Linux Sockets and the Virtual Filesystem

Daniel Noé

May 16, 2008

1 Introduction

The interface used by the Berkeley Sockets API uses file descriptors to identify sockets from user space. This allows standard interfaces such as the `read` and `write` system calls to operate on sockets as well as pipes, devices, and regular files. In Linux, the *Virtual Filesystem* is used to handle these operations in an object oriented manner. When a file operation is performed the VFS determines the appropriate subsystem to send the actual request to.

The VFS is described by *Understanding the Linux Kernel* but only brief mention is made of "sockfs" – the pseudofilesystem used by the socket interface to handle file operations on sockets. In this paper I will provide more details about *sockfs* and how VFS uses it to call the appropriate socket functions when system calls such as *read*, *write*, *close*, and so on are called on a file descriptor associated with a socket. To match *Understanding the Linux Kernel* all examples here are for Linux Kernel version 2.6.11 on the i386 architecture. We will assume the reader has a basic familiarity with VFS as covered by *Understanding the Linux Kernel*.

The key concept to both VFS and the `sockfs` mechanism is function pointers. By implementing filesystem and socket operations using functions called via a structure representing a common interface, lower level code can be "pluggable." This construct enables VFS to call `sockfs` functions when a system call is passed a socket file descriptor. It also enables `sockfs` to call the correct functions in the lower level networking code, no matter protocol is associated with the socket. An example of this is seen in Figure 6. This table shows the three IPv4 `proto_ops` structures, used to determine which function to call in order to satisfy a request. This technique is common throughout the kernel, especially in the VFS and networking systems.

2 Initialization

The first area of interest is initialization. This is performed when the system is booted, provided networking is enabled. These steps create the `sockfs` pseudofilesystem and hook it into the VFS layer.

```
struct socket_alloc {
        struct socket socket;
        struct inode vfs_inode;
};
```

Figure 1: `struct socket_alloc defined in include/net/sock.h`

When the kernel is booted, the networking subsystem is initialized. This is performed by the `sock_init` function from `net/socket.c`. This function is called from the `do_basic_init` routine which is called at boot time just prior to starting the *init* process. If the kernel has been compiled without networking, the `net/nonet.c` file is compiled and linked instead of `socket.c`. This causes a dummy stub of `sock_init` to be called. `nonet.c` supplies a basic `struct file_operations` which causes `open` to return `-ENXIO` ("No such device or address") thus preventing any further usage of sockets.

Nearly all Linux kernels are configured with networking, so the `sock_init` function from `net/socket.c` is more commonly used. This function begins by initializing SLAB caches. The first is for `struct sock` objects. The `struct sock` structure is defined in `include/net/sock.h` and is the internal representation of a socket as used throughout the network layer. If `SLAB_SKB` is defined, an additional SLAB cache is created for `struct sk_buff` objects. These are buffers used for socket data. Note that `SLAB_SKB` is defined in `include/linux/skbuff.h`, so the skbuff SLAB cache is always created in Linux 2.6.11.

Next, `init_inodecache` is called. This function initializes a SLAB cache for `struct socket_alloc` objects (Figure 1). In this case, `kmem_cache_create` is passed a constructor argument - `init_once` from `socket.c`. This constructor function is called whenever a `struct socket_alloc` object is allocated from the SLAB cache. The constructor function simply calls `inode_init_once` on the `vfs_inode` member of the `struct socket_alloc` (Figure 1). This function is part of common inode initialization code and simply initializes the fields of a `struct inode`.

The next step is to register the *sockfs* filesystem using the VFS function `register_filesystem`, defined in `fs/filesystems.c`. This accepts a pointer to `struct file_system_type` as an argument. See Figure 2. This structure includes the name of the filesystem ("sockfs") and functions to create and destroy the `struct super_block` object which contains information about the filesystem. The `sockfs_get_sb` function calls `get_sb_pseudo` which is used for the pseudofilesystems such as *sockfs* which cannot be mounted. The `kill_anon_super` is the counterpart for destroying these pseudofilesystem super blocks.

The `get_sb_pseudo` generic function takes as an argument a pointer to `struct super_operations` which contains the super block operations that *sockfs* supports. See Figure 3. These operations allow creation and destruction of *sockfs* inode objects. These will be described in greater detail in sections §3 and §5. Additionally, the `simple_statfs` function from `fs/libfs.c` is used to provide a basic implementation for the VFS statfs functionality.

```
static struct file_system_type sock_fs_type = {
        .name =         "sockfs",
        .get_sb =       sockfs_get_sb,
        .kill_sb =      kill_anon_super,
};
```

Figure 2: `static struct file_system_type sock_fs_type` defined in `net/socket.c`

```
static struct super_operations sockfs_ops = {
        .alloc_inode =  sock_alloc_inode,
        .destroy_inode =sock_destroy_inode,
        .statfs =       simple_statfs,
};
```

Figure 3: `static struct super_operations sockfs_ops` defined in `net/socket.c`

Finally, the `kern_mount` function is called which "mounts" the pseudofilesystem (of course, it does not have a filesystem mount point!). The `struct vfsmount` pointer returned by `kern_mount` is assigned to the static pointer `sock_mnt`. If `CONFIG_NETFILTER` (packet filtering) is enabled, a final call to `netfilter_init` is performed, but netfilter is beyond the scope of this document.

3 Socket Creation

This section follows the process of creating a socket and associated file descriptor, which will be returned to the user. This process involves creating the socket then mapping a file descriptor to it. The file descriptor also has a related file structure which is reachable from the current Task Control Block. After this process is complete the socket data is reachable from the file descriptor (via several structures of indirection).

The `socket` system call is the primary method used to construct a socket from user space. However, this is not the only method by which sockets are created. The `accept` system call is used to accept connections on a listening socket. It blocks until a new connection is received, then returns a new file descriptor for the new connection. The listening socket file descriptor is then available for additional calls to accept. Another socket construction system call is `socketpair` which creates a pair of connected sockets[1]. These system calls are entered via the `sys_socketcall` mechanism[2].

[1] On Linux, these must be `AF_UNIX` or `AF_LOCAL` family sockets. These are used for inter-process communication using the "Unix Domain Sockets" method.

[2] For details on the socket call mechanism on i386 and `sys_socketcall` please see my earlier paper, "`sys_socketcall`: Network systems calls on Linux".

The sys_socket system call and sys_socketpair calls follow a similar structure. The code used by sys_accept is somewhat different and will be discussed later. Both sys_socket and sys_socketpair take arguments for the socket type and protocol family. These will be used later to determine which lower level functions will be called. These arguments are passed down until they are needed. For the sake of simplicity, I will not mention them explicitly until they are used.

The sys_socket and sys_socketpair functions each call sock_create to create a socket then sock_map_fd to assign a file descriptor to it. The function sys_socketpair obviously does this twice, and there is an additional step which will be covered in section §4.

The sock_create function is a simple wrapper for the __sock_create function. The last argument of the underlying function is a boolean specifying whether it came from sock_create or sock_create_kern which has certain security implications[3]. The __sock_create function begins by sanity checking the family and type arguments. Next, a check is made for a deprecated configuration: PF_INET and SOCK_PACKET. A warning is printed to the console if the deprecated configuration is used and the family argument is modified to the updated PF_PACKET type. A static flag is used to limit the warning to one print per boot.

Next, a check if performed to see if the requested protocol is supported. If the kernel was configured with loadable modules (CONFIG_KMOD) an attempt is made to load a module for the requested protocol. Then the net_family_read_lock function is called to acquire the lock. If there is no support for the requested protocol at this point (whether or not the module autoload was attempted) EAFNOSUPPORT is returned.

Now sock_alloc is called in order to obtain a struct socket. This function calls the VFS function new_inode with the super block pointer from the vfsmount structure sock_mnt (see §2). Note that now the sock_alloc_inode operation set up during initialization is called (Figure 3). This function allocates a struct socket_alloc (Figure 1) and initializes the fields in the contained struct socket structure, then returns the address of the contained vfs_inode. The sock_alloc function then uses the SOCKET_I function to obtain the struct socket associated with the inode, via the containing structure. Some more initialization is performed and the struct socket pointer is returned.

Next some tricks are performed with the module access functions to ensure their reference count is kept accurate. The protocol family parameter discussed earlier is used as an index into the net_families array to find the appropriate create function and call it. For AF_INET, the function is inet_create[4]. This

[3]This is used by the call to security_socket_create, which is a no-op unless CONFIG_SECURITY_NETWORK is enabled. It provides a hook for security modules such as SELinux. The security hooks are complex and beyond the scope here, so I won't discuss them in detail.

[4]Protocol initialization code does not manipulate this array directly, but instead calls sock_register. Note that in the aforementioned case of a module loaded on demand this function is called as the module loads, so the array is filled out just before using it. IPv4 support is not typically built as a module, but most distributions choose to include less

```
static struct file_operations socket_file_ops = {
        .owner =        THIS_MODULE,
        .llseek =       no_llseek,
        .aio_read =     sock_aio_read,
        .aio_write =    sock_aio_write,
        .poll =         sock_poll,
        .unlocked_ioctl = sock_ioctl,
        .mmap =         sock_mmap,
        .open =         sock_no_open,
        .release =      sock_close,
        .fasync =       sock_fasync,
        .readv =        sock_readv,
        .writev =       sock_writev,
        .sendpage =     sock_sendpage
};
```

Figure 4: `socket_file_ops` defined in `include/net/sock.h`

performs protocol specific initialization, which is beyond the scope of this article. The return value of the `create` is passed back to the caller of `sock_create`.

Next the `sock_map_fd` function is called. This function takes the `struct socket` obtained previously and returns a file descriptor that references the socket. The first step is to call `get_unused_fd` to obtain an unused file descriptor number. If this succeeds, a `struct file` is next obtained from `get_empty_filp`. A name parameter is created for passing to `d_alloc` (dentry creation). The name parameter consists of simply the inode number in brackets. At this point `d_alloc` is called to obtain the dentry which is populated with the address of the static `sockfs_dentry_operations` structure. The sole assigned member of this structure is the `sockfs_delete_dentry` function which just returns 1.

The other item of significance is the static `socket_file_ops` structure. The address of this structure is assigned to both the inode `i_fop` table as well as the newly created file's `f_op` table. This structure is described in Figure 4. As you can see, this structure contains function pointers for each of the operations required by *sockfs*.

Finally, `fd_install` is called to associate the newly created file object with the file descriptor. Note the following, from the comment above `sock_map_fd`:

> *Note that another thread may close file descriptor before we return from this function. We use the fact that now we do not refer to socket after mapping. If one day we will need it, this function will increment ref. 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 kernel, but we take care of internal coherence yet.*

---

common protocols as modules.

This is not an issue. Just after calling `sock_map_fd` both `sys_socket` and `sys_socketpair` finish and return the file descriptors to user space without performing additional operations.

As previously stated, `sys_accept` is somewhat more complicated. The first step is to look up the listening socket's file descriptor using `sockfd_lookup`. This function obtains the `struct socket` associated with a file descriptor and will be described further in section §4. If this is successful, `sock_alloc` is called as previously described. The new socket is set to have the same type and protocol specific operations table as the listening socket.

Next `__module_get` is called to increment the protocol module's reference count. Note that it isn't necessary to go through the additional steps of trying to load the module - it must already be loaded since the listening socket is of the same protocol. The accept implementation from the protocol specific operations table (See Figure 6 for an IPv4 example of this table) is now called in order to perform the actual connection accept. If it succeeds, and the user passed a valid `struct sockaddr` structure, the protocol specific getname function is called in order to determine the peer's address, which is moved to user space using the `move_addr_to_user` socket helper function. The last thing before returning is to call `sockfd_put`, which decrements the reference count (incremented due to a call to `fget` inside `sockfd_lookup`).

4 Socket Operations

At this point we will trace the execution of a call to `read` through the VFS layer all the way to the protocol specific code. For this example we're assume the file descriptor passed to `read` corresponds to an IPv4 TCP socket, arguably the most common case on Linux systems today. Figure 5 provides a high level overview of the calls and structure dereferences that happen in this use case. Function calls are represented by gray arrows and structure lookups by shaded arrows.

Calls to other functions are similar and generally follow the same structure. It should be noted that some of the entries seen in Figure 4 are not working implementations but stubs which return an error because the operations do not make sense in the *sockfs* context. The `no_llseek` stub is defined in `fs/read_write.c` and returns `ESPIPE` ("Invalid seek"). The `sock_no_open` stub returns `ENXIO` ("No such device or address").

Our trace of the read system call begins with `sys_read` in `fs/read_write.c`. The first step here is to call `fget_light`, which finds the `struct file` object associated with a given file descriptor. This function performs an optimization by checking the `count` field of the open files structure in the current Task Control Block. If only one task is using it, then the more complicated locking is not needed. Otherwise, `get_file` is called to increment the reference count in the file structure, and a subsequent call to `fput` will be needed later. The `fput_needed` flag is set in this case.

If `fget_light` returns success, then the current file position is obtained using

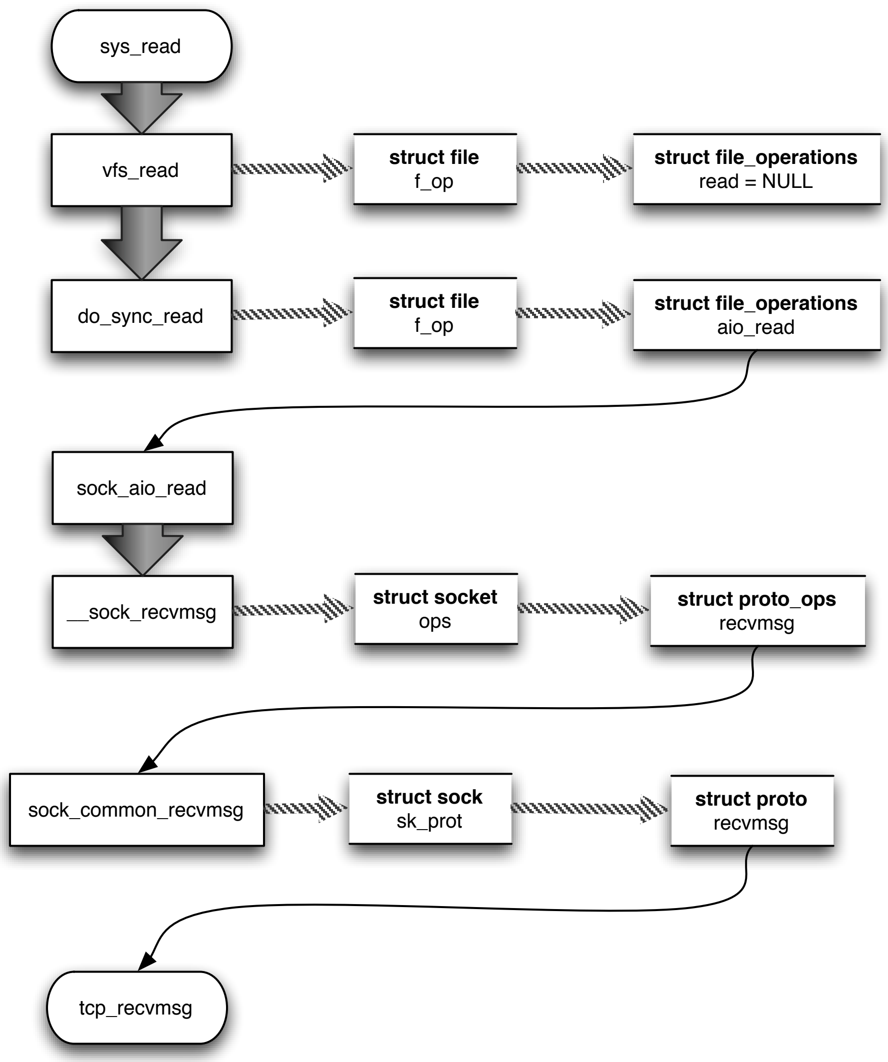

Figure 5: Flow diagram of a call to read on a file descriptor corresponding to an `AF_INET` TCP socket.

proto_ops field	inet_stream_ops	inet_dgram_ops	inet_sockraw_ops
family	PF_INET	PF_INET	PF_INET
owner	THIS_MODULE	THIS_MODULE	THIS_MODULE
release	inet_release	inet_release	inet_release
bind	inet_bind	inet_bind	inet_bind
connect	inet_stream_connect	inet_dgram_connect	inet_dgram_connect
socketpair	sock_no_socketpair	sock_no_socketpair	sock_no_socketpair
accept	inet_accept	sock_no_accept	sock_no_accept
getname	inet_getname	inet_getname	inet_getname
poll	tcp_poll	udp_poll	dgram_poll
ioctl	inet_ioctl	inet_ioctl	inet_ioctl
listen	inet_listen	sock_no_listen	sock_no_listen
shutdown	inet_shutdown	sock_shutdown	sock_shutdown
setsockopt	sock_common_setsockopt	sock_common_setsockopt	sock_common_setsockopt
getsockopt	sock_common_getsockopt	sock_common_getsockopt	sock_common_getsockopt
sendmsg	inet_sendmsg	inet_sendmsg	inet_sendmsg
recvmsg	sock_common_recvmsg	sock_common_recvmsg	sock_common_recvmsg
mmap	sock_no_mmap	sock_no_mmap	sock_no_mmap
sendpage	tcp_sendpage	inet_sendpage	inet_sendpage

Figure 6: Example `proto_ops` structures from `af_net.c`

`file_pos_read`. The position for sockets was initialized to zero during the setup phase. Next, a call is made to `vfs_read`, which accepts a pointer to the position as an argument. Once this returns the (potentially modified) position is written back using `file_pos_write`. As we'll see in a minute, for sockets the position remains at 0. Finally, `fput_light` is called. The `fput_needed` flag is used to determine if a call to `fput` is actually needed. If it is not, `fput_light` is a no-op.

The real work happens inside `vfs_read`. This function begins by checking the file's mode. If the file was not opened for reading then `EBADF` is immediately returned. Next, the `vfs_read` function checks that the file object has a valid `f_op` pointer and that the `f_op` table (see Figure 4) contains a mapping for at least one of read or aio_read (asynchronous IO read). If this condition is not satisfied then `EINVAL` is returned, indicating that read is not a valid operation on this file descriptor.

The next step is to perform access verification. The `access_ok` macro is used to verify that the user has access to the buffer. Then the `rw_verify_area` function is called, which performs sanity checks on the position and count arguments, and checks any file locking, if present (which it never is in the case of *sockfs*). Finally, a call to `security_file_permission` is performed (this is another hook for Linux security models active only if `CONFIG_SECURITY` has been enabled). If this point is passed then the file descriptor has been cleared for the read operation.

As noted before, either the read operation or the aio_read operation is sufficient to provide an implementation of read. If the read operation is present it is called directly. Note that *sockfs* only supports the aio_read operation (Figure 4). This means an additional VFS function `do_sync_read` must be called.

The `do_sync_read` function emulates the functionality of a standard blocking read using the asynchronous read operations. This allows filesystems to implement only the asynchronous functionality and have the simpler blocking read automatically available. The function simply sets up the kiocb structure required for an asynchronous read, then calls the aio_read operation from the f_op table. If the aio_read operation returns -EIOCBQUEUED, indicating a queued asynchronous operation, wait_on_sync_kiocb from fs_aio.c is called. This function sits in a loop in TASK_UNINTERRUPTIBLE, waiting for ki_users to become zero. Each time through the loop schedule is called in order to relinquish the CPU.

Once the asynchronous IO operation completes, control returns to vfs_read. If the read operation was successful, dnotify_parent is called to mark this file as accessed, and in the current Task Control Block (TCB) the read IO counter is updated with the number of bytes read. Independent of success or failure, the read system calls counter is incremented in the current TCB.

The actual sock_aio_read function performs some basic sanity checks on the pos and size arguments, then sets up the structures required for the asynchronous IO transfer. A pointer to the struct socket is pulled out of the iocb's file pointer's dentry's inode using the SOCKET_I function which gets the containing structure then returns a pointer to the socket member. Finally, __sock_recvmsg is called to do the actual work. This function looks up the protocol recvmsg function in the struct proto_ops structure.

The recvmsg function pointer for IPv4 TCP sockets is sock_common_recvmsg, which looks up recvmsg in the struct proto structure, obtained through the sock structure (a field in the more generic socket structure). This function is tcp_recvmsg, which finally does the actual protocol work to receive a message.

Implementation of the write call is essentially identical, and others are similar. One notable exception is the sock_ioctl function. This consists of a long switch statement. Many of the ioctls can be handled entirely by the abstract interface. For those that cannot be handled in sock_ioctl, the default case in the switch statement calls the protocol specific ioctl. This strategy simplifies writing of protocol specific functions by keeping the generalizable functionality central.

5 Socket Destruction

Without the ability to close sockets it would not be long before our servers ran out of file descriptors. So, we must bring things to a close. The VFS close call is a perfect example of how VFS makes things easier for the user space programmer. A simple call to close will clean up file descriptor resources, whether the descriptor represents a socket, pipe, or regular file,

Strangely, sys_close is defined in fs/open.c. The first step taken is to acquire the file_lock spinlock. Some checks are done to verify the file descriptor argument is valid, then the struct file pointer is retrieved from the table of file descriptors. If the retrieved pointer is valid, the corresponding entry in the current TCB's file descriptor table is set to NULL, and the file descriptor number is marked as free. At this point the file_lock spinlock can be released.

Next sys_close calls filp_close. This function clears any outstanding errors on the file and then checks and prints if the file already has a zero file count (which is a bug, since it indicates the file should already have been released). If the file object supports the flush operation, it is performed. However, this does not apply to *sockfs*, as there is no defined flush operation. Whether or not the flush operation succeeds, `dnotify_flush` is called to free any dnotify resources associated with this file. The next function called is `locks_remove_posix`, which cleans up resources related to file locking. Again, this will do nothing in the case of sockets.

The final step in `filp_close` is to call `fput` which releases the file resource. If this is the last task holding this file open, the `f_count` reaches zero and the atomic decrement and test operation returns true. This causes a call to `__fputc`, which performs the final cleanup for the file. This begins with calls to release eventpoll and flock objects. Finally, the `release` function from the f_op structure is called. This is defined in the *sockfs* file operations, so the `sock_release` function is called.

Inside `sock_release` the cleanup is straightforward. The first step is to determine the module owning this socket resource, so `module_put` can be called. At this point the protocol specific release routine is also called, enabling protocol specific resource cleanup. Next, the code verifies no asynchronous operations are currently in progress, and logs a message if they are. The CPU specific variable `sockets_in_use` is then decremented, and the associated inode is released using `iput` function[5].

At this point note that there is no explicit release of the SLAB cache resources allocated during the `sock_alloc` and `sock_alloc_inode`. These resources are freed when they are no longer used (their reference counts go to zero) at the time of the last "put" operation. The callback `sock_destroy_inode` was set up as part of the super operations structure (Figure 3). When the reference count (atomically decremented and checked) reaches zero `sock_destroy_inode` is called. This function calls `kmem_cache_free` which finally returns the `struct socket_alloc` to the SLAB cache.

We talked about mounting the `sockfs` filesystem in Section §2, but in fact there is no way to unmount `sockfs`. Since the generic networking support and `sockfs` cannot be built as a module, it is not necessary to provide an unmount function. The `sockfs` pseudofilesystem is always mounted from boot to shutdown.

I hope this has been a reasonable introduction to *sockfs* and the complex nature of the Linux Virtual Filesystem layer. While the VFS is complex, it vastly simplifies kernel programming for both filesystems and pseudofilesystems such as `sockfs`. The Berkeley Sockets API has succeeded in part because it uses the familiar programming model shared with all sorts of Unix I/O. By unifying these operations, both the kernel and user space layers are simplified.

[5]This performs the now familiar reference count decrement, and cleans up resources if the reference count goes to zero