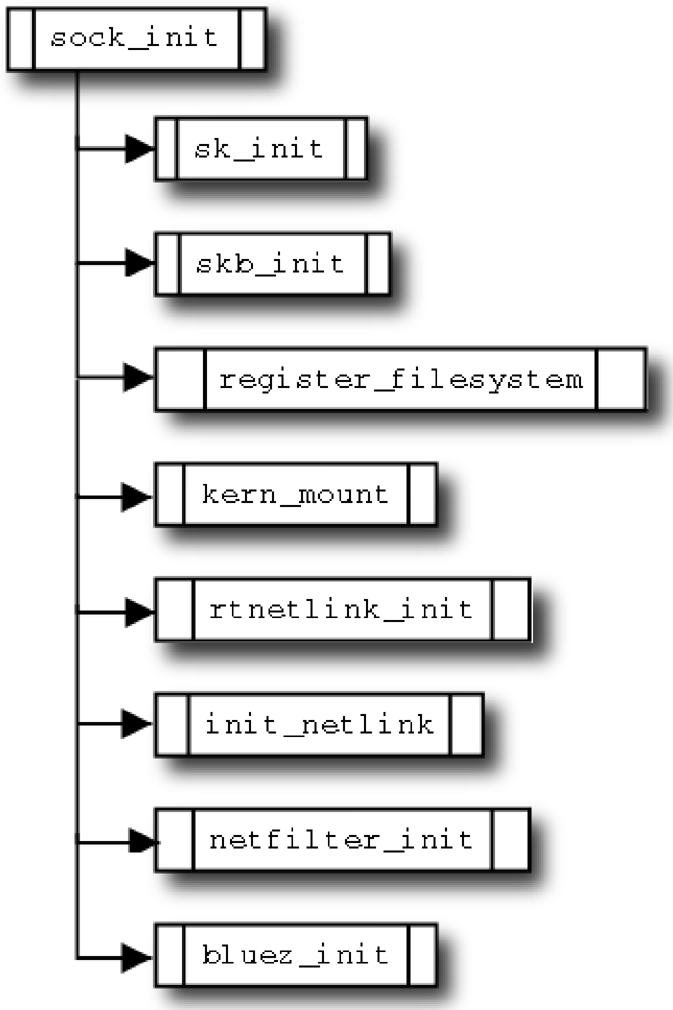

Figure 2.2: Callgraph for Function sock_init

```
#ifdef SLAB_SKB
    /*
     * Initialize skbuff SLAB cache
     */
    skb_init();
#endif
```

`skb_init()` function allocates a memory cache for `struct sk_buff` type objects. More on this in 2.2.2.

```
    /*
     * Wan router layer.
     */
#ifdef CONFIG_WAN_ROUTER
       wanrouter_init();
#endif
```

Initializes the Wan Router layer. See 2.2.3.

```
    /*
     * Initialize the protocols module.
     */
    register_filesystem(&sock_fs_type);
    sock_mnt = kern_mount(&sock_fs_type);

    /* The real protocol initialization is performed when
     * do_initcalls is run.
     */
```

The above code registers the file system socket abstraction[fix] and then mounts it. The filesystem is described by `sock_fs_type`. `sock_mnt` is a pointer to `struct vfsmount`.

```
    /*
     * The netlink device handler may be needed early.
     */
```

```
#ifdef CONFIG_NET
    rtnetlink_init();
#endif
```

Initializes the routing netlink socket interface. Implementation of this function is explained in 2.2.5.

```
#ifdef CONFIG_NETLINK_DEV
    init_netlink();
#endif
```

If `CONFIG_NETLINK_DEV` is defined then initialize netlink devices. Refer to 2.2.6

```
#ifdef CONFIG_NETFILTER
    netfilter_init();
#endif
```

`netfilter_init()` function performs some basic initialization for the netfilter architecture in the kernel. For more on this refer to 2.2.7.

```
#ifdef CONFIG_BLUEZ
    bluez_init();
#endif
}
```

The above function initializes the Bluetooth subsystem.(2.2.8)

The following are the functions and their descriptions in the order in which they are called to initialize the parts of the subsystem.

2.2.1 Function sk_init()

File : `net/core/sock.c`
Prototype:

```
void sk_init(void)
```

In this function the sock slab cache is initialized. 'sock' is the entity, `struct sock`[2], used by the kernel to represent a socket internally. It is protocol independent and is different from `struct socket`, which will be discussed later.

```
sk_cachep = kmem_cache_create("sock", sizeof(struct sock),
                              0, SLAB_HWCACHE_ALIGN, 0, 0);
```

Objects of type `sock` are created and used a lot throughout the networking subsystem. So its more efficient to reserve a 'memory pool' or 'memory cache' for allocation of objects of this type. Memory caches have a type of `kmem_cache_t` and are created with a call to `kmem_cache_create()`.

The above line of code creates a memory cache by the name of "sock", with each object being of size `sizeof(struct sock)` and assigns it to `sk_cachep` of type `kmem_cache_t` pointer.

```
if (!sk_cachep)
    printk(KERN_CRIT "sk_init: Cannot create sock SLAB cache!");
```

If the requested cache wasn't allocated then print an error message with log level `KERN_CRIT`.

```
if (num_physpages <= 4096) {
    sysctl_wmem_max = 32767;
    sysctl_rmem_max = 32767;
    sysctl_wmem_default = 32767;
    sysctl_rmem_default = 32767;
} else if (num_physpages >= 131072) {
    sysctl_wmem_max = 131071;
    sysctl_rmem_max = 131071;
}
```

The above code sets the default and the maximum values for the read and write buffers. If the number of physical pages (*num_physpages*) available on the system is less than 4096 (16MB on x86), then we set the values to 32767 bytes (32Kb -1).

But in case we have more than or equal to 131072 pages (512MB on x86) then we set the values to 131071 bytes (128Kb - 1).

What if the system RAM size doesn't satisfy any of those two conditions? Well, for that a default value is assigned to these variables in `net/core/sock.c`. Here is the code quoted from the source :

[2] `include/net/sock.h`

```
/* Run time adjustable parameters. */
__u32 sysctl_wmem_max = SK_WMEM_MAX;
__u32 sysctl_rmem_max = SK_RMEM_MAX;
__u32 sysctl_wmem_default = SK_WMEM_MAX;
__u32 sysctl_rmem_default = SK_RMEM_MAX;
```

`SK_WMEM_MAX`, `SK_RMEM_MAX` are defined in `include/linux/skbuff.h` and set to 65535 bytes (64K - 1).

2.2.2 Function skb_init()

File : `net/core/skbuff.c`
Prototype:

```
void skb_init(void)
```

This function creates the cache for objects of type `struct sk_buff`, and initializes the per-cpu skbuff queues. It is called by `sock_init()` only if `SLAB_SKB` is defined.

```
    int i;

    skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
                             sizeof(struct sk_buff),
                             0,
                             SLAB_HWCACHE_ALIGN,
                             skb_headerinit, NULL);
```

The above code creates the memory cache for objects of `struct sk_buff` type.

```
    if (!skbuff_head_cache)
        panic("cannot create skbuff cache");
```

Creation of the memory cache is critical, failing which the kernel panics via the `panic()` function.

```
    for (i=0; i<NR_CPUS; i++)
        skb_queue_head_init(&skb_head_pool[i].list);
```

The above loop initializes the per-CPU skbuff queues by calling `skb_queue_head_init()` (section 2.2.2.2). `skb_head_pool[i].list` is the skbuff queue.

2.2.2.1 union skb_head_pool

```
static union {
    struct sk_buff_head  list;
    char            pad[SMP_CACHE_BYTES];
} skb_head_pool[NR_CPUS];
```

`skb_head_pool` is an array of `NR_CPUS` `struct sk_buff_heads`, padding each struct out to `SMP_CACHE_BYTE` (32, the size of an x86 L1 cacheline) for performance reasons.

list

pad

2.2.2.2 Function skb_queue_head_init()

File : `include/linux/skbuff.h`
Prototype:

```
void skb_queue_head_init(struct sk_buff_head *list)
```

This function initializes the sk_buff queue.

```
spin_lock_init(&list->lock);
```

This initializes the spinlock for the queue. Refer to 2.2.2.3 for description of `sk_buff_head`.

```
list->prev = (struct sk_buff *)list;
list->next = (struct sk_buff *)list;
list->qlen = 0;
```

The above lines initialize the `prev` and `next` pointers of the queue to the head itself and set the queue length to zero.

2.2.2.3 struct sk_buff_head

File : `include/linux/skbuff.h`

```
struct sk_buff_head {
    struct sk_buff * next;
    struct sk_buff * prev;

    __u32           qlen;
    spinlock_t      lock;
};
```

next and `prev` are simply pointers to the next and the previous elements in the queue.

qlen It is the length of the queue.

lock It is the spinlock used for serializing access to this queue.

2.2.3 Function wanrouter_init()

File : `net/wanrouter/wanmain.c`
Prototype:

```
int wanrouter_init(void)
```

This function is called by `sock_init` only if `CONFIG_WAN_ROUTER` is defined.

2.2.3.1 Function wanrouter_proc_init()

2.2.3.2 Function sdladrv_init()

2.2.3.3 wanpipe_init()

2.2.4 Netlink sockets

Netlink is used to transfer information between kernel modules and user space processes. It consists of a standard sockets based interface for user processes and an internal kernel API for kernel modules.

2.2.5 Function rtnetlink_init()

File : `net/core/rtnetlink.c`
Prototype:

```
void rtnetlink_init(void)
```

This function initializes routing netlink socket interface. Rtnetlink allows the kernels routing tables to be read and altered. Rtnetlink allows to set up and read network routes, ip addresses, link parameters, neighbour setups, queueing disciplines and traffic classes packet classifiers.

```
#ifdef RTNL_DEBUG
    printk("Initializing RT netlink socket.\n");
#endif
```

If RTNL_DEBUG is defined then the message above is printed for debugging purposes. RTNL_DEBUG[3] is defined by default.

```
rtnl = netlink_kernel_create(NETLINK_ROUTE, rtnetlink_rcv);
```

rtnl is a struct sock pointer, which is assigned the netlink socket created via the function call netlink_kernel_create(..).

```
if (rtnl == NULL)
    panic("rtnetlink_init: cannot initialize rtnetlink\n");
```

If the socket could not be created then 'panic', as the *rtnetlink* is important for the networking subsystem.

```
register_netdevice_notifier(&rtnetlink_dev_notifier);

rtnetlink_links[PF_UNSPEC] = link_rtnetlink_table;
rtnetlink_links[PF_PACKET] = link_rtnetlink_table;
```

[fixme:netlink hooks?]

2.2.5.1 Function netlink_kernel_create()

2.2.5.2 Function register_netdevice_notifier()

2.2.6 Function init_netlink()

File : net/netlink/netlink_dev.c
Prototype:

```
int init_netlink(void)
```

This function initializes the netlink socket interface.[fixme]

```
if (devfs_register_chrdev(NETLINK_MAJOR,"netlink",
                                          &netlink_fops)) {

    printk(KERN_ERR "netlink: unable to get major %d\n",
                                              NETLINK_MAJOR);
    return -EIO;
}
```

[3] include/linux/rtnetlink.h

The above code tries to register a char device driver with devfs(device filesystem).

NETLINK_MAJOR This is the major number of the driver.

"netlink" name of the driver.

netlink_fops pointer to the file operations structure.

```
devfs_handle = devfs_mk_dir (NULL, "netlink", NULL);
```

Devfs has a highly hierarchical structure so we create a directory called `netlink`, `/dev/netlink`, to contain nodes for [fixme] .

```
/*Someone tell me the official names for the uppercase ones*/
make_devfs_entries ("route", 0);
make_devfs_entries ("skip", 1);
make_devfs_entries ("usersock", 2);
make_devfs_entries ("fwmonitor", 3);
make_devfs_entries ("tcpdiag", 4);
make_devfs_entries ("arpd", 8);
make_devfs_entries ("route6", 11);
make_devfs_entries ("ip6_fw", 13);
make_devfs_entries ("dnrtmsg", 13);
```

The lines above create nodes or entries under the directory that we created in the previous step, `/dev/netlink`, using the function `make_devfs_entries()`.

The first argument is the node name and the second one is the minor number for the node. `make_devfs_entries()` simply calls `devfs_register()` for creating the entry.

```
devfs_register_series (devfs_handle, "tap%u", 16,
                       DEVFS_FL_DEFAULT,
                       NETLINK_MAJOR, 16,
                       S_IFCHR | S_IRUSR | S_IWUSR,
                       &netlink_fops, NULL);
return 0;
```

The above code creates more nodes under `/dev/netlink`. The function `devfs_register_series()` is just a convinient wrapper function to create n number of nodes following the same naming scheme. Since it just calls `devfs_register()` to register the nodes, the parameters are the same except for second and third parameter.

"tap%u" The nodes will be named tap0,tap1.. tap15, representing the 16 ethertaps available. Ethertap is a pseudo network tunnel device that allows an ethernet driver to be simulated from user space.[fixme:?]

16 Third argument, the number of nodes to create

2.2.7 Function netfilter_init()

*File :*net/core/netfilter.c
Prototype:

```
void __init netfilter_init(void)
```

This function initializes the per-protocol list of hooks represented by the array `nf_hooks[][]`. It is called only if `CONFIG_NETFILTER` is enabled.

```
int i, h;

for (i = 0; i < NPROTO; i++) {
    for (h = 0; h < NF_MAX_HOOKS; h++)
        INIT_LIST_HEAD(&nf_hooks[i][h]);
}
```

`nf_hooks[][]` is declared in the same file :

```
struct list_head nf_hooks[NPROTO][NF_MAX_HOOKS];
```

`NF_MAX_HOOKS`, maximum number of hooks per protocol, is defined with a value of `8` in include/linux/netfilter.h.

2.2.8 Function bluez_init()

*File :*net/bluetooth/af_bluetooth.c
Prototype:

```
int bluez_init(void)
```

2.3 Function do_initcalls()

File: `init/main.c`

This function calls various functions to do initialization of different parts of the kernel. [fixme: better description required,proto init happens here]. Following are the functions in the order in which they are called to initialize the various protocols.

1. netlink_proto_init()
2. inet_init()
3. af_unix_init()
4. packet_init()

2.4 Function netlink_proto_init()

File : `net/netlink/af_netlink.c`
`Prototype:`

```
int __init netlink_proto_init(void)
```

This function registers the netlink protocol family with the networking subsystem and creates the relevant /proc entry.

```
struct sk_buff *dummy_skb;

if (sizeof(struct netlink_skb_parms) > sizeof(dummy_skb->cb)){
    printk(KERN_CRIT "netlink_init: panic\n");
    return -1;
}
```

The code above prints an error message and returns with an error code -1 if the `struct netlink_skb_parms` is too big to fit in the control buffer of a `struct sk_buff`. `dummy_skb` is created for the sole purpose of getting the size of the control buffer of `struct sk_buff`.

```
sock_register(&netlink_family_ops);
```

Register the netlink protocol family via the function `sock_register()`. See 2.5.1 for more on `sock_register()`. `netlink_family_ops` is defined as (in the same file):

```
struct net_proto_family netlink_family_ops = {
        PF_NETLINK,
        netlink_create
};
```

PF_NETLINK[4] Protocol family number/id

netlink_create Function which will create a *netlink* socket.

Refer to 2.5.1.1 for more on `struct net_proto_family`.

```
#ifdef CONFIG_PROC_FS
  create_proc_read_entry("net/netlink", 0, 0,
                         netlink_read_proc, NULL);
#endif
```

If `CONFIG_PROC_FS` is defined, then create the proc entry *"net/netlink"*, `netlink_read_proc` being the function which is called when this proc entry is read.

```
  return 0;
```

Return with no errors!

2.5 Function inet_init()

File : `net/ipv4/af_inet.c`
`Prototype:`

```
void inet_init(void)
```

This function does the intialization of the INET family of protocols. [fixme: INET -reference]

[4] `include/linux/socket.h`

```
struct sk_buff *dummy_skb;
struct inet_protocol *p;
struct inet_protosw *q;
struct list_head *r;

printk(KERN_INFO "NET4: Linux TCP/IP 1.0 for NET4.0\n");

if (sizeof(struct inet_skb_parm) > sizeof(dummy_skb->cb)) {
    printk(KERN_CRIT "inet_proto_init: panic\n");
    return -EINVAL;
}
```

The above code prints the message to announce the initialization of the INET protocol family. Then it checks to make sure that the ip-specific parameters contained in `struct inet_skb_parm` can fit in `dummy_skb->cb`, `sk_buff`'s control block.

Control block is 48bytes in size and is used differently by different layers. If it wont fit in the control buffer, then an error is printed and the function returns with an error code `-EINVAL`.

```
/*
 *      Tell SOCKET that we are alive...
 */

(void) sock_register(&inet_family_ops);
```

The above function registers the *inet* family with the subsystem. For more on this, refer to 2.5.1 `inet_family_ops` is defined in the same file as :

```
struct net_proto_family inet_family_ops = {
        family: PF_INET,
        create: inet_create
}
```

PF_INET[5] Protocol family number/id.

inet_create Function which will create an *inet* socket.

[5] `include/linux/socket.h`

```
/*
 *      Add all the protocols.
 */

printk(KERN_INFO "IP Protocols: ");
for (p = inet_protocol_base; p != NULL;) {
    struct inet_protocol *tmp =
           (struct inet_protocol *) p->next;
    inet_add_protocol(p);
    printk("%s%s",p->name,tmp?", ":"\n");
    p = tmp;
}
```

The above code adds all the protocols in the inet protocol family to the networking subsystem. The variable `inet_protocol_base` is of type `struct inet_protocol` pointer and points to the base of the list of inet protocol definitions. For more on this refer to 2.5.2.1.

The function `inet_add_protocol` is used to add the new protocols. More about this function is discussed in 2.5.2.2.

```
/* Register the socket-side information for inet_create. */
for(r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
    INIT_LIST_HEAD(r);
```

The above code initializes `inetsw`, which is an array of doubly linked lists. `inetsw` is used [fixme]. `SOCK_MAX`[6] is the upper limit on the number of socket types that can be registered with the kernel. It is set to 11 in kernel 2.4.18.

```
for(q=inetsw_array;q<&inetsw_array[INETSW_ARRAY_LEN];++q)
    inet_register_protosw(q);
```

The loop above registers the various socket types, for example : `SOCK_STREAM`, `SOCK_DGRAM` etc. `inetsw_array` is a static array of `struct inet_protosw`

[6] include/asm-i386/socket.h

which is used to define various stuff about the different socket types, including Protocol type, protocol level operations etc. For more on this refer to 2.5.3.5.

```
/*
 *      Set the ARP module up
 */

arp_init();
```

The above function initializes the ARP module. See 2.5.5.

```
/*
 * Set the IP module up
 */

ip_init();
```

Initializes the IP layer module of the subsystem. See 2.5.6.

```
tcp_v4_init(&inet_family_ops);

/* Setup TCP slab cache for open requests. */
tcp_init();
```

The above code calls `tcp_v4_init(..)` and `tcp_init()` to initialize the TCP layer of the networking stack. See 2.5.7 and 2.5.8 respectively for more explanation of the two functions.

```
/*
 *      Set the ICMP layer up
 */

icmp_init(&inet_family_ops);
```

Initializes the ICMP protocol in the subsystem. See 2.5.9.

```
    /* I wish inet_add_protocol had no constructor hook...
       I had to move IPIP from net/ipv4/protocol.c :-( --ANK
     */
#ifdef CONFIG_NET_IPIP
    ipip_init();
#endif
```

If CONFIG_NET_IPIP is defined, then tunneling of IP within IP is initialized by calling the ipip_init() function. See 2.5.10.

```
#ifdef CONFIG_NET_IPGRE
    ipgre_init();
#endif
```

If CONFIG_NET_IPGRE is defined, then GRE(General Routing Encapsulation) support is enabled in the kernel. See 2.5.11.

```
    /*
     *    Initialise the multicast router
     */
#if defined(CONFIG_IP_MROUTE)
    ip_mr_init();
#endif
```

The code above initializes the IP multicast routing in the kernel, provided that CONFIG_IP_MROUTE is defined. See 2.5.12.

```
    /*
     *    Create all the /proc entries.
     */
#ifdef CONFIG_PROC_FS
    proc_net_create ("raw", 0, raw_get_info);
    proc_net_create ("netstat", 0, netstat_get_info);
    proc_net_create ("snmp", 0, snmp_get_info);
    proc_net_create ("sockstat", 0, afinet_get_info);
    proc_net_create ("tcp", 0, tcp_get_info);
    proc_net_create ("udp", 0, udp_get_info);
#endif /* CONFIG_PROC_FS */

    return 0;
    }
```

The above code creates entries in the *Proc* filesystem which can be used to query the subsystem for various kinds of information. The first argument in the function `proc_net_create(..)` specifies the name of the entry, which will appear as a file under */proc/net*, and the 3rd argument is the function which is called whenever the particular entry is queried via */proc*. [fixme:reference to proc fs]

The functions and various structures used in `inet_init()` above are explained in detail below, in the order they are encoutered in the function.

2.5.1 Function sock_register()

File : `net/socket.c`
`Prototype:`

```
int sock_register(struct net_proto_family *ops)
```

This function is used to register a protocol family with the networking subsystem. The argument passed is `ops` which is of `struct net_proto_family` type which basically defines the protocol family and its 'create' function. See 2.5.1.1 for more on `struct net_proto_family`[fixme:register,diag]

```
int err;

if (ops->family >= NPROTO) {
   printk(KERN_CRIT "protocol %d >= NPROTO(%d)\n",
                              ops->family, NPROTO);
   return -ENOBUFS;
}
```

The above code does some basic checks. The `ops->family` cannot be more than `NPROTO`, if it is then an error is printed and the function returns.

```
net_family_write_lock();
```

The above code obtains write lock on `net_family`. This function is defined only for kernels compiled with SMP support, i.e., with `CONFIG_SMP` defined. On uni-processor systems `net_family_write_lock()` is defined as (elsewhere in the same file)

```
#define net_family_write_lock() do { } while(0)
```

The above defined *do-while* loop wont execute at all, so its just a no-op, which does nothing![fixme:ref to explanation of no-op]

```
err = -EEXIST;
if (net_families[ops->family] == NULL) {
    net_families[ops->family]=ops;
    err = 0;
}
```

The above code, first sets the default error to `-EEXIST` which simply indicates that the family type being registered is already registered. In other words its already there in `net_families[ ]` array!

The if construct checks for the condition explained above i.e., the family type being registered doesn't already exist in the array. If it doesn't then, the `ops` is assigned to `net_families[ops->family]` and `err` is set to zero.

```
net_family_write_unlock();
return err;
```

Lastly, the spinlock on `net_family` is released and the function returns with `err` as the return code. `net_family_write_unlock` again is defined as a function only for kernels with SMP support.

2.5.1.1 struct net_proto_family

File : `include/linux/net.h` `Definition :`

```
struct net_proto_family
{
    int    family;
    int    (*create)(struct socket *sock, int protocol);
    short  authentication;
    short  encryption;
    short  encrypt_net;
};
```

This struct is used to define a protocol family in the kernel. Its members are

family Protocol family number/id, for example `PF_INET`[7].

create Pointer to a function for creating a socket of this family.

authentication

[7] `include/linux/socket.h`

encryption

encrypt_net

Example : Function `inet_init()` passes `inet_family_ops` of this type as an argument to `sock_register()`, which defines the member fields as (elsewhere in the same file) :

```
struct net_proto_family inet_family_ops = {
        family: PF_INET,
        create: inet_create
}
```

2.5.2 Adding INET Protocols

[fixme: inet protos required?]

2.5.2.1 struct inet_protocol

File : `include/net/protocol.h` Prototype :

```
struct inet_protocol
{
    int                     (*handler)(struct sk_buff *skb);
    void                    (*err_handler)(struct sk_buff *skb, u32 info);
    struct inet_protocol    *next;
    unsigned char           protocol;
    unsigned char           copy:1;
    void                    *data;
    const char              *name;
};
```

This structure is used for defining the *inet* protocol to be added. Its members are :

handler Pointer to a function that will *receive* the packets of this protocol.

err_handler Pointer to the error handling routine for this protocol.

next Pointer to the next protocol in the INET family.

protocol Protocol, example: IPPROTO_TCP, IPPROTO_UDP etc.

copy:1 [fixme: explained elsewhere]

data [fixme?]

name Name of the protocol, example "TCP", "UDP" etc.

As the `*next` member of the struct indicates, `inet_protocol` acts as a list node. The various protocols are defined by exploiting this member, in the `net/ipv4/protocol.c`. The code in kernel 2.4.18 is quoted here :

```
#define IPPROTO_PREVIOUS NULL

#ifdef CONFIG_IP_MULTICAST

static struct inet_protocol igmp_protocol = {
    handler:                igmp_rcv,
    next:                   IPPROTO_PREVIOUS,
    protocol:               IPPROTO_IGMP,
    name:                   "IGMP"
};

#undef  IPPROTO_PREVIOUS
#define IPPROTO_PREVIOUS &igmp_protocol

#endif
```

`IPPROTO_PREVIOUS` is initialized to null , and the protocols are added taking it as the `next` node. The above definition defines `igmp_rcv` as the protocol `handler` for IGMP, no error handling function and `name` as *"IGMP"*. Then `IPRPTO_PREVIOUS` is re-defined to point to the protocol just added, `&igmp_protocol` here.

IGMP protocol is added only if `CONFIG_IP_MULTICAST` is defined.

```
static struct inet_protocol tcp_protocol = {
    handler:            tcp_v4_rcv,
    err_handler:        tcp_v4_err,
    next:               IPPROTO_PREVIOUS,
    protocol:           IPPROTO_TCP,
    name:               "TCP"
};
```

```
#undef  IPPROTO_PREVIOUS
#define IPPROTO_PREVIOUS &tcp_protocol

static struct inet_protocol udp_protocol = {
    handler:        udp_rcv,
    err_handler:    udp_err,
    next:           IPPROTO_PREVIOUS,
    protocol:       IPPROTO_UDP,
    name:           "UDP"
};

#undef  IPPROTO_PREVIOUS
#define IPPROTO_PREVIOUS &udp_protocol

static struct inet_protocol icmp_protocol = {
    handler:        icmp_rcv,
    next:           IPPROTO_PREVIOUS,
    protocol:       IPPROTO_ICMP,
    name:           "ICMP"
};
```

The three protocols above, namely TCP, UDP and ICMP are defined in the same manner, assigning the `handler`, `err_handler`, `protocol` and the name. The different protocol definitions are chained together in the list by re-defining `IPPROTO_PREVIOUS` to point to the protocol added.

2.5.2.2 Function inet_add_protocol()

File : `net/ipv4/protocol.c`
`Prototype :`

```
void inet_add_protocol(struct inet_protocol *prot)
```

This function adds a protocol of *inet* family. It takes the `prot` as the only argument which defines the protocol to be added.

`inet_protos[]` is an array of `inet_protocol` pointers. It is accessed as a hash table.[fixme?]

```
{
    unsigned char hash;
    struct inet_protocol *p2;
```

```
hash = prot->protocol & (MAX_INET_PROTOS - 1);
br_write_lock_bh(BR_NETPROTO_LOCK);
```

The above code calculates the hash key to be used as the index in `inet_protos[]` array. Basically, bitwise and'ing with `MAX_INET_PROTOS -1` makes sure that the hash is always less than `MAX_INET_PROTOS`.

Then write lock is obtained [fixme: proper line, locks explained..]

```
prot ->next = inet_protos[hash];
inet_protos[hash] = prot;
prot->copy = 0;
```

The above code makes `prot->next` point to the element at `inet_protos[hash]` and then assigns `prot` to the same position in the array. `prot->copy` is initialized to zero.

```
/*
 *     Set the copy bit if we need to.
 */

p2 = (struct inet_protocol *) prot->next;
while (p2) {
    if (p2->protocol == prot->protocol) {
        prot->copy = 1;
        break;
    }
    p2 = (struct inet_protocol *) p2->next;
}
```

The above loop sets `prot->copy` to 1 if the `prot` being added is the same as the entry that was at `inet_proto[hash]` before we updated it by placing ourselves there.[fixme:why copy=1??]

Since, `prot->next` points to the entry at `inet_proto[hash]`, `p2` would point to that. The while loop runs stepping through list pointed to by `p2`, and if `p2->protocol` is equal to `prot->protocol` then the `copy` variable is set to 1 and the loop ends.

```
    br_write_unlock_bh(BR_NETPROTO_LOCK);
}
```

Release the write-lock obtained earlier and return.

2.5.3 Register Socket Interfaces

[fixme:reqd]

2.5.3.1 struct inet_protosw

File : `include/net/protocol.h`
`Definition :`

```
struct inet_protosw {
    struct list_head list;
    unsigned short   type;
    int              protocol;
    struct proto     *prot;
    struct proto_ops *ops;
    int              capability;
    char             no_check;
    unsigned char    flags;
};
```

This structure is used to register socket interfaces for the IP/INET protocols. The member variables are :

list a linked list used for [fixme]

type socket type, for example: `SOCK_STREAM`,`SOCK_DGRAM` etc.[reference]

protocol Protocol type, for example : `IPPROTO_TCP,IPPROTO_UDP` etc. [reference]

prot Socket layer to transport layer operations. See 2.5.3.2.

ops Transport layer to network interface operations. See 2.5.3.3.

capability [fixme]

no_check Checksum on receive/transmit or none

flags Flags, example : `INET_PROTOSW_PERMANENT`[reference]

`flags` can be

INET_PROTOSW_REUSE [8] Ports are automatically re-usable. [fixme: explain?]

[8] `include/net/protocol.h`

INET_PROTOSW_PERMANENT [9] Protocol is permanent and so cannot be removed from the subsystem.

Operations defined by `prot` and `ops` allow the underlying network subsystem in the kernel to remain protocol-neutral. This is explained better in **??**.

2.5.3.2 struct proto

File : `include/net/sock.h`
`Definition :`

This structure defines the socket layer to transport layer operations i.e., the operations that can be performed from a socket's point-of-view. For example, `tcp_prot`, `struct proto` type, defines the socket to *tcp* layer operations.

```
struct proto {
    void         (*close)(struct sock *sk, long timeout);
    int          (*connect)(struct sock *sk,
                            struct sockaddr *uaddr,int addr_len);
    int          (*disconnect)(struct sock *sk, int flags);
    struct sock* (*accept)(struct sock *sk, int flags, int *err);
    int          (*ioctl)(struct sock *sk, int cmd,
                          unsigned long arg);
    int          (*init)(struct sock *sk);
    int          (*destroy)(struct sock *sk);
    void         (*shutdown)(struct sock *sk, int how);
    int          (*setsockopt)(struct sock *sk, int level,
                          int optname, char *optval, int optlen);
    int          (*getsockopt)(struct sock *sk, int level,
                          int optname, char *optval, int *option);
    int          (*sendmsg)(struct sock *sk,struct msghdr *msg, int len);
    int          (*recvmsg)(struct sock *sk, struct msghdr *msg,
                          int len, int noblock, int flags, int *addr_len);
    int          (*bind)(struct sock *sk,
                          struct sockaddr *uaddr, int addr_len);
    int          (*backlog_rcv) (struct sock *sk, struct sk_buff *skb);
    void         (*hash)(struct sock *sk);
    void         (*unhash)(struct sock *sk);
    int          (*get_port)(struct sock *sk, unsigned short snum);
```

[9] `include/net/protocol.h`

```
    char        name[32];
    struct {
        int inuse;
        u8  __pad[SMP_CACHE_BYTES - sizeof(int)];
    } stats[NR_CPUS];
};
```

The struct mostly contains pointers to functions which represent the various operations possible from socket to transport layer. All the operations defined here take `struct sock *sk` as the first argument, this is the socket that these functions act upon. [explanation of the various functions, in approriate layer.]

void (*close)() Function to close the socket.[fixme]

int (*connect)() Function to establish a connection.

int (*disconnect)() Function to disconnect the connection.

struct sock* (*accept)() Function to accept an incoming connection.

int (*ioctl)() Function to ioctl the socket.

int (*init)() Function to create the socket.

int (*destroy)() Function to destroy the socket. [fixme]

void (*shutdown)() Function to shutdown the socket. [fixme]

int (*setsockopt)() Function to set socket options.

int (*getsockopt)() Function to get socket options.

int (*sendmsg)() Function to send/write a message to the socket.

int (*recvmsg)() Function to receive/read a message from the socket.

int (*bind)() Function to bind the socket to a given ip-address and a port.

int (*backlog_rcv)() Function to process the backlog receive buffer.

void (*hash) Function to [fixme]

void (*unhash) Function to [fixme]

int (*get_port) Function to obtain reference to a local port for the given socket.

char name[32]Name of the transport layer protocol which these operations define.

struct stats[ ] [fixme]

The actual functions which are assigned to the pointers above are explained in the respective transport layer sections.[fixme: reference] Taking the same example of `tcp_prot`, the `(*close)()` maps to `tcp_close()` function. Same way the other function pointers map to their respective *tcp* layer functions.

2.5.3.3 struct proto_ops

File : `include/linux/net.h`
Definition :

```
struct proto_ops {
    int family;
    int (*release)(struct socket *sock);
    int (*bind)(struct socket *sock, struct sockaddr *umyaddr,
                         int sockaddr_len);
    int (*connect)(struct socket *sock, struct sockaddr *uservaddr,
                         int sockaddr_len, int flags);
    int (*socketpair)(struct socket *sock1, struct socket *sock2);
    int (*accept)(struct socket *sock, struct socket *newsock,
                         int flags);
    int (*getname)(struct socket *sock, struct sockaddr *uaddr,
                         int *usockaddr_len, int peer);
    unsigned int (*poll)(struct file *file, struct socket *sock,
                         struct poll_table_struct *wait);
    int (*ioctl)(struct socket *sock, unsigned int cmd,
                         unsigned long arg);
    int (*listen)(struct socket *sock, int len);
    int (*shutdown)(struct socket *sock, int flags);
    int (*setsockopt)(struct socket *sock, int level, int optname,
                         char *optval, int optlen);
    int (*getsockopt)(struct socket *sock, int level, int optname,
                         char *optval, int *optlen);
    int (*sendmsg)(struct socket *sock, struct msghdr *m,
                         int total_len, struct scm_cookie *scm);
    int (*recvmsg)(struct socket *sock, struct msghdr *m, int total_len,
```

```
                         int flags, struct scm_cookie *scm);
    int (*mmap)(struct file *file, struct socket *sock,
                         struct vm_area_struct * vma);
    ssize_t (*sendpage)(struct socket *sock, struct page *page,
                         int offset, size_t size, int flags);
};
```

This struct is used to define the transport layer to network layer operations. Except for `family` all other members are function pointers representing the operations. These operations take `struct socket *sock` as their first parameter. `struct socket` is used by the user-space processes to refer to a socket.[fixme:diag?]

int family Protocol Family, example `PF_INET` etc.

int (*release)() Function to release the socket

int (*bind)() Function to bind the socket to the specified ip-address and port.

int (*connect)() Function to establish a connection.

int (*socketpair)() Function to create a pair of unnamed sockets.[fixme]

int (*accept)() Function to accept an incoming connection request.

int (*getname)() [fixme]

unsigned int (*poll)() [fixme]

int (*ioctl)() Function to ioctl the socket.

int (*listen)() Function to listen on a socket.

int (*shutdown)() Function to cause part or all of a full-duplex connection to shutdown.

int (*setsockopt)() Function to set socket options.

int (*getsockopt)() Function to get socket options.

int (*sendmsg)() Function to send/write a message to the socket.

int (*recvmsg)() Function to receive/read a message from the socket.

int (*mmap)() [fixme]

ssize_t (*sendpage)() [fixme]

The actual functions which are assigned to the pointers above are explained in the INET layer chapter .[fixme: reference].

Example : `inet_stream_ops`, `inet_dgram_ops` etc.

2.5.3.4 Array inetsw_array

File : `net/ipv4/af_inet.c`

Definition :

```
/* Upon startup we insert all the elements in inetsw_array[] into
 * the linked list inetsw.
 */
static struct inet_protosw inetsw_array[] =
```

`inetsw_array[]` is a static array of `struct inet_protosw`, the sole purpose of which is to define the various socket types and their related operations to be added to the subsystem at startup. This is done in the function `inet_init()`, 2.5.

In kernel 2.4.18, this array has 3 entries. Their definitions are quoted below:

```
{
    {
       type:         SOCK_STREAM,
       protocol:     IPPROTO_TCP,
       prot:         &tcp_prot,
       ops:          &inet_stream_ops,
       capability:   -1,
       no_check:     0,
       flags:        INET_PROTOSW_PERMANENT,
    },

    {
        type:         SOCK_DGRAM,
        protocol:     IPPROTO_UDP,
        prot:         &udp_prot,
        ops:          &inet_dgram_ops,
        capability:   -1,
```

```
        no_check:    UDP_CSUM_DEFAULT,
        flags:       INET_PROTOSW_PERMANENT,
    },


    {
        type:        SOCK_RAW,
        protocol:    IPPROTO_IP,/* wild card */
        prot:        &raw_prot,
        ops:         &inet_dgram_ops,
        capability:  CAP_NET_RAW,
        no_check:    UDP_CSUM_DEFAULT,
        flags:       INET_PROTOSW_REUSE,
    }
};
```

[fixme: no_check,capability] Taking the first one as an example, the various fields are defined for socket of `type` `SOCK_STREAM`. `tcp_prot` defines the various functions as :

```
struct proto tcp_prot = {
    name:           "TCP",
    close:          tcp_close,
    connect:        tcp_v4_connect,
    disconnect:     tcp_disconnect,
    accept:         tcp_accept,
    ioctl:          tcp_ioctl,
    init:           tcp_v4_init_sock,
    destroy:        tcp_v4_destroy_sock,
    shutdown:       tcp_shutdown,
    setsockopt:     tcp_setsockopt,
    getsockopt:     tcp_getsockopt,
    sendmsg:        tcp_sendmsg,
    recvmsg:        tcp_recvmsg,
    backlog_rcv:    tcp_v4_do_rcv,
    hash:           tcp_v4_hash,
    unhash:         tcp_unhash,
    get_port:       tcp_v4_get_port,
};
```

The above is defined in net/ipv4/tcp_ipv4.c. The various function pointers are being set to the corresponding functions for the tcp layer. For example, `close` is set to `tcp_close`.

And the `ops` is set to `inet_stream_ops` which is defined in net/ipv4/af_inet.c. The definition :

```
struct proto_ops inet_stream_ops = {
    family:          PF_INET,

    release:         inet_release,
    bind:            inet_bind,
    connect:         net_stream_connect,
    socketpair:      sock_no_socketpair,
    accept:          inet_accept,
    getname:         inet_getname,
    poll:            tcp_poll,
    ioctl:           inet_ioctl,
    listen:          inet_listen,
    shutdown:        inet_shutdown,
    setsockopt:      inet_setsockopt,
    getsockopt:      inet_getsockopt,
    sendmsg:         inet_sendmsg,
    recvmsg:         inet_recvmsg,
    mmap:            sock_no_mmap,
    sendpage:        tcp_sendpage
};
```

The operations for the other two socket types, SOCKET_DGRAM and SOCKET_RAW are defined in the same manner. The corresponding structs are :

SOCKET_DGRAM

udp_prot Defined in net/ipv4/udp.c.

inet_dgram_ops Defined in net/ipv4/af_inet.c.

SOCKET_RAW

raw_prot Defined in `net/ipv4/raw.c`.

inet_dgram_ops Defined in `net/ipv4/af_inet.c`.

2.5.3.5 Function inet_register_protosw(..)

2.5.4 Initializing Protocols

Now that various protocol families, socket types have been registered, its time to register the protocols and packet types. When a packet is received from the network, it must be passed on to the network layer handler corresponding to the packet type of the incoming packet. For example, an incoming IP packet must be passed on to ip-packet handler, `ip_rcv(..)`.

Each Network Layer[fixme:correct?] protocol is registered in a `struct packet_type` which is held by a `ptype_all` list or a `ptype_base` hash table. `ptype_all` and `ptype_base` hold respectively, handlers for generic packets and for specific packets. The protocols are added or registered using the `dev_add_pack(..)` function (2.5.4.2).

`struct packet_type`, desribed in 2.5.4.1, holds information about the protocol type and the corresponding receive routine for processing the incoming packets. The relevant receive routine is called from the function `net_rx_action(..)` [fixme:any others?] by matching the protocol types of the incoming packets with one or more of the `packet_type` structures held in `ptype_all` and `ptype_base`.

2.5.4.1 struct packet_type

File : `include/linux/netdevice.h`
`Definition :`

```
struct packet_type
{
    unsigned short     type;
    struct net_device  *dev;
    int                (*func)(struct sk_buff *, struct net_device *,
                               struct packet_type *);
    void               *data;
```

```
    struct packet_type *next;
};
```

[fixme: only network layer protos?] This structure is used to describe a packet for registering with the subsystem. The members are:

unsigned short type Ethernet ID for the packet, for example `ETH_P_IP` is for IP protocol, `ETH_P_ARP` for ARP(Address Resolution Protocol). Defined in `include/linux/if_ether.h`.

struct net_device *dev The network device for this protocol. NULL means "all devices".

int (*func)() Pointer to the function which will receive and process the incoming packets.

void *data [fixme:??]

struct packet_type *next Pointer to the next `packet_type` structure.

2.5.4.2 Function dev_add_pack()

File : `net/core/dev.c`
`Prototype :`

```
/**
 * dev_add_pack - add packet handler
 * @pt: packet type declaration
 *
 * Add a protocol handler to the networking stack. The passed &packet_type
 * is linked into kernel lists and may not be freed until it has been
 * removed from the kernel lists.
 */

void dev_add_pack(struct packet_type *pt)
```

This function is used to add a protocol to the networking stack. The argument `pt` passed defines the protocol for which the packet handler has to be added.

```
{
    int hash;

    br_write_lock_bh(BR_NETPROTO_LOCK);
```

The above code obtains write lock on `BR_NETPROTO_LOCK`.[fixme]

```
#ifdef CONFIG_NET_FASTROUTE
    /* Hack to detect packet socket */
    if ((pt->data) && ((int)(pt->data)!=1)) {
        netdev_fastroute_obstacles++;
        dev_clear_fastroute(pt->dev);
    }
#endif
```

The if-construct is compiled only if `CONFIG_NET_FASTROUTE` is defined. `CONFIG_NET_FASTROUTE` enables direct NIC-to-NIC data transfers. The above loop [fixme:]

```
    if (pt->type == htons(ETH_P_ALL)) {
        netdev_nit++;
        pt->next=ptype_all;
        ptype_all=pt;
```

`ETH_P_ALL`[10] means "all packets", so if this is packet type then `pt` is added to the `ptype_all` list, which as mentioned earlier holds packet handlers for generic packets. `netdev_nit` is [fixme:network interface taps???]

```
    } else {
        hash=ntohs(pt->type)&15;
        pt->next = ptype_base[hash];
        ptype_base[hash] = pt;
    }
    br_write_unlock_bh(BR_NETPROTO_LOCK);
}
```

If the packet type is anything other than `ETH_P_ALL` then add it to the `ptype_base` hash table. The new protocol `pt` is added at the beginning of the list at `ptype_base[hash]`.[fixme: hash calc- discuss required?]

[10] `include/linux/if_ether.h`

2.5.5 Function arp_init()

File : net/ipv4/arp.c
`Prototype :`

```
void __init arp_init (void)
```

This function initializes the ARP(Address Resolution Protocol) layer.

```
{
    neigh_table_init(&arp_tbl);
```

The above function call initializes the neighbour tables. Refer to **??**

```
    dev_add_pack(&arp_packet_type);
```

The above function call adds the ARP packet handler to the subsystem. Refer to 2.5.5.1 for more on `arp_packet_type`.

```
    proc_net_create ("arp", 0, arp_get_info);
```

Create the entry *arp* under the proc filesystem. `proc_net_create` is a wrapper function for creating a proc entry under the [fixme: path]. It is defined in include/linux/proc_fs.h.

```
#ifdef CONFIG_SYSCTL
    neigh_sysctl_register(NULL, &arp_tbl.parms, NET_IPV4, NET_IPV4_NEIGH, "ipv4");
#endif

}
```

Refer to **??** for explanation of the above function.

2.5.5.1 arp_packet_type

File : net/ipv4/arp.c
`Definition :`

```
static struct packet_type arp_packet_type = {
    type:   __constant_htons(ETH_P_ARP),
    func:   arp_rcv,
    data:   (void*) 1, /* understand shared skbs */
};
```

This defines the packet handler for ARP packets. The packet `type` is `ETH_P_ARP`[11], the packet handling/receiving function is `arp_rcv` and `data` [fixme]. Refer to 2.5.4.1 for more on `struct packet_type`.

2.5.5.2 Function neigh_table_init(..)

File : `net/core/neighbour.c`

2.5.5.3 Function neigh_sysctl_register(..)

File : `net/core/neighbour.c`

2.5.6 Function ip_init()

File : `net/ipv4/ip_output.c`
`Prototype :`

```
/*
 * IP registers the packet type and then calls the subprotocol initialisers
 */

void __init ip_init(void)
```

This function IP protocol is registered. Basically, the IP packet's handler is registered and [fixme]

```
{
    dev_add_pack(&ip_packet_type);
```

Add packet handler for IP packets. See 2.5.6.1 for `ip_packet_type`.

```
    ip_rt_init();
    inet_initpeers();
```

Initialize routing tables and [fixme: better descr]. See 2.5.6.2 and 2.5.6.3.

```
#ifdef CONFIG_IP_MULTICAST
    proc_net_create("igmp", 0, ip_mc_procinfo);
#endif
}
```

The above code creates a proc entry for `igmp` if `CONFIG_IP_MULTICAST` is defined.[fixme: more in network layer]

[11] `include/linux/if_ether.h`

2.5.6.1 ip_packet_type

File : net/ipv4/ip_output.c
Definition :

```
/*
 *      IP protocol layer initialiser
 */

static struct packet_type ip_packet_type =
{
    __constant_htons(ETH_P_IP),
    NULL,   /* All devices */
    ip_rcv,
    (void*)1,
    NULL,
};
```

ip_packet_type defines the IP packet handler. Packet type is `ETH_P_IP`[12] i.e., IP packet, ip-packet handling/receiving routine is `ip_rcv` and [fixme:data].

2.5.6.2 Function ip_rt_init()

File : net/ipv4/route.c
Prototype :

```
void __init ip_rt_init(void)
{
    int i, order, goal;

#ifdef CONFIG_NET_CLS_ROUTE
    for (order = 0;
        (PAGE_SIZE << order) < 256 * sizeof(ip_rt_acct) * NR_CPUS; order++)
        /* NOTHING */;
    ip_rt_acct = (struct ip_rt_acct *)__get_free_pages(GFP_KERNEL, order);
    if (!ip_rt_acct)
        panic("IP: failed to allocate ip_rt_acct\n");
    memset(ip_rt_acct, 0, PAGE_SIZE << order);
```

[12] include/linux/if_ether.h

```
#endif

    ipv4_dst_ops.kmem_cachep = kmem_cache_create("ip_dst_cache",
                                sizeof(struct rtable),
                                0, SLAB_HWCACHE_ALIGN,
                                NULL, NULL);

    if (!ipv4_dst_ops.kmem_cachep)
        panic("IP: failed to allocate ip_dst_cache\n");

    goal = num_physpages >> (26 - PAGE_SHIFT);

    for (order = 0; (1UL << order) < goal; order++)
        /* NOTHING */;

    do {
        rt_hash_mask = (1UL << order) * PAGE_SIZE /
                sizeof(struct rt_hash_bucket);
        while (rt_hash_mask & (rt_hash_mask - 1))
                rt_hash_mask--;
        rt_hash_table = (struct rt_hash_bucket *)
                __get_free_pages(GFP_ATOMIC, order);
    } while (rt_hash_table == NULL && --order > 0);

    if (!rt_hash_table)
        panic("Failed to allocate IP route cache hash table\n");

    printk("IP: routing cache hash table of %u buckets, %ldKbytes\n",
       rt_hash_mask,
       (long) (rt_hash_mask * sizeof(struct rt_hash_bucket)) / 1024);

    for (rt_hash_log = 0; (1 << rt_hash_log) != rt_hash_mask; rt_hash_log++)
        /* NOTHING */;

    rt_hash_mask--;
    for (i = 0; i <= rt_hash_mask; i++) {
        rt_hash_table[i].lock = RW_LOCK_UNLOCKED;
        rt_hash_table[i].chain = NULL;
    }

    ipv4_dst_ops.gc_thresh = (rt_hash_mask + 1);
```

```
    ip_rt_max_size = (rt_hash_mask + 1) * 16;

    devinet_init();
    ip_fib_init();

    rt_flush_timer.function = rt_run_flush;
    rt_periodic_timer.function = rt_check_expire;

    /* 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);

    proc_net_create ("rt_cache", 0, rt_cache_get_info);
    proc_net_create ("rt_cache_stat", 0, rt_cache_stat_get_info);
#ifdef CONFIG_NET_CLS_ROUTE
    create_proc_read_entry("net/rt_acct", 0, 0, ip_rt_acct_read, NULL);
#endif
}
```

2.5.6.3 Function inet_initpeers()

File : net/ipv4/inetpeer.c
Prototype :

```
/* Called from ip_output.c:ip_init  */
void __init inet_initpeers(void)
{
    struct sysinfo si;

    /* Use the straight 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 = kmem_cache_create("inet_peer_cache",
                sizeof(struct inet_peer),
                0, SLAB_HWCACHE_ALIGN,
                NULL, NULL);

    /* 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);
}
```

2.5.7 Function tcp_v4_init(..)

File : `net/ipv4/tcp_ipv4.c`
`Prototype :`

```
void __init tcp_v4_init(struct net_proto_family *ops)
```

This function sets up the *tcp control socket*, used to send RST(reset the connection) segments.

A *reset* is sent by TCP whenever a segment arrives that is for a non-existant socket. Since we need a socket to send a packet out, the *tcp control socket* is used for sending the RST's. This is explained in detail in the chapter on TCP.[fixme:ref]

```
{
    int err;

    tcp_inode.i_mode = S_IFSOCK;
    tcp_inode.i_sock = 1;
    tcp_inode.i_uid = 0;
    tcp_inode.i_gid = 0;
```

`tcp_inode` defines the `inode` for the control socket. Every socket has an `inode` associated with it. This concept will be explained in the chapter on BSD sockets.

`tcp_inode.i_mode` is set to `S_IFSOCK`[13], meaning that this inode represents a *socket*.
`tcp_inode.i_sock` is initialized to 1, [fixme:?]
`tcp_inode.i_uid` and `tcp_inode.i_gid` are set to 0, which means that the control socket is owned by the superuser - root!

```
init_waitqueue_head(&tcp_inode.i_wait);
init_waitqueue_head(&tcp_inode.u.socket_i.wait);
```

Initialize the two wait-queues namely, `tcp_inode.i_wait` and `tcp_inode.u.socket_i.wait`.

```
tcp_socket->inode = &tcp_inode;
tcp_socket->state = SS_UNCONNECTED;
tcp_socket->type=SOCK_RAW;
```

The above code sets the `tcp_socket`'s inode to `tcp_inode`, sets its state to `SS_UNCONNECTED`[14] and its type to `SOCK_RAW`. The control socket is not used for establishing any actual connections so it stays in `SS_UNCONNECTED` state. Its `type` is `SOCK_RAW` so that it can [fixme: send rst's??]

```
if ((err=ops->create(tcp_socket, IPPROTO_TCP))<0)
    panic("Failed to create the TCP control socket.\n");
```

The above code creates the tcp control socket by calling `ops->create(..)` which is a function pointer to `inet_create(..)`.
If there was an error in creating the control socket, the kernel panics via `panic(..)` since the control socket is essential for the networking subsytem.

```
tcp_socket->sk->allocation=GFP_ATOMIC;
tcp_socket->sk->protinfo.af_inet.ttl = MAXTTL;
```

Set the `tcp_socket->sk`'s (`struct sock`) memory allocation mode to `GFP_ATOMIC`. Refer to the *Memory Management* document for more on this.[fixme:url?].
Set the `ttl`(Time To Live) of the socket to `MAXTTL`[15].

---

[13] include/linux/stat.h
[14] include/linux/net.h
[15] include/linux/ip.h

```
    /* Unhash it so that IP input processing does not even
     * see it, we do not wish this socket to see incoming
     * packets.
     */
    tcp_socket->sk->prot->unhash(tcp_socket->sk);
}
```

The above code unhashes the socket so that it does not receive any incoming packets, since the control doesn't need to receive any incoming packets.

2.5.8 Function tcp_init()

File : net/ipv4/tcp.c
Prototype :

```
void __init tcp_init(void)
```

This function initializes the memory caches, allocates the hash tables and sets the tunable paramaters (sysctl_*).

```
{
    struct sk_buff *skb = NULL;
    unsigned long goal;
    int order, i;

    if(sizeof(struct tcp_skb_cb) > sizeof(skb->cb))
        __skb_cb_too_small_for_tcp(sizeof(struct tcp_skb_cb),
                      sizeof(skb->cb));
```

[fixme]

```
    tcp_openreq_cachep = kmem_cache_create("tcp_open_request",
                            sizeof(struct open_request),
                            0, SLAB_HWCACHE_ALIGN,
                            NULL, NULL);
    if(!tcp_openreq_cachep)
        panic("tcp_init: Cannot alloc open_request cache.");
```

The above code allocates the memory cache for entities of type struct tcp_open_request.[fixme:open requests??] If it could not be allocated then the kernel panics via the function panic(..).

```
tcp_bucket_cachep = kmem_cache_create("tcp_bind_bucket",
                            sizeof(struct tcp_bind_bucket),
                            0, SLAB_HWCACHE_ALIGN,
                            NULL, NULL);
if(!tcp_bucket_cachep)
    panic("tcp_init: Cannot alloc tcp_bind_bucket cache.");
```

The code above allocated the memory cache for entities of type `struct tcp_bind_bucket`.[fixme]. If it could not be allocated then the kernel panics.

```
tcp_timewait_cachep = kmem_cache_create("tcp_tw_bucket",
                            sizeof(struct tcp_tw_bucket),
                            0, SLAB_HWCACHE_ALIGN,
                            NULL, NULL);
if(!tcp_timewait_cachep)
    panic("tcp_init: Cannot alloc tcp_tw_bucket cache.");
```

The above block of code allocate a memory cache for entities of type `tcp_tw_bucket`. It represents sockets in *TIME_WAIT* state.[fixme?] The kernel panics if the allocation was unsuccessful.

The next part of the function allocates memory for the two hash tables, namely `tcp_ehash` and `tcp_bhash`. They are pointers to `struct tcp_ehash_bucket` and `tcp_bind_hashbucket` respectively, both defined in `include/net/tcp.h`.

Here, the page-oriented technique of memory allocation is used, which is done via the function `__get_free_pages(..)`. This function takes two arguments, second of which is `order`. *order* is a block of pages allocated as a power of 2. When order is zero - 2^0 (1) page is allocated, when order is one - 2^1 (2) pages are allocated, order 2 - 2^2 (4) pages and so on.

So, *1 << order* is the number of pages that `__get_free_pages(GFP_ATOMIC, order)` will allocate. See **??** for more on this function.

```
/* Size and allocate the main established and bind bucket
 * hash tables.
 *
 * The methodology is similar to that of the buffer cache.
 */
if (num_physpages >= (128 * 1024))
```

```
        goal = num_physpages >> (21 - PAGE_SHIFT);
    else
        goal = num_physpages >> (23 - PAGE_SHIFT);
```

[fixme]

```
    for(order = 0; (1UL << order) < goal; order++)
        ;
```

In the above for loop we try to find the minimum value of order that would fit `goal` number of pages in it. We keep increasing the value of order till the block of pages (2^order or 1UL<<order) would fit `goal` into it.

The do-while loop below tries to allocate memory for the `tcp_ehash` hash table for sockets in the *ESTABLISHED* state. We keep dropping the value of `order` till we can get a block of pages allocated.

```
    do {
        tcp_ehash_size = (1UL << order) * PAGE_SIZE /
            sizeof(struct tcp_ehash_bucket);
        tcp_ehash_size >>= 1;

        while (tcp_ehash_size & (tcp_ehash_size-1))
            tcp_ehash_size--;
        tcp_ehash = (struct tcp_ehash_bucket *)
            __get_free_pages(GFP_ATOMIC, order);
    } while (tcp_ehash == NULL && --order > 0);


    if (!tcp_ehash)
        panic("Failed to allocate TCP established hash table\n");
```

Panic if memory for the hash table could not be allocated.

```
    for (i = 0; i < (tcp_ehash_size<<1); i++) {
        tcp_ehash[i].lock = RW_LOCK_UNLOCKED;
        tcp_ehash[i].chain = NULL;
    }


    do {
        tcp_bhash_size = (1UL << order) * PAGE_SIZE /
```

```
            sizeof(struct tcp_bind_hashbucket);
        if ((tcp_bhash_size > (64 * 1024)) && order > 0)
            continue;
        tcp_bhash = (struct tcp_bind_hashbucket *)
            __get_free_pages(GFP_ATOMIC, order);
    } while (tcp_bhash == NULL && --order >= 0);


    if (!tcp_bhash)
        panic("Failed to allocate TCP bind hash table\n");

    for (i = 0; i < tcp_bhash_size; i++) {
        tcp_bhash[i].lock = SPIN_LOCK_UNLOCKED;
        tcp_bhash[i].chain = NULL;
    }


    /* Try to be a bit smarter and adjust defaults depending
     * on available memory.
     */
    if (order > 4) {
        sysctl_local_port_range[0] = 32768;
        sysctl_local_port_range[1] = 61000;
        sysctl_tcp_max_tw_buckets = 180000;
        sysctl_tcp_max_orphans = 4096<<(order-4);
        sysctl_max_syn_backlog = 1024;
    } else if (order < 3) {
        sysctl_local_port_range[0] = 1024*(3-order);
        sysctl_tcp_max_tw_buckets >>= (3-order);
        sysctl_tcp_max_orphans >>= (3-order);
        sysctl_max_syn_backlog = 128;
    }
    tcp_port_rover = sysctl_local_port_range[0] - 1;

    sysctl_tcp_mem[0] = 768<<order;
    sysctl_tcp_mem[1] = 1024<<order;
    sysctl_tcp_mem[2] = 1536<<order;
    if (sysctl_tcp_mem[2] - sysctl_tcp_mem[1] > 512)
        sysctl_tcp_mem[1] = sysctl_tcp_mem[2] - 512;
    if (sysctl_tcp_mem[1] - sysctl_tcp_mem[0] > 512)
        sysctl_tcp_mem[0] = sysctl_tcp_mem[1] - 512;
```

```
    if (order < 3) {
        sysctl_tcp_wmem[2] = 64*1024;
        sysctl_tcp_rmem[0] = PAGE_SIZE;
        sysctl_tcp_rmem[1] = 43689;
        sysctl_tcp_rmem[2] = 2*43689;
    }
```

For explanation of all the *sysctl* options see **??**.

```
    printk("TCP: Hash tables configured (established %d bind %d)\n",
                                  tcp_ehash_size<<1, tcp_bhash_size);

    tcpdiag_init();
}
```

2.5.8.1 Function tcpdiag_init()

File : `net/ipv4/tcp_diag.c`
`Prototype :`

```
 void __init tcpdiag_init(void)
```

This function sets up a *netlink* socket for monitoring tcp sockets.[fixme:better explanation?]

```
{
    tcpnl = netlink_kernel_create(NETLINK_TCPDIAG, tcpdiag_rcv);
```

`tcpnl`, defined in the same file, is of type `struct sock` and is initialized to a netlink socket created by calling the function `netlink_kernel_create(..)`. `NETLINK_TCPDIAG`[16] is [fixme:] `tcpdiag_rcv` is a function defined in the same file, [fixme]

```
    if (tcpnl == NULL)
        panic("tcpdiag_init: Cannot create netlink socket.");
}
```

The kernel panics if the `tcpnl`, netlink socket could not be created!

[16] `include/linux/netlink.h`

2.5.9 Function icmp_init(..)

File : `net/ipv4/icmp.c` Prototype :

```
void __init icmp_init(struct net_proto_family *ops)
```

This function creates a control socket for *icmp* protocol which is used for icmp replies.[fixme:?? more] The actual code of this function is nearly the same as of that for `tcp_v4_init(..)`,2.5.7.

```
{
    int err;

    icmp_inode.i_mode = S_IFSOCK;
    icmp_inode.i_sock = 1;
    icmp_inode.i_uid = 0;
    icmp_inode.i_gid = 0;
```

`icmp_inode` defines the `inode` for the control socket. Every socket has an `inode` associated with it. This concept will be explained in the chapter on BSD sockets.

`icmp_inode.i_mode` is set to `S_IFSOCK`[17], meaning that this inode represents a *socket*.
`icmp_inode.i_sock` is initialized to 1, [fixme:?]
`icmp_inode.i_uid` and `icmp_inode.i_gid` are set to 0, which means that the control socket is owned by the superuser - root!

```
    init_waitqueue_head(&icmp_inode.i_wait);
    init_waitqueue_head(&icmp_inode.u.socket_i.wait);
```

Initialize the two wait queues, namely `icmp_inode.i_wait` and `icmp_mode.u.socket_i.wait`.

```
    icmp_socket->inode = &icmp_inode;
    icmp_socket->state = SS_UNCONNECTED;
    icmp_socket->type=SOCK_RAW;
```

The above code sets the `icmp_socket`'s inode to `tcp_inode`, sets its state to `SS_UNCONNECTED`[18] and its type to `SOCK_RAW`. The control socket is not used for establishing any actual connections so it stays in `SS_UNCONNECTED` state. Its `type` is `SOCK_RAW` so that it can [fixme:??]

[17] `include/linux/stat.h`
[18] `include/linux/net.h`

```
    if ((err=ops->create(icmp_socket, IPPROTO_ICMP))<0)
        panic("Failed to create the ICMP control socket.\n");
```

The above code creates the icmp control socket by calling `ops->create(..)` which is a function pointer to `inet_create(..)`.
If there was an error in creating the control socket, the kernel panics via `panic(..)` since the control socket is essential for the networking subsytem.

```
    icmp_socket->sk->allocation=GFP_ATOMIC;
    icmp_socket->sk->sndbuf = SK_WMEM_MAX*2;
    icmp_socket->sk->protinfo.af_inet.ttl = MAXTTL;
    icmp_socket->sk->protinfo.af_inet.pmtudisc = IP_PMTUDISC_DONT;
```

`icmp_socket->sk` is of type `struct sock`, which will be discussed in a later chapter.

Set the `icmp_socket->sk`'s (`struct sock`) memory allocation mode to `GFP_ATOMIC`. Refer to the *Memory Management* document for more on this.[fixme:url?].
The *send buffer*, `icmp_socket->sk->sndbuf`, is set to double of `SK_WMEM_MAX`[19]. [fixme:why?]
Set the `ttl`(Time To Live) of the socket to `MAXTTL`[20].
The last line of the code above, sets the value for *IP MTU Discovery*. `IP_PMTUDISCOVERY_DONT`[21] means *"never send DF(dont fragment) frames"*. This will become clearer after discussion about the network layer.

```
    /* Unhash it so that IP input processing does not even
     * see it, we do not wish this socket to see incoming
     * packets.
     */
    icmp_socket->sk->prot->unhash(icmp_socket->sk);
}
```

Since `icmp_socket` is control socket, it doesnt need to receive any incoming packets, so we unhash it(remove it from the table of sockets).

[19] include/linux/skbuff.h
[20] include/linux/ip.h
[21] include/linux/in.h

2.5.10 Function ipip_init()

File : net/ipv4/ipip.c
Prototype :

```
int __init ipip_init(void)
{
    printk(banner);

    ipip_fb_tunnel_dev.priv = (void*)&ipip_fb_tunnel;
    register_netdev(&ipip_fb_tunnel_dev);
    inet_add_protocol(&ipip_protocol);
    return 0;
}
```

2.5.11 Function ipgre_init()

File : net/ipv4/ipgre.c
Prototype :

```
/*
 *      And now the modules code and kernel interface.
 */

#ifdef MODULE
int init_module(void)
#else
int __init ipgre_init(void)
#endif
{
    printk(KERN_INFO "GRE over IPv4 tunneling driver\n");

    ipgre_fb_tunnel_dev.priv = (void*)&ipgre_fb_tunnel;
    register_netdev(&ipgre_fb_tunnel_dev);
    inet_add_protocol(&ipgre_protocol);
    return 0;
}
```

2.5.12 Function ip_mr_init()

File : net/ipv4/ipmr.c
Prototype :

```
/*
 *      Setup for IP multicast routing
 */

void __init ip_mr_init(void)
{
    printk(KERN_INFO "Linux IP multicast router 0.06 plus PIM-SM\n");
    mrt_cachep = kmem_cache_create("ip_mrt_cache",
                                   sizeof(struct mfc_cache),
                                   0, SLAB_HWCACHE_ALIGN,
                                   NULL, NULL);
    init_timer(&ipmr_expire_timer);
    ipmr_expire_timer.function=ipmr_expire_process;
    register_netdevice_notifier(&ip_mr_notifier);
#ifdef CONFIG_PROC_FS
    proc_net_create("ip_mr_vif",0,ipmr_vif_info);
    proc_net_create("ip_mr_cache",0,ipmr_mfc_info);
#endif
}
```

2.6 Function af_unix_init()

File : net/unix/af_unix.c
Prototype :

```
static int __init af_unix_init(void)
```

This function initializes the BSD Unix domain sockets. The Unix domain protocols are not an actual protocol suite, but a way of performing client-server communication on a single host using the same API that is used for clients and servers on different hosts : sockets.

```
{
    struct sk_buff *dummy_skb;

    printk(banner);
```

The above `printk()` function call prints the `banner` which is defined in the same file as :

```
static char banner[] __initdata =
  KERN_INFO "NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.\n";
```

This banner is visible when the kernel boots and can also be seen in *dmesg* output.

```
    if (sizeof(struct unix_skb_parms) > sizeof(dummy_skb->cb))
    {
        printk(KERN_CRIT "unix_proto_init: panic\n");
        return -1;
    }
```

The above code tries to make sure that the paramters for the unix domain sockets, `struct unix_skb_parms`, will fit in the control buffer of a skb, `struct sk_buff`.

```
    sock_register(&unix_family_ops);
```

The above function call registers the unix domain sockets family with the subsystem. Refer to 2.5.1 for more on `sock_register(..)`. `unix_family_ops` is of `struct net_family_ops` type, defined in the same file as :

```
    struct net_proto_family unix_family_ops = {
        family:         PF_UNIX,
        create:         unix_create
    };
```

It defines the socket family as `PF_UNIX` and the function for creating a socket of this type as `unix_create(..)`. See 2.5.1.1 for more on `net_proto_family`.

```
#ifdef CONFIG_PROC_FS
    create_proc_read_entry("net/unix", 0, 0, unix_read_proc, NULL);
#endif
```

If `CONFIG_PROC_FS` is defined, then the *"net/unix"* proc entry is created, and sets the *read* function for this entry to `unix_read_proc`.

```
    unix_sysctl_register();
```

Registers the *sysctl* interface for the unix domain sockets. This will be explained in the relevant chapter.[fixme:ref]

```
    return 0;
}
```

2.7 Function packet_init()

File : `net/packet/af_packet.c`
`Prototype :`

```
static int __init packet_init(void)
```

This function initializes the raw socket interface. Raw sockets allow new IPv4 protocols to be implemented in user space. A raw socket receives or sends the raw datagram not including link level headers. More on raw sockets is discussed in the relevant chapter. [fixme: ref]

```
{
    sock_register(&packet_family_ops);
```

The above function call registers the `PF_PACKET`[22] protocol family used solely for raw sockets. `packet_family_ops` is defined in the same file as :

```
    static struct net_proto_family packet_family_ops = {
        family:         PF_PACKET,
        create:         packet_create,
    };
```

It defines the protocol family as `PF_PACKET` and the function for creating this type of a socket is set to `packet_create(..)`. See 2.5.1.1 for `net_proto_family`.

```
    register_netdevice_notifier(&packet_netdev_notifier);
```

[22] `include/linux/socket.h`

The above function call registers network notifier block. [fixme:better n more]

`packet_netdev_notifier`, of type `struct notifier_block`, is defined in the same file as :

```
static struct notifier_block packet_netdev_notifier = {
    notifier_call:  packet_notifier,
};
```

[fixme:]

```
#ifdef CONFIG_PROC_FS
    create_proc_read_entry("net/packet", 0, 0, packet_read_proc, NULL);
#endif
```

Create a proc entry *"net/packet"* and sets the function called when this entry is 'read' to the function `packet_read_proc`.

```
    return 0;
}
```

2.7.0.1 Function register_netdevice_notifier(..)

File : `net/core/dev.c` `Prototype :`

```
    /**
     *  register_netdevice_notifier - register a network notifier block
     *  @nb: notifier
     *
     *  Register a notifier to be called when network device events occur.
     *  The notifier passed is linked into the kernel structures and must
     *  not be reused until it has been unregistered. A negative errno code
     *  is returned on a failure.
     */

int register_netdevice_notifier(struct notifier_block *nb)
{
    return notifier_chain_register(&netdev_chain, nb);
}
```

2.7.0.2 struct notifier_block

Chapter 3

BSD Sockets

3.1 Overview

Chapter 4

INET Sockets

4.1 Overview

Chapter 5

Transport Layer

5.1 Overview

Chapter 6

Network Layer

6.1 Overview

Chapter 7

Syscall Trace

7.1 Overview

7.2 Socket structres

7.3 Syscalls

7.3.1 Establishing Connection

7.3.2 socket creation

7.3.3 bind walkthrough

7.3.4 listen walkthrough

7.3.5 accept walkthrough

7.3.6 connect walkthrough

7.3.7 close walkthrough

7.4 Linux Functions

Chapter 8

Receiving Messages

8.1 Overview

8.2 Receiving Walkthrough

8.2.1 Reading from a socket - I

8.2.2 Receiving a Packet

8.2.3 SoftIRQ - net_rx_action

8.2.4 Unwrapping Packet in IP

8.2.5 Accepting a Packet in UDP

8.2.6 Accepting a Packet in TCP

8.2.7 Reading from a Socket - II

8.3 Linux Functions

Chapter 9

Sending Messages

9.1 Overview

9.2 Sending Walkthrough

9.2.1 Writing to a socket

9.2.2 Creating a Packet with UDP

9.2.3 Creating a Packet with TCP

9.2.4 Wrapping a Packet in IP

9.2.5 Transmitting a Packet

9.3 Linux Functions

Chapter 10

IP Routing

10.1 Overview

Chapter 11

IP Forwarding

11.1 Overview

Chapter 12

Netfilter

12.1 Overview

GNU Free Document License

Bibliography

[1] Daniel P. Bovet & Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001, ISBN 81-7366-233-9.