UDP Socket Creation

The *socketcall* system call.

All standard library functions that operate upon sockets (e.g., `socket()`, `bind()`, `listen()`, `accept()`) share the single system call, *sys_socketcall*. This front end is defined in `net/socket.c`. The parameter, *call*, that is passed to the *sys_socketcall* front end is an *integer* that identifies the specific operation to be performed. The possible values for *call* are defined in `include/linux/net.h`:

```c
30 #define SYS_SOCKET  1  /* sys_socket(2) */
31 #define SYS_BIND    2  /* sys_bind(2) */
32 #define SYS_CONNECT 3  /* sys_connect(2) */
33 #define SYS_LISTEN  4  /* sys_listen(2) */
34 #define SYS_ACCEPT  5  /* sys_accept(2) */
35 #define SYS_GETSOCKNAME 6  /* sys_getsockname(2) */
36 #define SYS_GETPEERNAME 7  /* sys_getpeername(2) */
37 #define SYS_SOCKETPAIR 8  /* sys_socketpair(2) */
38 #define SYS_SEND    9  /* sys_send(2) */
39 #define SYS_RECV   10  /* sys_recv(2) */
40 #define SYS_SENDTO 11  /* sys_sendto(2) */
41 #define SYS_RECVFROM 12  /* sys_recvfrom(2) */
42 #define SYS_SHUTDOWN 13  /* sys_shutdown(2) */
43 #define SYS_SETSOCKOPT 14  /* sys_setsockopt(2) */
44 #define SYS_GETSOCKOPT 15  /* sys_getsockopt(2) */
45 #define SYS_SENDMSG 16  /* sys_sendmsg(2) */
46 #define SYS_RECVMSG 17  /* sys_recvmsg(2) */
```
The `sys_socketcall()` interface.

The `sys_socketcall` function is passed one of the function call identifiers (SYS_SOCKET.., SYS_RECVMSG) in the `call` parameter. It is also passed a pointer to the table of argument pointers via the parameter `args`.

```c
1971 asmlinkage long sys_socketcall(int call, 
        unsigned long __user *args)
1972 {
1973     unsigned long a[6];  // <--- local copy of args
1974     unsigned long a0,a1;
1975     int err;

1977     if(call < 1 || call>SYS_RECVMSG)
1978         return -EINVAL;
1979```

It first checks whether the specific socket function call identifier is a valid one.
Passing arguments through `sys_socketcall()`

An array containing the number of argument passed to each of the calls, `nargs[18]`, is defined in net/socket.c. There is one element for each of the 17 socket functions. The value of `nargs[j]` is the number of arguments required by function `j`. The macro `AL(x)` converts the number of arguments to the size of the argument list by multiplying by the size of a long integer. For example, the `bind` function has index `j = 2` and has 3 arguments and thus the value of `nargs[2]` is 12.

```c
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
```

The entry for bind is shown in emphasized typeface in the table below.

```
1530 /* Argument list sizes for sys_socketcall */
1531 #define AL(x) ((x) * sizeof(unsigned long))
1532 static unsigned char
1533 nargs[18] = {AL(0), AL(3), AL(3), AL(3), AL(2), AL(3),
1534 AL(3), AL(3), AL(4), AL(4), AL(6),
1535 AL(6), AL(2), AL(5), AL(5), AL(3), AL(3)};
```

The `sys_socketcall()` function next copies the arguments from user-space to the local array `a[]` in kernel-space using the size of argument list specified in the `nargs[]` array. Auditing is a feature new to kernel 2.6 and is associated with Security Enhance Linux (SEL).

```c
1980     /* copy_from_user should be SMP safe. */
1981     if (copy_from_user(a, args, nargs[call]))
1982         return -EFAULT;
1983
1984     err = audit_socketcall(nargs[call] /
1985         sizeof(unsigned long), a);
1986     if (err)
1987         return err;
```
All socket functions have at least two arguments. The first two are stored in a0 and a1.

```c
1988    a0=a[0];
1989    a1=a[1];
```

**Invoking the call-specific handler**

A switch statement then calls to the required socket service function identified by the socket function call identifier. A few of these are shown here:

```c
1991    switch(call) 
1992    { 
1993    case SYS_SOCKET: 
1994        err = sys_socket(a0,a1,a[2]); 
1995        break; 
1996    case SYS_BIND: 
1997        err = sys_bind(a0,(struct sockaddr __user *)a1, a[2]); 
1998        break; 
1999    case SYS_CONNECT: 
2000        err = sys_connect(a0, (struct sockaddr __user *)a1, 
2001              a[2]); 
2002        break; 
2003    case SYS_LISTEN: 
2004        err = sys_listen(a0,a1); 
2005    : 
2006    default: 
2007        err = -EINVAL; 
2008        break; 
2009    } 
2050    return err; 
```
Generic Socket Creation

A user program creates a socket via the `socket()` function call:

```
int socket (int family, int type, int protocol);
```

- The `family` parameter specifies a protocol family such as `PF_INET`.
- The `type` parameter is normally `SOCK_STREAM (TCP)` or `SOCK_DGRAM (UDP)`, but can also be `SOCK_COP` after your protocol is inserted.
- The `protocol` field is normally set to zero (`IPROTO_IP`) for `PF_INET` sockets, but should be set to `IPROTO_COP` for COP sockets.

If the default protocol, `IPROTO_IP` is selected, the socket type alone is used in identifying the proper `inet_protosw` structure. Thus there is necessarily only ONE choice: TCP for `SOCK_STREAM`; and UDP for `SOCK_DGRAM`.

If the socket type is `SOCK_RAW`, then the protocol `IPPROTO_RAW` should also be specified and the application is expected to provide full IP and transport layer headers. Sockets of this type require root privilege. Values of `IPPROTO_` are enums in `include/linux/in.h` and are in the Linux cross reference.

A COP socket is created as follows. The return value is an `fd_array[]` index if non-negative or an error code if negative.

```
sd = socket (PF_INET, SOCK_COP, IPPROTO_COP);
```
The *sys_socket* function

The application function *socket()* produces a call to the *sys_socket* function which performs two major steps:

- Creates the *struct socket*, and the *struct sock*, and links them together
- Maps the *struct socket* into the file system space

As indicated before, the prefix *sock_* indicates that a variable or function pertains to a *struct socket* while the prefix *sk_* indicates it pertains to a struct sock.

```c
1239 asmlinkage long sys_socket(int family, int type, int protocol)  
1240 {  
1241    int retval;  
1242    struct socket *sock; // socket addr returned here.  
1243
1244    retval = sock_create(family, type, protocol, &sock);  
1245    if (retval < 0)  
1246          goto out;  
1247
1248    retval = sock_map_fd(sock);  
1249    if (retval < 0)  
1250          goto out_release;  
1251
```

Mapping the socket into the file space

If the socket was created successfully, it must be mapped into the file space so that it can be accessed via the *fd* which is returned in *retval*.

```c
```
Return from sys_socket()

The value returned retval is the index of the struct file associated with the sock in the fd table.

1252 out:
1253 /* It may be already another descriptor 8)
1254       Not kernel problem. */
1255     return retval;
1256

1256 out_release:
1257     sock_release(sock);
1258     return retval;
1259 }
The `sock_create()` function

The socket layer functions are found in `net/socket.c`. This function just calls `__sock_create()` indicating that this is not an internal kernel requestion.

```c
1229 int sock_create(int family, int type, int protocol,
    struct socket **res)
1230 {
1231     return __sock_create(family, type, protocol, res, 0);
1232 }
```

The first step is to verify that both the `family` (PF_INET) and the `type` (SOCK_DGRAM) are in range. This test does not ensure that they are actually valid.

```c
1124 static int __sock_create(int family, int type, int protocol,
    struct socket **res, int kern)
1125 {
1126     int err;
1127     struct socket *sock;
1128 
1129     /*
1130      * Check protocol is in range
1131     */
1132     if (family < 0 || family >= NPROTO)
1133         return -EAFNOSUPPORT;
1134     if (type < 0 || type >= SOCK_MAX)
1135         return -EINVAL;
1136 ```
Exactly one warning that the type *SOCK_PACKET* is deprecated is also generated if that type is specified.

1137    /* Compatibility.
1138
1139      This uglymoron is moved from INET layer to here to avoid
1140      deadlock in module load.
1141    */
1142    if (family == PF_INET && type == SOCK_PACKET) {
1143        static int warned;
1144        if (!warned) {
1145            warned = 1;
1146            printk(KERN_INFO "%s uses obsolete 
1147                    (PF_INET,SOCK_PACKET)\n", current->comm);
1148        }
1149        family = PF_PACKET;
1150    }
Dynamic loading of entire protocol family

The security socket create is also related to new SEL "features" in kernel 2.6.

```c
1151   err = security_socket_create(family, type, protocol, kern);
1152   if (err)
1153       return err;
1154
1155#if defined(CONFIG_KMOD)
1156   /* Attempt to load a protocol module if the find failed. */
1157   */
1158   if (net_families[family]==NULL)
1159   {
1160       request_module("net-pf-%d",family);
1161   }
1162#endif
```

Linux supports the dynamic loading of complete protocol stacks (families) as modules. If `sock_create()` finds the requested protocol family pointer in `net_families[]` to be null, it attempts to dynamically load it using the `request_module()` function. This function tries to locate and load the module using the alias entries in `/etc/modules.conf` and in the folder `/lib/modules/'kernel-version'`. For example, if

```c
#define PF_NEWPROTO 9
```

is requested then the module name requested will be `net-pf-9`. Since the PF_INET is always registers itself at boot time, this code is not relevant to creating sockets of type PF_INET.

```c
1155#if defined(CONFIG_KMOD)
1156   /* Attempt to load a protocol module if the find failed. */
1157   */
1162   if (net_families[family]==NULL)
1163   {
1164       request_module("net-pf-%d",family);
1165   }
1166#endif
```

After the attempt to load the protocol completes, if the protocol family remains unregistered then a fatal error is recognized. This error should never occur during creation of a UDP socket.

```c
1168   net_family_read_lock();
1169   if (net_families[family] == NULL) {
1170       err = -EAFNOSUPPORT;
1171       goto out;
1172   }
1173```
The **struct socket**

The importance of the **struct socket**, defined in `include/linux/net.h` has diminished over time as the more important data structures have moved to the **struct sock**. The **ops** field provides the binding from the thin **socket** layer to the **PF_** layer.

The **struct socket** and the **inode** are embedded in the **struct socket_alloc**. The **sock_inode_cache** of **socket_alloc** structures was created during the call to **sock_init**.

```c
682 struct socket_alloc {
   683     struct socket socket;
   684     struct inode vfs_inode;
685 }
```

```c
107 struct socket {
   108     socket_state state;
   109     unsigned long flags;
   110     const struct proto_ops *ops;
   111     struct fasync_struct *fasync_list;
   112     struct file *file;
   113     struct sock *sk;
   114     wait_queue_head_t wait;
   115     short type;
116 }
```

- **type:** SOCK_STREAM, SOCK_DGRAM, etc.

- **proto_ops:** A pointer to the **proto_ops** structure that has been retrieved from the **ip_protosw** structure.

- **file:** Use to access file_ops function pointer table for read/write calls.
Allocation of the *struct socket*

Next *sock_create()* invokes *sock_alloc()* to create the socket and inode structures. This leads to a torturous path the eventually leads back to the function *sock_alloc_inode()* which also resides in *socket.c*

```
 1179  if (!((sock = sock_alloc()))) {
 1180      if (net_ratelimit())
 1181          printk(KERN_WARNING "socket: no more sockets\n");
 1182          err = -ENFILE;   /* Not exactly a match, but its the
 1183              closest posix thing */
 1184          goto out;
 1185      }
 1186  }
 1187
```

When *sock_alloc()* returns the new socket structure address to *sock_create()* the socket type (SOCK_DGRAM for UDP) is stored in the structure.

```
 1188  sock->type  = type;   /* SOCK_STREAM, etc */
 1189
```
Protocol family dependent initialization

It is not necessary to store the PF because that is implicit in the `create` function that is used. For PF_INET this function is `inet_create()` which was previously encountered in the `inet_init()` function. Failure to provide a create function will cause a kernel oops here!

Failure of the family dependent create function to fill in the `ops` pointer is also fatal.

```c
1194    err = -EAFNOSUPPORT;
1195    if (!try_module_get(net_families[family]->owner))
1196        goto out_release;
1197
This is the call to `inet_create()`
```

```c
1198    if ((err = net_families[family]->create(sock, protocol)) < 0) {
1199        sock->ops = NULL;
1200        goto out_module_put;
1201    }
```
Return from *sock_create*

The *sock_create()* function stores the socket pointer in the location provided by *sys_socket()* and then returns *err* to *sys_socket()*.

    1215   module_put(net_families[family]->owner);
    1216   *res = sock;
    1217   security_socket_post_create(sock, family, type,
          protocol, kern);
    1218  
    1219     out:
    1220   net_family_read_unlock();
    1221   return err;
    1222  
    1223     out_module_put:
    1224   module_put(net_families[family]->owner);
    1225  
    1226     out_release:
    1227   sock_release(sock);
    1228   goto out;
Socket allocation details

The `sock_alloc()` function defined in `net/socket.c` eventually allocates a `struct socket_alloc`. The `struct socket` and the `inode` are embedded in the `struct socket_alloc`.

The `sock_mnt` pointer is a global variable in `socket.c` that was set up to point to the super block of the socket virtual filesystem when the system was mounted. The superblock contains a pointer to a table of super block operations. Among these is a pointer to the inode allocator.

```c
514 static struct socket *sock_alloc(void)
515 {
516    struct inode *inode;
517    struct socket *sock;
518
519    inode = new_inode(sock_mnt->mnt_sb);
520    if (!inode)
521        return NULL;
522
523    sock = SOCKET_I(inode);
524    inode->i_mode = S_IFSOCK|S_IRWXUGO;
525    inode->i_uid = current->fsuid;
526    inode->i_gid = current->fsgid;
527
528    get_cpu_var(sockets_in_use)++;
529    put_cpu_var(sockets_in_use);
530    return sock;
531 }
```

The `SOCKET_I` and `SOCK_INODE` macros recover the socket pointer from the inode pointer and vice versa.
The *struct socket_alloc* cache

Both socket and inode structures are imbedded in the *struct socket_alloc*.

A cache of these structure was created by the *sock_init* function seen earlier in the *netinit* section. The *sock_mnt* variable is a global variable that points to the *super block* of the *sock_fs*.

```
2226        init_inodecache();
2227        register_filesystem(&sock_fs_type);
2228        sock_mnt = kern_mount(&sock_fs_type);
```

The socket_alloc structure is a linux "container".

```
682 struct socket_alloc {
683    struct socket socket;
684    struct inode vfs_inode;
685   };
```

Macros are provided to convert a pointer to one element to a pointer to the other.

```
687 static inline struct socket *SOCKET_I(struct inode *inode)
688 {    return &container_of(inode, struct socket_alloc,
689          vfs_inode)->socket;
690 }
691
692 static inline struct inode *SOCK_INODE(struct socket *socket)
693 {    return &container_of(socket, struct socket_alloc,
694          socket)->vfs_inode;
695 }
```
/**
 * container_of - cast a member of a structure out to the containing structure
 * @ptr:        the pointer to the member.
 * @type:       the type of the container struct this is embedded in.
 * @member:     the name of the member within the struct.
 *
 #define container_of(ptr, type, member){               
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    
    (type *)( (char *)__mptr - offsetof(type,member) );})
*/
The *inode* structure

The union at the end of the *inode* structure used to contain instances of a variety of *fs* type dependent structures including the *struct socket*. Now little if anything within the *inode* is used in network processing. Its primary purpose is to bind the *fd* space to the *struct socket*.

```c
struct inode {
    struct hlist_node       i_hash;
    struct list_head        i_list;
    struct list_head        i_sb_list;
    struct list_head        i_dentry;
    unsigned long           i_ino;
    atomic_t                i_count;
    umode_t                 i_mode;
    unsigned int            i_nlink;

    atomic_t                i_writecount;
    void                    *i_security;
    union {
        void            *generic_ip;
    } u;

    #ifdef __NEED_I_SIZE_ORDERED
    seqcount_t              i_size_seqcount;
    #endif
    #endif
    #endif
};
```
Inode allocation

The `new_inode()` function is defined in fs/inode.c. Memory resident generic `inode` structures can be allocated by the slab allocator from an inode cache created at boot time or provided from special cache created by the vfs itself.

The `new_inode()` functions ensures that inode allocation succeeded then proceeds to enqueue in on the `inode_in_use` list and initialize several fields. The `inode` numbers are sequentially assigned using the static variable `last_ino`.

```c
547 struct inode *new_inode(struct super_block *sb) 548 { 549     static unsigned long last_ino; 550     struct inode * inode; 551 552     spin_lock_prefetch(&inode_lock); 553 554     inode = alloc_inode(sb); 555     if (inode) { 556         spin_lock(&inode_lock); 557         inodes_stat.nr_inodes++; 558         list_add(&inode->i_list, &inode_in_use); 559         list_add(&inode->i_sb_list, &sb->s_inodes); 560         inode->i_ino = ++last_ino; 561         inode->i_state = 0; 562         spin_unlock(&inode_lock); 563     } 564     return inode; 565 }
```
The `alloc_inode()` function.

The `alloc_inode()` macro is defined in fs/inode.c. If the `super_block`'s `s_op` structure doesn't contain a pointer to an inode allocator, it allocates an inode from the generic inode_cache.

In the case of a socket such a pointer does exist and the actual allocation is performed by `sock_alloc_inode()`.

```c
331 static struct super_operations sockfs_ops = {
332     alloc_inode = sock_alloc_inode,
333     destroy_inode = sock_destroy_inode,
334     statfs = simple_statfs,
335 };;
```

```c
100 static kmem_cache_t * inode_cachep __read_mostly;
101
102 static struct inode *alloc_inode(struct super_block *sb) {
104     static const struct address_space_operations empty_aops;
105     static struct inode_operations empty_iops;
106     static const struct file_operations empty_fops;
107     struct inode *inode;
108
109     if (sb->s_op->alloc_inode)
110         inode = sb->s_op->alloc_inode(sb);
111     else
112         inode = (struct inode *) kmem_cache_alloc(
113             inode_cachep, SLAB_KERNEL);
```
If allocation succeeds a considerable amount of generic initialization follows.

```
114     if (inode) {
115         struct address_space * const mapping = &inode->i_data;
116
117         inode->i_sb = sb;
118         inode->i_blkbits = sb->s_blocksize_bits;
119         inode->i_flags = 0;
120         atomic_set(&inode->i_count, 1);
121         inode->i_op = &empty_iops;
122         inode->i_fop = &empty_fops;
123         inode->i_nlink = 1;
124         atomic_set(&inode->i_writecount, 0);
125         inode->i_size = 0;
126         inode->i_blocks = 0;
127         inode->i_bytes = 0;
128         inode->i_generation = 0;
168     }
169     return inode;
170}
```
The *sock_alloc_inode()* function.

Actual allocation of *socket* and *inode* structures from the *sock_inode_cache* is performed here.

285 static struct inode *sock_alloc_inode(struct super_block *sb)  
286 {  
287         struct socket_alloc *ei;  
288         ei = (struct socket_alloc *)  
289             kmem_cache_alloc(sock_inode_cachep, SLAB_KERNEL);  
289         if (!ei)  
290             return NULL;  
291         init_waitqueue_head(&ei->socket.wait);  
292   
293         ei->socket.fasync_list = NULL;  
294         ei->socket.state = SS_UNCONNECTED;  
295         ei->socket.flags = 0;  
296         ei->socket.ops = NULL;  
297         ei->socket.sk = NULL;  
298         ei->socket.file = NULL;  
299         ei->socket.flags = 0;  
300        return &ei->vfs_inode;  
302 }

This function is also responsible for initializing the struct *socket* to a consistent state.
Protocol family dependent initialization

The mission of IPV4 dependent socket creation is to create and initialize the struct inet_sock. The struct socket and struct sock are two related and easy-to-confuse structures that are widely used within the TCP/IP implementation. To minimize the confusion a standard naming convention is used.

- sock always refer to the generic struct socket
- sk always refer to the protocol dependent structure struct sock.

It used to be the case that the struct sock contained a mix of somewhat generic stuff along with IP dependent stuff. The IP related fields are now moved to the inet_sock and transport dependent fields live in the a transport dependent extension such as tcp_sock.

Each transport sock must contain an instance of an inet_sock as its first element. The inet_sock contains an instance of the struct sock as its first element.

Therefore pointers to any of the three may be freely cast to any other.
The struct inet_sock

The IPv4 struct inet_sock is defined in include/net/sock.h. The elements shown in blue must be used by your COP protocol for managing connections. All fields except num must be in network byte order.

```
108 struct inet_sock {
109 /* sk and pinet6 must be the first two members of inet_sock */
110     struct sock             sk;
111#if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
112        struct ipv6_pinfo       *pinet6;
113#endif
114    /* Socket demultiplex comparisons on incoming packets. */
115    __u32 daddr;
116    __u32 rcv_saddr;
117    __u16 dport;
118    __u16 num;
119    __u32 saddr;
120    __s16 uc_ttl;
121    __u16 cmsg_flags;
122    struct ip_options       *opt;
123    __u16 sport;
124    __u16 id;
125    __u8  tos;
```

daddr  IP address of the remote endpoint of a connected socket
rcv_saddr  Local IP address to which this socket is bound. This must be the destination IP address that is carried by incoming packet to be matched or the wildcard 0.
dport  Remote port address for a connected socket
num  Local port address in host byte order to which this socket is bound (or protocol number for raw sockets
sport  Local port to which socket is bound in network byte order
saddr  Local IP address which outgoing packets will carry as source address. This is usually the same as rcv_saddr if rcv_saddr != 0.
Sock stuctures

The `struct sock_common` is apparently intended to house elements that are common to either all transport protocols or all protocol families.

The `hlist_node` elements are used to link the `struct sock` into hash queues. Your protocol must provide a table, `struct hlist_head cop_hash[COP_HTABLE_SIZE]` hashed list headers and the helper function will use the `skc_node` structure to link the `struct sock` into your queue. These queues underly the mechanism by which incoming packets are matched to a particular `struct sock`.

For raw sockets, the `skc_node` is used to link all the sk's associated with a given `struct proto`. A pointer to the hashing function is contained in the `struct proto`. For the raw `struct proto` the hash key is the low order 5 bits of the protocol number.

In contrast, for UDP, the 64K port space is divided into 128 hash queues and the `struct socks` are mapped to a hash queue by local port number. The queuing mechanism is a bit unusual here.

The `skc_bind_node` is not used in raw or UDP sockets. It is are used by TCP, to link all `struct socks` that are bound to a single port.

Other fields are dependent upon system configuration and include areas that are private to specific protocols.

```c
struct sock_common {
    unsigned short skc_family;
    volatile unsigned char skc_state;
    unsigned char skc_reuse;
    int skc_bound_dev_if;
    struct hlist_node skc_node;
    struct hlist_node skc_bind_node;
    atomic_t skc_refcnt;
    unsigned int skc_hash;
    struct proto *skc_prot;
};
```
The **struct sock**

Elements of this large structure are critical to network operation, but very few of them must be directly manipulated by a correctly written transport protocol. The first group is a collection of aliases designed to ease migration to the use of the *sock_common*. In the 2.4 kernel the elements did not possess the *sk_* prefix so a *lot* of code had to be changed on the way to 2.6.

```
182 struct sock {
183 /*
184 * Now struct inet_timewait_sock also uses sock_common, so please just
185 * don't add nothing before this first member (__sk_common)
186 */
187 struct sock_common __sk_common;
188 #define sk_family            __sk_common.skc_family
189 #define sk_state             __sk_common.skc_state
190 #define sk_reuse             __sk_common.skc_reuse
191 #define sk_bound_dev_if      __sk_common.skc_bound_dev_if
192 #define sk_node              __sk_common.skc_node
193 #define sk_bind_node         __sk_common.skc_bind_node
194 #define sk_refcnt            __sk_common.skc_refcnt
195 #define sk_hash              __sk_common.skc_hash
196 #define sk_prot              __sk_common.skc_prot

197 unsigned char            sk_shutdown : 2,
198 unsigned char            sk_no_check : 2,
199 unsigned char            sk_userlocks : 4;
200 unsigned char            sk_protocol;    // 6 is TCP
201 unsigned short           sk_type;
202 int                      sk_rcvbuf;     // buffer quota
203 socket_lock_t            sk_lock;
204 wait_queue_head_t        *sk_sleep;
205 struct dst_entry         *sk_dst_cache;  // routing data
206 struct xfrm_policy       *sk_policy[2];
207 rwlock_t                  sk_dst_lock;
```
The backlog queue plays a big role in TCP input processing but UDP/COP do not use one.
struct sk_buff_head sk_error_queue;
struct proto *sk_prot_creator;
rwlock_t sk_callback_lock;
int sk_err,
     sk_err_soft;
unsigned short sk_ack_backlog;
unsigned short sk_max_ack_backlog;
__u32 sk_priority;
struct ucred sk_peercred;
long sk_rcvtimeo;
long sk_sndtimeo;
struct sk_filter *sk_filter;
void *sk_protinfo;
struct timer_list sk_timer;
struct timeval sk_stamp;
struct socket *sk_socket;
void *sk_user_data;
struct page *sk_sndmsg_page;
struct sk_buff *sk_send_head;
__u32 sk_sndmsg_off;
int sk_write_pending;
void *sk_security;
void (*sk_state_change)(struct sock *sk);
void (*sk_data_ready)(struct sock *sk, int bytes);
void (*sk_write_space)(struct sock *sk);
void (*sk_error_report)(struct sock *sk);
int (*sk_backlog_rcv)(struct sock *sk,
                      struct sk_buff *skb);
void (*sk_destruct)(struct sock *sk);
IP dependent socket creation with \texttt{inet\_create()}

The \texttt{inet\_create()} function resides in \texttt{linux/net/ipv4/af\_inet.c}. It is passed a pointer to the generic \texttt{struct socket} and the either the IP specific transport protocol number passed to \texttt{sys\_socket} or (typically in the case of UDP and TCP) the wild-card value of 0. Its primary function is to create and initialize the \texttt{struct sock}.

The \textit{answer} variables are set in the initial lookup and frequently referenced throughout.

\begin{verbatim}
static int inet_create(struct socket *sock, int protocol)
{
        struct sock *sk;
        struct list_head *p;
        struct inet_protosw *answer;
        struct inet_sock *inet;
        struct proto *answer_prot;
        unsigned char answer_flags;
        char answer_no_check;
        int try_loading_module = 0;
        int err;

        sock->state = SS_UNCONNECTED;

        ... (remaining code here) ...

}
\end{verbatim}
The protocol matching procedure

In the protocol lookup loop, the list of all the *inet_protosw* structures (normally only 1) associated with socket type, (SOCK_DGRAM for UDP), is searched.

The input protocol is typically 0 (IPPROTO_IP), and in line number 249 a protocol match is found. In line 253 protocol is set to the default protocol namely IPPROTO_UDP for the SOCK_DGRAM socket type.

It's also possible that IPPROTO_IP may also be the protocol type specified in the *struct inet_protosw*. In that case the match will occur at line 256.

238  /* Look for the requested type/protocol pair. */
239     answer = NULL;
240 lookup_protocol:
241     err = -ESOCKTNOSUPPORT;
242     rcu_read_lock();
243     list_for_each_rcu(p, &inetsw[sock->type]) {
244         answer = list_entry(p, struct inet_protosw, list);
245
246       /* Check the non-wild match. */
247       if (protocol == answer->protocol) {
248           if (protocol != IPPROTO_IP)
249               break;
250         } else {
251           /* Check for the two wild cases. */
252           if (IPPROTO_IP == protocol) {
253               protocol = answer->protocol;
254               break;
255           }
256           if (IPPROTO_IP == answer->protocol)
257               break;
258       }
259     } else {
260         /* Check for the two wild cases. */
261         if (IPPROTO_IP == protocol) {
262             protocol = answer->protocol;
263             break;
264         }
265     }
266     err = -EPROTONOSUPPORT;
267     answer = NULL;
Dynamic loading of transport protocol modules

In kernel 2.6 the capability to demand load specific transport protocols was also added. It should be possible to load NTP this way but it will not be a requirement. Apparently the `try_loading_module` variable determines the naming strategy. If a module is loaded, its necessary to go back to `lookup_protocol` to see if the actual socket type and protocol registered by the module actually match.

```c
263     if (unlikely(answer == NULL)) {
264         if (try_loading_module < 2) {
265             rcu_read_unlock();
266         /*
267             * Be more specific, e.g. net-pf-2-proto-132-type-1
268             * (net-pf-PF_INET proto-IPPROTO_SCTP type- SOCK_STREAM)
269                 */
270             if (++try_loading_module == 1)
271                 request_module("net-pf-%d-proto-%d-type-%d", "PF_INET", protocol, sock->type);
272         */
274     /*
275        * Fall back to generic, e.g. net-pf-2-proto-132
276        * (net-pf-PF_INET proto-IPPROTO_SCTP)
277                */
278     else
279         request_module("net-pf-%d-proto-%d", PF_INET, protocol);
280     goto lookup_protocol;
281     } else
282     goto out_rcu_unlock;
283 }
```

Verify that required capability is possessed by the process trying to create the socket. This is where unprivileged apps trying to open raw sockets get snagged.

```c
285     err = -EPERM;
286     if (answer->capability > 0 &&
287         !capable(answer->capability))
288     goto out_rcu_unlock;
```
Setting the *struct proto_ops*.

Here the *struct proto_ops* pointer is copied from the *inet_protosw* structure to the *struct socket*. This contains pointers to the high level (somewhat transport independent) entry points to the AF_INET protocol stack. Storing of the *struct proto* pointer *answer_prot* must be deferred because the *struct sock* is not yet allocated.

```c
289   sock->ops = answer->ops;
290   answer_prot = answer->prot;
```

For sockets of type SOCK_DGRAM, *sock->ops* points to the *inet_dgram_ops* structure which is defined in net/ipv4/af_inet.c

```c
973   struct proto_ops inet_dgram_ops = {
974       family: PF_INET,
975       release: inet_release,
976       bind: inet_bind,
977       connect: inet_dgram_connect,
978       socketpair: sock_no_socketpair,
979       accept: sock_no_accept,
980       getname: inet_getname,
981       poll: datagram_poll,
982       ioctl: inet_ioctl,
983       listen: sock_no_listen,
984       shutdown: inet_shutdown,
985       setsockopt: inet_setsockopt,
986       getsockopt: inet_getsockopt,
987       sendmsg: inet_sendmsg,
988       recvmsg: inet_recvmsg,
989       mmap: sock_no_mmap,
990       sendpage: sock_no_sendpage,
991   };
```
The value of `no_check` for UDP is `UDP_CSUM_DEFAULT` and the value of flags is `INET_PROTOSW_PERMANENT`.

```c
291        answer_no_check = answer->no_check;
292        answer_flags = answer->flags;
293        rcu_read_unlock();
294
295        BUG_TRAP(answer_prot->slab != NULL);
296
297        err = -ENOBUFS;
298        sk = sk_alloc(PF_INET, GFP_KERNEL, answer_prot, 1);
299
300        if (sk == NULL)
301              goto out;
302
303        err = 0;
304        sk->sk_no_check = answer_no_check;
305        if (INET_PROTOSW_REUSE & answer_flags)
306              sk->sk_reuse = 1;
307
308        inet = inet_sk(sk);
309        inet->is_icsk = INET_PROTOSW_ICSK & answer_flags;
```

The `inet_sk()` macro simply casts a struct sock pointer to a struct inet_sock pointer. This is a useful macro for you to use. You can also build a `cop_sk()`.
The overloaded sk->num field

Here we see that sk->num is set to the protocol number for sockets of type raw. The comments in the structure definition say that this field is the local port number. Later in this section we will see that for non-raw sockets it does indeed contain a the local port number.

```c
if (SOCK_RAW == sock->type) {
    inet->num = protocol;
    if (IPPROTO_RAW == protocol)
        inet->hdrincl = 1;
}
```

```c
if (ipv4_config.no_pmtu_disc)
    inet->pmtudisc = IP_PMTUDISC_DONT;
else
    inet->pmtudisc = IP_PMTUDISC_WANT;
```

```c
inet->id = 0;
```

The sock_init_data() function defined in net/core/sock.c initializes the struct sock and links it with the struct socket. The sk pointer in the struct socket points to a struct sock.

```c
sock_init_data(sock, sk);
```

```c
sk->sk_destruct = inet_sock_destruct;
sk->sk_family = PF_INET;
```

```c
sk->sk_protocol = protocol;
```

```c
sk->sk_backlog_rcv = sk->sk_prot->backlog_rcv;
```

```c
inet->uc_ttl = -1;
inet->mc_loop = 1;
inet->mc_ttl = 1;
inet->mc_index = 0;
inet->mc_list = NULL;
```

```c
sk_refcnt_debug_inc(sk);
```

34
Linking raw sockets into a hash queue.

The value of `sk->num` as stated earlier is overloaded. For raw sockets it was set to the protocol number. Thus, it is mandatory that the protocol provide a hash function. For UDP/TCP sockets created as SOCK_DGRAM or SOCK_STREAM `sk->num` is the local port number and will be 0 here. Sockets of type SOCK_DGRAM or SOCK_STREAM are hashed at bind time.

For non-raw sockets `inet->num != 0` is a very unusual circumstance. Normally port numbers are assigned only at bind/connect time.

```c
338    if (inet->num) {
339        /* It assumes that any protocol which allows
340         * the user to assign a number at socket
341         * creation time automatically
342         * shares.
343         */
344        inet->sport = htons(inet->num);
345        /* Add to protocol hash chains. */
346        sk->sk_prot->hash(sk);
347    }
348```
Invoking the transport protocol initialization procedure

Since the `udp_prot` structure doesn’t provide an `init` function the following code block does nothing in the case of a UDP socket. Your COP module must provide an `init` function. For now it should just issue a `printk()` verifying that it was called. The `sk_common_release(sk)` function will be called by your `cop_close()` function to free the `struct sock`.

```c
349    if (sk->sk_prot->init) {
350       err = sk->sk_prot->init(sk);
351       if (err)
352          sk_common_release(skb);
353    }  
354 out:
355    return err;
356 out_rcu_unlock:
357    rcu_read_unlock();
358    goto out;
359}
```
The `sk_alloc()` function resides in net/core/sock.c and allocates the requested structure from the cache created during the call to `proto_register()`. Here the item allocated has `sizeof(struct copsock)`. The `struct proto` contains the entry points into the true transport protocol. The `sk->prot` field is set to the `struct proto` pointer from the `answer` pointer to `struct inet_protosw`.

```c
840 struct sock *sk_alloc(int family, gfp_t priority,
841                      struct proto *prot, int zero_it)
842 {
843    struct sock *sk = NULL;
844    kmem_cache_t *slab = prot->slab;
845
846    if (slab != NULL)
847        sk = kmem_cache_alloc(slab, priority);
848    else
849          sk = kmalloc(prot->obj_size, priority);
850
851    if (sk) {
852        if (zero_it) {
853            memset(sk, 0, prot->obj_size);
854            sk->sk_family = family;
855        /* See comment in struct sock definition to understand
856          * why we need sk_prot_creator -acme
857        */
858            sk->sk_prot = sk->sk_prot_creator = prot;
859            sock_lock_init(sk);
860        }
861    
862    if (security_sk_alloc(sk, family, priority))
863        goto out_free;
864 865    if (!try_module_get(prot->owner))
866        goto out_free;
867 868    }
869    return sk;
870```

Allocation of the `struct sock`
out_free:
    if (slab != NULL)
        kmem_cache_free(slab, sk);
    else
        kfree(sk);
    return NULL;
**Initialization of the `struct_sock`**

1477 void sock_init_data(struct socket *sock, struct sock *sk)
1478 {

Initialize buffer queues for packets awaiting local delivery or transmission

1479 skb_queue_head_init(&sk->sk_receive_queue);
1480 skb_queue_head_init(&sk->sk_write_queue);
1481 skb_queue_head_init(&sk->sk_error_queue);

1482#ifndef CONFIG_NET_DMA
1483 skb_queue_head_init(&sk->sk_async_wait_queue);
1484#endif
1485
1486 sk->sk_send_head = NULL;
1487
1488 init_timer(&sk->sk_timer);

Initialize buffer space allocation flags and space quotas.

1490 sk->sk_allocation = GFP_KERNEL;
1491 sk->sk_rcvbuf = sysctl_rmem_default;
1492 sk->sk_sndbuf = sysctl_wmem_default;
1493 sk->sk_state = TCP_CLOSE;

Set back pointer to struct socket.

1494 sk->sk_socket = sock;
1495 sock_set_flag(sk, SOCK_ZAPPED);
1496
1497

39
For UDP the sock pointer will point to the struct socket just allocated. The \textit{waitqueue} pointer of the \textit{struct sock} refers to the \textit{waitqueue} structure embeded in the \textit{struct socket}. The forward pointer in the struct socket is set to point to the struct sock in line 1502.

\begin{verbatim}
1498        if(sock)
1499        {
1500           sk->sk_type = sock->type;
1501           sk->sk_sleep = &sock->wait;
1502           sock->sk = sk;
1503        } else
1504           sk->sk_sleep = NULL;
1505
1506        rwlock_init(&sk->sk_dst_lock);
1507        rwlock_init(&sk->sk_callback_lock);
1508        lockdep_set_class(&sk->sk_callback_lock, af_callback_keys + sk->sk_family);
1509
1510        func_pointers for managing sleep/wakeup transitions. Note for PF_INET all of these are
generics that live in the \textit{socket} layer.

1511        sk->sk_state_change = sock_def_wakeup;
1512        sk->sk_data_ready = sock_def_readable;
1513        sk->sk_write_space = sock_def_write_space;
1514        sk->sk_error_report = sock_def_error_report;
1515        sk->sk_destruct = sock_def_destruct;
1516
1517        sk->sk_sndmsg_page = NULL;
1518        sk->sk_sndmsg_off = 0;
1519
1520        sk->sk_peercred.pid = 0;
1521        sk->sk_peercred.uid = -1;
1522        sk->sk_peercred.gid = -1;
1523        sk->sk_write_pending = 0;
1524        sk->sk_rcvlowat = 1;
1525        sk->sk_rcvtimeo = MAX_SCHEDULE_TIMEOUT;
1526        sk->sk_sndtimeo = MAX_SCHEDULE_TIMEOUT;
1527
1528        sk->sk_stamp.tv_sec = -1L;
1529        sk->sk_stamp.tv_usec = -1L;
1530        atomic_set(&sk->sk_refcnt, 1);
1531    }
\end{verbatim}
Data structures used in mapping the socket structure into the file space

Access to a `struct socket` is via a chain of data structures:

```
struct task_struct
struct files_struct
fd_array
struct file
struct dentry
struct inode
struct socket
```
The task_struct

File descriptors are small integers used in Linux to identify open files via the files_struct pointer in the task_struct of the process.

```
767 struct task_struct {
768     volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
769     struct thread_info *thread_info;
770     atomic_t usage;
771     unsigned long flags;  /* per process flags, defined below */
772     unsigned long ptrace;
    :
881/* open file information */
882     struct files_struct *files;
883/* namespace */
```
### The files_struct

The `files_struct` contains the following fields. The field of interest to us is the pointer to the FD array. The value returned by the `open` and `socket` system calls is an index into this table.

```c
48 struct files_struct {
   49     /* read mostly part */
   50       atomic_t count;
   51       struct fdtable *fdt;
   52       struct fdtable fdtab;
   53     /* written part on a separate cache line in SMP */
   54       spinlock_t file_lock ____cacheline_aligned_in_smp;
   55       int next_fd;
   56       struct embedded_fd_set close_on_exec_init;
   57       struct embedded_fd_set open_fds_init;
   58   struct file * fd_array[NR_OPEN_DEFAULT];
   59 }
```
The *struct file*

The kernel manages each open file or socket via the *struct file*. Important linkage elements are the *dentry* pointer and the *file_operations* pointer. The *f_op* table is used in vectoring generic file operations such as *read* and *write* to their handler in the socket system.

```
671 struct file {
675         */
676        union {
677                struct list_head        fu_list;
678                struct rcu_head         fu_rcuhead;
679        } f_u;
680        struct dentry           *f_dentry;
681        struct vfsmount         *f_vfsmnt;
682        const struct file_operations    *f_op;
683        atomic_t                f_count;
684        unsigned int            f_flags;
685        mode_t                  f_mode;
686        loff_t                  f_pos;
687        struct fown_struct      f_owner;
688        unsigned int            f_uid, f_gid;
689        struct file_ra_state    f_ra;
690        unsigned long           f_version;
691        void                    *f_security;
692        void                    *private_data;
693      /* needed for tty driver, and maybe others */
```

44
The **struct dentry**

The kernel cache the directory entries of for recently used files and all sockets in the *dcache*. The *struct dentry* provides access to the *inode*. We know that the *inode* and the *socket* structures are allocated together as a single entity.

```c
struct dentry {
    atomic_t d_count;
    unsigned int d_flags;       /* protected by d_lock */
    spinlock_t d_lock;           /* per dentry lock */
    struct inode *d_inode;       /* Where the name belongs to 

        /*
          * The next three fields are touched by __d_lookup.
          * Place them here
        */
    struct hlist_node d_hash;       /* lookup hash list */
    struct dentry *d_parent;        /* parent directory */
    struct qstr d_name;
};
```
The mapping mechanism

Mapping a file structure associated with the socket into the \textit{fd} space of the current process is performed by \textit{sock\_map\_fd()}, defined in net/socket.c.

```
422 int sock_map_fd(struct socket *sock)  
423 {  
424        struct file *newfile;

The call to \textit{sock\_alloc\_fd()} obtains the index of a free spot in the \textit{fd\_array} and acquires an new \textit{struct file}.

```
425        int fd = sock_alloc_fd(&newfile);
426
```

The call to \textit{sock\_attach\_fd()} does the work of linking file, dentry, and inode.

```
427        if (likely(fd >= 0)) {
428                int err = sock_attach_fd(sock, newfile);
429        
430                if (unlikely(err < 0)) {
431                        put_filp(newfile);
432                        put_unused_fd(fd);
433                        return err;
434                }

The call to \textit{fd\_install()} actually stores the struct file pointer in the \textit{fd} array.

```
435        fd_install(fd, newfile);
436    }
437    return fd;
438}
```
The *sock_alloc_fd()* function.

The following comment taken from the function describes some complicating factors:

```
`This function creates a file structure and maps it to the *fd* space of the current process. On success it returns the file descriptor and the file struct implicitly stored in *sock->file*. Note that another thread may close the file descriptor before we return from this function. We do not refer to socket after the mapping is complete. If one day we will need it, this function will increment reference count on file by 1. In any case returned *fd* may be not valid! This race condition is unavoidable with shared *fd* spaces. We cannot solve it inside the kernel, but do we take care of internal coherence.```

```c
376 static int sock_alloc_fd(struct file **filep)
377 {
378        int fd;
379
380        fd = get_unused_fd();
381        if (likely(fd >= 0)) {
382                struct file *file = get_empty_filp();
383                *filep = file;
384                if (unlikely(!file)) {
385                        put_unused_fd(fd);
386                        return -ENFILE;
387                }
388        } else
389                *filep = NULL;
390        return fd;
392}
```
The `sock_attach_fd()` function

This function creates a name for the pseudo file associated with the socket, allocates a directory entro for the file, points the the struct file at the struct dentry, and asks `d_add()` to link the inode to the dentry.

```c
394 static int sock_attach_fd(struct socket *sock,
                          struct file *file)
395 {
396    struct qstr this;
397    char name[32];
398
399    this.len = sprintf(name, "[%lu]", SOCK_INODE(sock)->i_ino);
400    this.name = name;
401    this.hash = SOCK_INODE(sock)->i_ino;
402
403    file->f_dentry = d_alloc(sock_mnt->mnt_sb->s_root, &this);
404    if (unlikely(!file->f_dentry))
        return -ENOMEM;
405
406    file->f_dentry->d_op = &sockfs_dentry_operations;
407    d_add(file->f_dentry, SOCK_INODE(sock));
408    file->f_vfsmnt = mntget(sock_mnt);
409    file->f_mapping = file->f_dentry->d_inode->i_mapping;
410```

The variable `this` is of type `struct qstr` (quickstring), an efficient way of storing a string and its meta data (len, hash) for quick retrieval. The name of this pseudo file is the ASCII encoding of its inode number.

```c
403    file->f_dentry = d_alloc(sock_mnt->mnt_sb->s_root, &this);
404    if (unlikely(!file->f_dentry))
        return -ENOMEM;
405
406    file->f_dentry->d_op = &sockfs_dentry_operations;
407    d_add(file->f_dentry, SOCK_INODE(sock));
408    file->f_vfsmnt = mntget(sock_mnt);
409    file->f_mapping = file->f_dentry->d_inode->i_mapping;
410```
Binding the standard system I/O calls to the socket

The `socket_file_ops` structure contains the bindings that map "regular" file system calls in to the socket layer.

```c
412    sock->file = file;
413    file->f_op = SOCK_INODE(sock)->i_fop = &socket_file_ops;
414    file->f_mode = FMODE_READ | FMODE_WRITE;
415    file->f_flags = O_RDWR;
416    file->f_pos = 0;
417    file->private_data = sock;
418
419    return 0;
420 }
```

The `socket_file_ops` structure provides the bindings that vector standard file system operations to handlers in the socket file system.

```c
126 static struct file_operations socket_file_ops = {
127     .owner =        THIS_MODULE,
128     .llseek =       no_llseek,
129     .aio_read =     sock_aio_read,
130     .aio_write =    sock_aio_write,
131     .poll =         sock_poll,
132     .unlocked_ioctl = sock_ioctl,
133#ifdef CONFIG_COMPAT
134     .compat_ioctl = compat_sock_ioctl,
135#endif
136     .mmap =         sock_mmap,
137     .open =         sock_no_open, /* special open code to
138                  disallow open via /proc */
139     .release =      sock_close,
140     .fasync =       sock_fasync,
141     .readv =        sock_readv,
142     .writev =       sock_writev,
143     .sendpage =     sock_sendpage,
144     .splice_write = generic_splice_sendpage,
145};
```
Allocating a free fd

The `get_unused_fd()` function is defined in `fs/open.c`... This is the kernel 2.4 version.

```c
713 /*
714 * Find an empty file descriptor entry, and mark it busy.
715 */
716 int get_unused_fd(void)
717 {
718     struct files_struct * files = current->files;
719     int fd, error;
720     error = -EMFILE;
721     write_lock(&files->file_lock);
722     repeat:
723     fd = find_next_zero_bit(files->open_fds,
724         files->max_fdset, files->next_fd);

find_next_zero_bit() defined in include/asm/bitops.h finds the next available fd from the fdset, by
finding the next available bit that can be set as an used fd. When it returns, it is necessary to see if
the number of open files exceeds the maximum.

```c
729 /*
730 * N.B. For clone tasks sharing a files structure, this
test will limit the total number of files that can be opened.
732 */
733     if (fd >= current->rlim[RLIMIT_NOFILE].rlim_cur)
734         goto out;

The fdset can only hold 1024 descriptors initially, but if that limit is reached it may be increased.

```c
736 /* Do we need to expand the fdset array? */
737     if (fd >= files->max_fdset) {
738         error = expand_fdset(files, fd);
739         if (!error) {
740             error = -EMFILE;
741             goto repeat;
742         }
743         goto out;
744     }
```
The \textit{fd} array can only hold 32 (NR\_OPEN\_DEFAULT) file pointers initially, but it may also be expanded.

\begin{verbatim}
    /* Check whether we need to expand the fd array. */
    if (fd >= files->max_fds) {
        error = expand_fd_array(files, fd);
        if (!error) {
            error = -EMFILE;
            goto repeat;
        }
        goto out;
    }

    FD_SET(fd, files->open_fds);
    FD_CLR(fd, files->close_on_exec);
    files->next_fd = fd + 1;
    if 1 /* Sanity check */
    if (files->fd[fd] != NULL) {
        printk(KERN_WARNING "get_unused_fd:slot %d not NULL!\n", fd);
        files->fd[fd] = NULL;
    }
    #endif
    error = fd;
    out:
    write_unlock(&files->file_lock);
    return error;
\end{verbatim}

The \textit{FD\_SET} and \textit{FD\_CLR} macros set or clear the bit indexed by \textit{fd} in the specified arrays.
The *fd_install()* function

The *fd_install()* function is defined in include/linux/file.h, it sets the new file structure in the fd_array. Potential race conditions exist here and are described in a contradictory way in comments at the beginning of the function. The comment:

```
`The VFS is full of places where we drop the file lock between setting the open_fds bitmap and installing the file pointer in the file array. At any such point, we are vulnerable to a dup2() race installing a file in the array before us. We need to detect this and fput() the struct file we are about to overwrite in this case.''
```

seems to be in conflict with the following statement:

```
`It should never happen - if we allow dup2() do it, _really_ bad things will follow.''
```

The code indicates that the latter comment is actually in effect here. If the target slot in the files->fd array is occupied at the time of the call a system crash will be requested by *BUG()*.

```
1064 1065 void fastcall fd_install(unsigned int fd, struct file * file) 1066 { 1067        struct files_struct *files = current->files; 1068        struct fdtable *fdt; 1069        spin_lock(&files->file_lock); 1070        fdt = files_fdtable(files); 1071        BUG_ON(fdt->fd[fd] != NULL); 1072        rcu_assign_pointer(fdt->fd[fd], file); 1073        spin_unlock(&files->file_lock); 1074 } 1075```