NetBSD Internals

(2011/07/21)

The NetBSD Developers

NetBSD Internals
by The NetBSD Developers

Published 2011/07/21 17:27:57

Table of Contents

List of Tables

Purpose of this book

This book describes the NetBSD Operating System internals. The main idea behind it is to provide solid documentation for contributors that wish to develop extensions for NetBSD or want to improve its existing code. Ideally, there should be no need to reverse-engineer the system's code in order to understand how something works.

A lot of work is still required to finish this book: some chapters are not finished and some are not even started. Those parts that are planned but which are still pending to do are already part of the book but are clearly marked as incomplete by using a `XXX` marker.

This book is currently maintained by the NetBSD www team (<`www@NetBSD.org`>). Corrections, suggestions and extensions should be sent to that address.

Chapter 1

Memory management

XXX: This chapter is extremely incomplete. It currently contains supporting documentation for Chapter 2 but nothing else.

1.1 The UVM virtual memory manager

UVM is the NetBSD's virtual memory manager.

1.1.1 UVM objects

An UVM object — or also known as *uobj* — is a contiguous region of virtual memory backed by a specific system facility. This can be a file (vnode), XXX What else?.

In order to understand what "to be backed by" means, here is a review of some basic concepts of virtual memory management. In a system with virtual memory support, the system can manage an address space bigger than the physical amount of memory available to it. The address space is broken into chunks of fixed size, namely *pages*, as is the physical memory, which is divided into *page frames*.

When the system needs to access a memory address, it can either find the page it belongs to (page hit) or not (page fault). In the former case, the page is already stored in main memory so its data can be directly accessed. In the latter case, the page is not present in main memory.

When a page fault occurs, the processor's memory management unit (MMU) signals the kernel through an exception and asks it to handle the fault: this can either result in a resolved page fault or in an error. Assuming that all memory accesses are correct (and hence there are no errors), the kernel needs to bring the requested page into memory. But where is the requested page? Is it in the swap space? In a file? Should it be filled with zeros?

Here is where the backing mechanism enters the game. A backing object defines where the pages should be read from and where shall them be stored after modifications, if any. Talking about implementation, reading a page from the backing object is preformed by a getpages function while writing to it is done by a putpages one.

Example: consider a 32-bit address space, a page size of 4096 bytes and an uobj of 40960 bytes (10 pages) starting at the virtual address 0x00010000; this uobj's backing object is a vnode that represents a text file in your file system. Assume that the file has not been read at all yet, so none of its pages are in main memory. Now, the user requests a read from offset 5000 and with a length of 4000. This offset falls into the uobj's second page and the ending address (9000) falls into the third page. The kernel converts these logical offsets into memory addresses (0x00011388 and 0x00012328) and reads all the data contained in between. So what happens? The MMU causes two page faults and the vnode's getpages method is called for each of them, which then reads the pages from the corresponding file, puts them into main memory and returns control to the caller. At this point, the read has been served.

Similarly, pages can be modified in memory after they have been brought to it; at some point, these changes will need to be flushed to the backing store, which happens with the backing object's putpages operation. There are multiple reasons for the flush, including the need to reclaim the least recently used page frame from main memory, explicitly synchronizing the uobj with its backing store (think about synchronizing a file system), closing a file, etc.

1.2 Managing wired memory

The malloc(9) and free(9) functions provided by the NetBSD kernel are very similar to their userland counterparts. They are used to allocate and release wired memory, respectively.

1.2.1 Malloc types

Malloc types are used to group different allocation blocks into logical clusters so that the kernel can manage them in a more efficient manner.

A malloc type can be defined in a static or dynamic fashion. Types are defined statically when they are embedded in a piece of code that is linked together the kernel during build time; if they are part of a standalone module, they are defined dynamically.

For static declarations, the MALLOC_DEFINE(9) macro is provided, which is then used somewhere in the global scope of a source file. It has the following signature:

```
MALLOC_DEFINE(struct malloc_type *type, const char *short_desc, const char
*long_desc);
```

The first parameter takes the name of the malloc type to be defined; do not let the type shown above confuse you, because it is an internal detail you ought not know. Malloc types are often named in uppercase, prefixed by `M_`. Some examples include `M_TEMP` for temporary data, `M_SOFTINTR` for soft-interrupt structures, etc.

The second and third parameters are a character string describing the type; the former is a short description while the later provides a longer one.

For a dynamic declaration, you must first define the type as static within the source file. Later on, the malloc_type_attach(9) and malloc_type_detach(9) functions are used to notify the kernel about the presence or removal of the type; this is usually done in the module's initialization and finalization routines, respectively.

Chapter 2
File system internals

This chapter describes in great detail the concepts behind file system development under NetBSD. It presents some code examples under the name of egfs, a fictitious file system that stands for *example file system*.

Throughout this chapter, the word *file* is used to refer to *any* kind of object that may exist in a file system; this includes directories, regular files, symbolic links, special devices and named pipes. If there is a need to mention a file that stores data, the term *regular file* will be used explicitly.

Understanding a complex body of code like the storage subsystem can be difficult. This chapter begins with a structural overview, explaining how specific file systems and the virtual file system (VFS) code interact. It continues with a description of both the *vnode* interface (the interface to files; Section 2.2) and the *VFS* interface (the interface to whole file systems; Section 2.3) and then summarizes the existing file systems. These sections should be read in order; they provide a general outline for the whole storage subsystem and a foundation for reading and understanding existing code.

The subsequent sections of this chapter dig into specific issues and constructs in detail. These sections may be read in any order, and are heavily cross-linked to one another to ease navigation. These later sections should be considered a reference guide rather than an introduction.

At the very end there is a section that summarizes, based on ready-to-copy-and-paste code examples, how to write a file system driver from scratch. Note that this section does not contain explanations per se but only links to the appropriate sections where each point is described.

2.1 Structural overview

The storage subsystem is divided into four basic parts. First and highest level is the VFS-level code, file system independent code that performs common functions on behalf of the rest of the kernel. This portion sits on top of the second part, the individual file systems. The third part, common or generic implementations of file system level logic, sits at the same conceptual level as file systems themselves but is file system independent and shared rather than being part of a single file system. The fourth part is lower level support code that file systems call into. This code is also file system independent. (A fifth portion, device drivers for storage buses and hardware, is not discussed in this chapter.)

The interface between the VFS-level code and the file systems is very clearly defined. It is made up of two parts, the *vnode interface* and the *VFS interface*, described in more detail in the next two sections. The other interfaces are much less clear, as is the upper interface that the VFS-level code provides to the system call layer and the rest of the kernel. Work is ongoing to clarify these interfaces.

Confusingly, the VFS-level code, the combination of the VFS and vnode interfaces, and the VFS interface alone are all sometimes referred to as "the VFS layer".

2.2 vnode interface overview

A vnode is an abstract representation of an active file within the NetBSD kernel; it provides a generic way to operate on the real file it represents regardless of the file system it lives on. Thanks to this abstraction layer, all kernel subsystems only deal with vnodes. It is important to note that there is a *unique vnode for each active file*.

A vnode is described by the struct vnode structure; its definition can be found in the `src/sys/sys/vnode.h` file and information about its fields is available in the vnode(9) manual page. The following analyzes the most important ideas related to this structure.

As the rule says, abstract representations must be specialized before they can be instantiated. vnodes are not an exception: each file system extends both the static and dynamic parts of an vnode as follows:

- The static part — the data fields that represent the object — is extended by attaching a custom data structure to an vnode instance during its creation. This is done through the `v_data` field as described in Section 2.2.1.
- The dynamic part — the operations applicable to the object — is extended by attaching a vnode operations vector to a vnode instance during its creation. This is done through the `v_op` field as described in Section 2.2.3.

2.2.1 The vnode data field

The `v_data` field in the struct vnode type is a pointer to an external data structure used to represent a file within a concrete file system. This field must be initialized after allocating a new vnode and must be set to `NULL` before releasing it (see Section 2.8.4).

This external data structure contains any additional information to describe a specific file inside a file system. In an on-disk file system, this might include the file's initial cluster, its creation time, its size, etc. As an example, the NetBSD's Fast File System (FFS) uses the in-core memory representation of an inode as the vnode's data field.

2.2.2 vnode operations

A vnode operation is implemented by a function that follows the following contract: return an integer describing the operation's exit status and take a single void * parameter that carries a structure with the real operation's arguments.

Using an external structure to describe the operation's arguments instead of using a regular argument list has a reason: some file systems extend the vnode with additional, non-standard operations; having a common prototype makes this possible.

The following table summarizes the standard vnode operations. Keep in mind, though, that each file system is free to extend this set as it wishes. Also note that the operation's name is shown in the table as the macro used to call it (see Section 2.2.4).

Table 2-1. vnode operations summary

Operation	Description	See also

Operation	Description	See also
VOP_LOOKUP	Performs a path name lookup.	See Section 2.10.
VOP_CREATE	Creates a new file.	See Section 2.11.1.
VOP_MKNOD	Creates a new special file (a device or a named pipe).	See Section 2.14.
VOP_LINK	Creates a new hard link for a file.	See Section 2.11.2.
VOP_RENAME	Renames a file.	See Section 2.11.4.
VOP_REMOVE	Removes a file.	See Section 2.11.3.
VOP_OPEN	Opens a file.	
VOP_CLOSE	Closes a file.	
VOP_ACCESS	Checks access permissions on a file.	See Section 2.11.8.
VOP_GETATTR	Gets a file's attributes.	See Section 2.11.6.1.
VOP_SETATTR	Sets a file's attributes.	See Section 2.11.6.2.
VOP_READ	Reads a chunk of data from a file.	See Section 2.11.5.4.
VOP_WRITE	Writes a chunk of data to a file.	See Section 2.11.5.4.
VOP_IOCTL	Performs an ioctl(2) on a file.	
VOP_FCNTL	Performs a fcntl(2) on a file.	
VOP_POLL	Performs a poll(2) on a file.	
VOP_KQFILTER	XXX	
VOP_REVOKE	Revoke access to a vnode and all aliases.	
VOP_MMAP	Maps a file on a memory region.	See Section 2.11.5.3.
VOP_FSYNC	Synchronizes the file with on-disk contents.	
VOP_SEEK	Test and inform file system of seek	
VOP_MKDIR	Creates a new directory.	See Section 2.13.1.
VOP_RMDIR	Removes a directory.	See Section 2.13.2.
VOP_READDIR	Reads directory entries from a directory.	See Section 2.13.3.
VOP_SYMLINK	Creates a new symbolic link for a file.	See Section 2.12.1.
VOP_READLINK	Reads the contents of a symbolic link.	See Section 2.12.2.
VOP_TRUNCATE	Truncates a file.	See Section 2.11.6.2.
VOP_UPDATE	Updates a file's times.	See Section 2.11.7.
VOP_ABORTOP	Aborts an in-progress operation.	
VOP_INACTIVE	Marks the vnode as inactive.	See Section 2.8.1.
VOP_RECLAIM	Reclaims the vnode.	See Section 2.8.1.

Operation	Description	See also
VOP_LOCK	Locks the vnode.	See Section 2.8.5.
VOP_UNLOCK	Unlocks the vnode.	See Section 2.8.5.
VOP_ISLOCKED	Checks whether the vnode is locked or not.	See Section 2.8.5.
VOP_BMAP	Maps a logical block number to a physical block number.	See Section 2.11.5.5.
VOP_STRATEGY	Performs a file transfer between the file system's backing store and memory.	See Section 2.11.5.5.
VOP_PATHCONF	Returns pathconf(2) information.	
VOP_ADVLOCK	XXX	
VOP_BWRITE	Writes a system buffer.	
VOP_GETPAGES	Reads memory pages from the file.	See Section 2.11.5.2.
VOP_PUTPAGES	Writes memory pages to the file.	See Section 2.11.5.2.

2.2.3 The vnode operations vector

The `v_op` field in the struct vnode type is a pointer to the vnode operations vector, which maps logical operations to real functions (as seen in Section 2.2.2). This vector is file system specific as the actions taken by each operation depend heavily on the file system where the file resides (consider reading a file, setting its attributes, etc.).

As an example, consider the following snippet; it defines the `open` operation and retrieves two parameters from its arguments structure:

```
int
egfs_open(void *v)
{
        struct vnode *vp = ((struct vop_open_args *)v)->a_vp;
        int mode = ((struct vop_open_args *)v)->a_mode;

        ...
}
```

The whole set of vnode operations defined by the file system is added to a vector of struct vnodeopv_entry_desc-type entries, with each entry being the description of a single operation. The purpose of this vector is to define a mapping from logical operations such as `vop_open` or `vop_read` to real functions such as `egfs_open`, `egfs_read`. *It is not directly used by the system* under normal operation. This vector is not tied to a specific layout: it only lists operations available in the file system it describes, in any order it wishes. It can even list non-standard (and unknown) operations as well as lack some of the most basic ones. (The reason is, again, extensibility by third parties.)

There are two minor restrictions, though:

- The first item always points to an operation used in case a non-existent one is called. For example, if the file system does not implement the `vop_bmap` operation but some code calls it, the call will be redirected to this default-catch function. As such, it is often used to provide a generic error routine but it is also useful in different scenarios. E.g., layered file systems use it to pass the call down the stack.

  It is important to note that there are two standard error routines available that implement this functionality: `vn_default_error` and `genfs_eopnotsupp`. The latter correctly cleans up vnode references and locks while the former is the traditional error case one. New code should only use the former.

- The last item always is a pair of null pointers.

Consider the following vector as an example:

```
const struct vnodeopv_entry_desc egfs_vnodeop_entries[] = {
        { vop_default_desc, vn_default_error },
        { vop_open_desc, egfs_open },
        { vop_read_desc, egfs_read },
        ... more operations here ...
        { NULL, NULL }
};
```

As stated above, this vector is not directly used by the system; in fact, it only serves to construct a secondary vector that follows strict ordering rules. This secondary vector is automatically generated by the kernel during file system initialization, so the code only needs to instruct it to do the conversion.

This secondary vector is defined as a pointer to an array of function pointers of type int (**vops)(void *). To tell the kernel where this vector is, a mapping between the two vectors is established through a third vector of struct vnodeopv_desc-type items. This is easier to understand with an example:

```
int (**egfs_vnodeop_p)(void *);
const struct vnodeopv_desc egfs_vnodeop_opv_desc =
        { &egfs_vnodeop_p, egfs_vnodeop_entries };
```

Out of the file-system's scope, users of the vnode layer will only deal with the `egfs_vnodeop_p` and `egfs_vnodeop_opv_desc` vectors.

2.2.4 Executing vnode operations

All vnode operations are subject to a very strict locking protocol among several other call and return contracts. Furthermore, their prototype makes their call rather complex (remember that they receive a structure with the real arguments). These are some of the reasons why they cannot be called directly (with a few exceptions that will not be discussed here).

The NetBSD kernel provides a set of macros and functions that make the execution of vnode operations trivial; please note that they are the standard call procedure. These macros are named after the operation they refer to, all in uppercase, prefixed by the `VOP_`string. Then, they take the list of arguments that will be passed to them.

For example, consider the following implementation for the access operation:

```
int
egfs_access(void *v)
```

```
{
        struct vnode *vp = ((struct vop_access_args *)v)->a_vp;
        int mode = ((struct vop_access_args *)v)->a_mode;
        struct ucred *cred = ((struct vop_access_args *)v)->a_cred;
        struct proc *p = ((struct vop_access_args *)v)->a_p;

        ...
}
```

A call to the previous method could look like this:

```
result = VOP_ACCESS(vp, mode, cred, p);
```

For more information, see the vnodeops(9) manual page, which describes all the mappings between vnode operations and their corresponding macros.

2.3 VFS interface overview

The kernel's Virtual File System (VFS) subsystem provides access to all available file systems in an abstract fashion, just as vnodes do with active files. Each file system is described by a list of well-defined operations that can be applied to it together with a data structure that keeps its status.

2.3.1 The mount structure

File systems are attached to the virtual directory tree by means of mount points. A mount point is a redirection from a specific directory[1] to a different file system's root directory and is represented by the generic struct mount type, which is defined in `src/sys/sys/mount.h`.

A file system extends the static part of a struct mount object by attaching a custom data structure to its `mnt_data` field. As with vnodes, this happens when allocating the structure.

The kind of information that a file system stores in its mount structure heavily depends on its implementation. Generally, it will typically include a pointer (either physical or logical) to the file system's root node, used as the starting point for further accesses. It may also include several accounting variables as well as other information whose context is the whole file system attached to a mount point.

2.3.2 VFS operations

A file system driver exposes a well-known interface to the kernel by means of a set of public operations. The following table summarizes them all; note that they are sorted according to the order that they take in the VFS operations vector (see Section 2.3.3).

Table 2-2. VFS operations summary

Operation	Description	Considerations	See also
`fs_mount`	Mounts a new instance of the file system.	Must be defined.	See Section 2.6.

Operation	Description	Considerations	See also
fs_start	Makes the file system operational.	Must be defined.	
fs_unmount	Unmounts an instance of the file system.	Must be defined.	See Section 2.6.
fs_root	Gets the file system root vnode.	Must be defined.	See Section 2.9.
fs_quotactl	Queries or modifies space quotas.	Must be defined.	
fs_statvfs	Gets file system statistics.	Must be defined.	See Section 2.7.
fs_sync	Flushes file system buffers.	Must be defined.	
fs_vget	Gets a vnode from a file identifier.	Must be defined.	See Section 2.8.3.
fs_fhtovp	Converts a NFS file handle to a vnode.	Must be defined.	See Section 2.15.
fs_vptofh	Converts a vnode to a NFS file handle.	Must be defined.	See Section 2.15.
fs_init	Initializes the file system driver.	Must be defined.	See Section 2.5.
fs_reinit	Reinitializes the file system driver.	May be undefined (i.e., null).	See Section 2.5.
fs_done	Finalizes the file system driver.	Must be defined.	See Section 2.5.
fs_mountroot	Mounts an instance of the file system as the root file system.	May be undefined (i.e., null).	
fs_extattrctl	Controls extended attributes.	The generic vfs_stdextattrctl function is provided as a simple hook for file systems that do not support this operation.	

The list of VFS operations may eventually change. When that happens, the kernel version number is bumped.

2.3.3 The VFS operations structure

Regardless of mount points, a file system provides a struct vfsops structure as defined in `src/sys/sys/mount.h` that describes itself type is. Basically, it contains:

- A public identifier, usually named after the file system's name suffixed by the `fs` string. As this identifier is used in multiple places — and specially both in kernel space and in userland —, it is typically defined as a macro in `src/sys/sys/mount.h`. For example: `#define MOUNT_EGFS "egfs"`.

- A set of function pointers to file system operations. As opposed to vnode operations, VFS ones have different prototypes because the set of possible VFS operations is well known and cannot be extended by third party file systems. Please see Section 2.3.2 for more details on the exact contents of this vector.
- A pointer to a null-terminated vector of struct vnodeopv_desc * const items. These objects are listed here because, as stated in Section 2.2.3, the system uses them to construct the real vnode operations vectors upon file system startup.

  It is interesting to note that this field may contain more than one pointer. Some file systems may provide more than a single set of vnode operations; e.g., a vector for the normal operations, another one for operations related to named pipes and another one for operations that act on special devices. See the FFS code for an example of this and Section 2.14 for details on these special vectors.

Consider the following code snipped that illustrates the previous items:

```
const struct vnodeopv_desc * const egfs_vnodeopv_descs[] = {
        &egfs_vnodeop_opv_desc,
        ... more pointers may appear here ...
        NULL
};

struct vfsops egfs_vfsops = {
        MOUNT_EGFS,
        egfs_mount,
        egfs_start,
        egfs_unmount,
        egfs_root,
        egfs_quotactl,
        egfs_statvfs,
        egfs_sync,
        egfs_vget,
        egfs_fhtovp,
        egfs_vptofh,
        egfs_init,
        NULL, /* fs_reinit: optional */
        egfs_done,
        NULL, /* fs_mountroot: optional */
        vfs_stdextattrctl,
        egfs_vnodeopv_descs
};
```

The kernel needs to know where each instance of this structure is located in order to keep track of the live file systems. For file systems built inside the kernel's core, the `VFS_ATTACH` macro adds the given VFS operations structure to the appropriate link set. See GNU ld's info manual for more details on this feature.

```
VFS_ATTACH(egfs_vfsops);
```

Standalone file system modules need not do this because the kernel will explicitly get a pointer to the information structure after the module is loaded.

2.4 File systems overview

2.4.1 On-disk file systems

On-disk file systems are those that store their contents on a physical drive.

- Fast File System (ffs): XXX
- Log-structured File System (lfs): XXX
- Extended 2 File System (ext2fs): XXX
- FAT (msdosfs): XXX
- ISO 9660 (cd9660): XXX
- NTFS (ntfs): XXX

2.4.2 Network file systems

- Network File System (nfs): XXX
- Coda (codafs): XXX

2.4.3 Synthetic file systems

- Memory File System (mfs): XXX
- Kernel File System (kernfs): XXX
- Portal File System (portalfs): XXX
- Pseudo-terminal File System (ptyfs): XXX
- Temporary File System (tmpfs): XXX

2.4.4 Layered file systems

- Null File System (nullfs): XXX
- Union File System (unionfs): XXX
- User-map File System (umapfs): XXX

2.4.5 Helper file systems

Helper file systems are just a set of functions used to easily implement other file systems. As such, they can be considered as libraries. These are:

- fifofs: Implements all operations used to deal with named pipes in a file system.
- genfs: Implements generic operations shared across multiple file systems.
- layerfs: Implements generic operations shared across layered file systems (see Section 2.4.4).
- specfs: Implements all operations used to deal with special files in a file system.

2.5 Initialization and cleanup

Drivers often have an initialization routine and a finalization one, called when the driver becomes active (e.g., at system startup) or inactive (e.g., unloading its module) respectively. File systems are subject to these rules too, so that they can do global tasks as a whole, regardless of any mount point.

These initialization and finalization tasks can be done from the `fs_init` and `fs_done` hooks, respectively. If the driver is provided as a module, the initialization routine is called when it is loaded and the cleanup function is executed when it is unloaded. Instead, if it is built into the kernel, the initialization code is executed at very early stages of kernel boot but *the cleanup stuff is never run*, not even when the system is shut down.

Furthermore, the `fs_reinit` operation is provided to... XXX...

These three operations take the following prototypes:

```
int fs_init(void);
```

```
int fs_reinit(void);
```

```
int fs_done(void);
```

Note how they do not take any parameter, not even a mount point.

As an example, consider the following functions that deal with a malloc type (see Section 1.2.1) defined for a specific file system:

```
MALLOC_JUSTDEFINE(M_EGFSMNT, "egfs mount", "egfs mount structures");

void
egfs_init(void)
{

        malloc_type_attach(M_EGFSMNT);

        ...
}

void
```

```
egfs_done(void)
{

        ...

        malloc_type_detach(M_EGFSMNT);
}
```

2.6 Mounting and unmounting

The mount operation, namely `fs_mount`, is probably the most complex one in the VFS layer. Its purpose is to set up a new mount point based on the arguments received from userland. Basically, it receives the mount point it is operating on and a data structure that describes the mount call parameters.

Unfortunately, this operation has been overloaded with some semantics that do not really belong to it. More specifically, it is also in charge of updating the mount point parameters as well as fetching them from userland. This ought to be cleaned up at some point.

We will see all these details in the following subsections.

2.6.1 Mount call arguments

Most file systems pass information from the userland mount utility to the kernel when a new mount point is set up; this information generally includes user-tunable properties that tell the kernel how to mount the file system. This data set is encapsulated in what is known as the mount arguments structure and is often named after the file system, prepending the `_args` string to it.

Keep in mind that this structure is only used to communicate the userland and the kernel. Once the call that passes the information finishes, it is discarded in the kernel side.

The arguments structure is versioned to make sure that the kernel and the userland always use the same field layout and size. This is achieved by inserting a field at the very beginning of the object, holding its version.

For example, imagine a virtual file system — one that is not stored on disk; for real (and very similar) code, you can look at tmpfs. Its mount arguments structure could describe the ownership of the root directory or the maximum number of files that the file system may hold:

```
#define EGFS_ARGSVERSION 1
struct egfs_args {
        int ea_version;

        off_t ea_size_max;

        uid_t ea_root_uid;
        gid_t ea_root_gid;
        mode_t ea_root_mode;

        ...
}
```

2.6.2 The mount utility

XXX: To be written. Slightly describe how a userland mount utility works.

2.6.3 The fs_mount operation

The `fs_mount` operation is called whenever a user issues a mount command from userland. It has the following prototype:

```
int vfs_mount(struct mount *mp, const char *path, void *data, struct
nameidata *ndp, struct proc *p);
```

The caller, which is always the kernel, sets up a struct mount object and passes it to this routine through the `mp` parameter. It also passes the mount arguments structure (as seen in Section 2.6.1) in the `data` parameter. There are several other arguments, but they do not important at this point.

The `mp->mnt_flag` field indicates what needs to be done (remember that this operation is semantically overloaded). The following is an outline of all the tasks this function does and also describes the possible flags for the `mnt_flag` field:

1. If the `MNT_GETARGS` flag is set in `mp->mnt_flag`, the operation returns the current mount parameters for the given mount point.

   This is further detailed in Section 2.6.3.1.

2. Copy the mount arguments structure from userland to kernel space using copyin(9).

   This is further detailed in Section 2.6.3.2.

3. If the `MNT_UPDATE` flag is set in `mp->mnt_flag`, the operation updates the current mount parameters of the given mount point based on the new arguments given (e.g., upgrade to read-write from read-only mode).

   This is further detailed in Section 2.6.3.3.

4. At this point, if neither `MNT_GETARGS` nor `MNT_UPDATE` were set, the operation sets up a new mount point.

   This is further detailed in Section 2.6.3.4.

2.6.3.1 Retrieving mount parameters

When the `fs_mount` operation is called with the `MNT_GETARGS` flag in `mp->mnt_flag`, the routine creates and fills the mount arguments structure based on the data of the given mount point and returns it to userland by using copyout(9).

This heavily depends on the file system, but consider the following simple example:

```
if (mp->mnt_flag & MNT_GETARGS) {
        struct egfs_args args;
        struct egfs_mount *emp;

        if (mp->mnt_data == NULL)
```

```
                return EIO;
        emp = (struct egfs_mount *)mp->mnt_data;

        args.ea_version = EGFS_ARGSVERSION;

        ... fill the args structure here ...

        return copyout(&args, data, sizeof(args));
}
```

2.6.3.2 Getting the arguments structure

The `data` argument given to the `fs_mount` operation points to a memory region in user-space. Therefore, it must be first copied into kernel-space by means of copyin(9) to be able to access it in a safe fashion.

Here is a little example:

```
int error;
struct egfs_args args;

if (data == NULL)
        return EINVAL;

error = copyin(data, &args, sizeof(args));
if (error)
        return error;

if (args.ea_version != EGFS_ARGSVERSION)
        return EINVAL;
```

2.6.3.3 Updating mount parameters

When the `fs_mount` operation is called with the `MNT_UPDATE` flag in `mp->mnt_flag`, the routine modifies the current parameters of the given mount point based on the new parameters given in the mount arguments structure.

2.6.3.4 Setting up a new mount point

If neither `MNT_GETARGS` nor `MNT_UPDATE` were set in `mp->mnt_flag` when calling `fs_mount`, the operation sets up a new mount point. In other words: it fills the struct mount object given in `mp` with correct data.

The very first thing that it usually does is to allocate a structure that defines the mount point. This structure is named after the file system, appending the `_mount` string to it, and is often very similar to the mount arguments structure. Once allocated and filled with appropriate data, the object is attached to the mount point by means of its `mnt_data` field.

Later on, the operation gets a file system identifier for the mount point being set up using the vfs_getnewfsid(9) function and assigns.

At last, it sets up any statvfs-related information for the mount point by using the `set_statvfs_info` function.

This is all clearer by looking at a simple code example:

```
emp = (struct egfs_mount *)malloc(sizeof(struct egfs_mount), M_EGFSMOUNT, M_WAITOK);
KASSERT(emp != NULL);

/* Fill the emp structure with file system dependent values. */
emp->em_root_uid = args.ea_rood_uid;
... more comes here ...

mp->mnt_data = emp;
mp->mnt_flag = MNT_LOCAL;
mp->mnt_stat.f_namemax = MAXNAMLEN;
vfs_getnewfsid(mp);

return set_statvfs_info(path, UIO_USERSPACE, args.ea_fspec, UIO_SYSSPACE, mp, p);
```

2.6.4 The vfs_unmount function

Unmounting a file system is often easier than mounting it, plus there is no need to write a file system dependent userland utility to do an unmount. This is accomplished by the `fs_unmount` operation, which has the following signature:

```
int fs_unmount(struct mount *mp, int mntflags, struct proc *p);
```

The function's outline is similar to the following:

1. Ask the kernel to finalize all pending I/O on the given mount point. This is done through the vflush(9) function. Note that its last argument is a flags bitfield which must carry the `FORCECLOSE` flag if the file system is being forcibly unmounted — in other words, if the `MNT_FORCE` flag was set in `mntflags`.
2. Free all resources attached to the mount point — i.e., to the mount structure pointed to by `mp->mnt_data`. This heavily depends on the file system internals.
3. Destroy the file system specific mount structure and detach it from the `mp` mount point.

Here is a simple example of the previous outline:

```
int error, flags;
struct egfs_mount *emp;

flags = (mntflags & MNT_FORCE) ? FORCECLOSE : 0;

error = vflush(mp, NULL, flags);
if (error != 0)
        return error;
```

```
emp = (struct egfs_mount *)mp->mnt_data;
... free emp contents here ...

free(mp->mnt_data, M_EGFSMNT);
mp->mnt_data = NULL;

return 0;
```

2.7 File system statistics

The statvfs(2) system call is used to retrieve statistical information about a mounted file system, such as its block size, number of used blocks, etc. This is implemented in the file system driver by the `fs_statvfs` operation whose prototype is:

```
int fs_statvfs(struct mount *mp, struct statvfs *sbp, struct proc *p);
```

The execution flow of this operation is quite simple: it basically fills `sbp`'s fields with appropriate data. This data is derivable from the current status of the file system — e.g., through the contents of `mp->mnt_data`.

It is interesting to note that some of the information returned by this operation is stored in the generic part of the `mp` structure, shared across all file systems. The `copy_statvfs_info` function takes care to copy this common information into the resulting structure with minimum efforts. Among other things, it copies the file system's identifier, the number of writes, the maximum length of file names, etc.

As a general rule of thumb, the code in `fs_statvfs` manually initializes the following fields in the `sbp` structure: `f_iosize`, `f_frsize`, `f_bsize`, `f_blocks`, `f_bavail`, `f_bfree`, `f_bresvd`, `f_files`, `f_favail`, `f_ffree` and `f_fresvd`. Details information about each field can be found in statvfs(2).

For example, the operation's content may look like:

```
... fill sbp's fields as described above ...

copy_statvfs_info(sbp, mp);

return 0;
```

2.8 vnode management

2.8.1 vnode's life cycle

A vnode, like any other system object, has to be allocated before it can be used. Similarly, it has to be released and deallocated when unused. Things are a bit special when it comes to handling a vnode, hence this whole section dedicated to explain it.

XXX: A graph could be excellent to have at this point.

A vnode is first brought to life by the getnewvnode(9) function; this returns a clean vnode that can be used to represent a file. This new vnode is also marked as *used* and remains as such until it is marked inactive. A vnode is inactivated by calling releasing the last reference to it. When this happens, `VOP_INACTIVE` is called for the vnode and the vnode is placed on the free list.

The *free list*, despite its confusing name, contains real, live, but not currently used vnodes. It is like a big LRU list. vnodes can be brought to life again from this list by using the vget(9) function, and when that happens, they leave the free list and are marked as used again until they are inactivated. Why does this list exist, anyway? For example, think about all the commands that need to do path lookups on `/usr`. Anything in `/usr/bin`, `/usr/sbin`, `/usr/pkg/bin` and `/usr/pkg/sbin` will need the `/usr` vnode. If it had to be regenerated from scratch each time, it could be slow. Therefore, it is kept around on the free list.

vnodes on the free list can also be `reclaimed` which means that they are effectively killed. This can either happen because the vnode is being reused for a new vnode (through `getnewvnode`) or because it is being shut down (e.g., due to a revoke(2)).

Note that the `kern.maxvnodes` sysctl(9) node specifies how many vnodes can be kept active at a time.

2.8.2 vnode tags

vnodes are tagged to identify their type. The tag attached to them must not be used within the kernel; it is only provided to let userland applications (such as pstat(8)) to print information about vnodes.

Note that its usage is deprecated because it is not extensible from dynamically loadable modules. However, since they are currently used, each file system defines a tag to describe its own vnodes. These tags can be found in `src/sys/sys/vnode.h` and vnode(9).

2.8.3 Allocation of a vnode

vnodes are allocated in three different scenarios:

- Access to existing files: the kernel does a file name lookup as described in Section 2.10.2. When the vnode lookup operation finds a match, it allocates a vnode for the chosen file and returns it to the system.
- Creation of a new file: the file system specific code allocates a new vnode after the successful creation of the new file and returns it to the file system generic code. This can happen as a result of the vnode create, mkdir, mknod and symlink operations.
- Access to a file through a NFS file handle: when the file system is asked to convert an NFS file handle to a vnode through the fhtovp vnode operation, it may need to allocate a new vnode to represent the file. See Section 2.15.

It is important to recall that vnodes are unique per file. Special care is taken to avoid allocating more than one vnode for a single physical file. Each file system has its own method to achieve this; as an example, tmpfs keeps a map between file system nodes and vnodes, where the former are its keys.

However, please do note that there may be files with no in-core representation (i.e., no vnode). Only active and inactive but not-yet-reclaimed files are represented by a vnode.

A simple example that illustrates vnode allocation can be found in the `tmpfs_alloc_vp` function of `src/sys/fs/tmpfs/tmpfs_subr.c`.

XXX: I think fs_vget has to be described in this section.

2.8.4 Deallocation of a vnode

The procedure to deallocate vnodes is usually trivial: it generally cleans up any file system specific information that may be attached to the vnode.

Keep in mind that there is *a single place* in the code where vnodes can be detached from their underlying nodes and destroyed. This place is in the vnode reclaim operation. Doing it from any other place will surely cause further trouble because the vnode may still be active or reusable (see Section 2.8.1).

Note that the `v_data` pointer must be set to null before exiting the reclaim vnode operation or the system will complain because the vnode was not properly cleaned.

This function is also in charge of releasing the underlying real node, if needed. For example, when a file is deleted the corresponding vnode operation is executed — be it a delete or a rmdir — but the vnode is not released until it is reclaimed. This means that if the real node was deleted before this happened, the vnode would be left pointing to an invalid memory area.

Consider the following sample operation:

```
int
egfs_reclaim(void *v)
{
        struct vnode *vp = ((struct vop_reclaim_args *)v)->a_vp;

        struct egfs_node *node;

        node = (struct egfs_node *)vp->v_data;

        cache_purge(vp);
        vp->v_data = NULL;
        node->en_vnode = NULL;

        if (node->en_nlinks == 0)
                ... free the underlying node ...

        return 0;
}
```

However, keep in mind that releasing (marking it inactive) a vnode is not the same as reclaiming it. The real reclaiming will often happen at a much later time, unless explicitly requested. The operations that remove files from disk often execute the reclaim code on purpose so that the vnode and its associated disk space is released as soon as possible. This can be done by using the vrecycle(9) function.

As an example:

```
int
egfs_inactive(void *v)
{
        struct vnode *vp = ((struct vop_inactive_args *)v)->a_vp;
```

```
        struct egfs_node *node;

        node = (struct egfs_node *)vp->v_data;

        if (node->en_nlinks == 0) {
                /* The file was deleted from the disk; reclaim it as
                 * soon as possible to free its physical space. */
                vrecycle(vp, NULL, p);
        }

        return 0;
}
```

2.8.5 vnode's locking protocol

vnodes have, as almost all other system objects, a locking protocol associated to them to avoid access interferences and deadlocks. These may arise in two scenarios:

- In uniprocessor systems: a vnode operation returns before the operation is complete, thus having to lock the vnode to prevent unrelated modifications until the operation finishes. This happens because most file systems are asynchronous.

  For example: the read operation prepares a read to a file, launches it, puts the process requesting the read to sleep and yields execution to another process. Some time later, the disk responds with the requested data, returning it to the original process, which is awoken. The system must ensure that while the process was sleeping, the vnode suffers no changes.
- In multiprocessor systems: two different CPUs want to access the same file at the same time, thus needing to pass through the same vnode to reach it. Furthermore, the same problems that appear in uniprocessor systems can also appear here.

Each vnode operation has a specific locking contract it must comply to, which is often different from other operations (this makes the protocol very complex and ought to be simplified). These contracts are described in vnode(9) and vnodeops(9). You can also find them in the form of assertions in tmpfs' code, should you want to see them expressed in logical notation.

As regards vnode operations, each file system implements locking primitives in the vnode layer. These primitives allow to lock a vnode (`vop_lock`), unlock it (`vop_unlock`) and test whether it is locked or not (`vop_islocked`). Given that these operations are common to all file systems, the genfs pseudo-file system provides a set of functions that can be used instead of having to write custom ones. These are `genfs_lock`, `genfs_unlock` and `genfs_islocked` and are always used except for very rare cases.

It is very important to note that *`vop_lock` is never used directly*. Instead, the vn_lock(9) function is used to lock vnodes. Unlocking, however, is in charge of `vop_unlock`.

2.9 The root vnode

As described in Section 2.10, the kernel does all path name lookups in an iterative way. This means that in order to reach any file within a mount point, it must first traverse the mount point itself. In other words, the mount point is the only place through which the system can access a file system and thus it must be able to resolve it.

In order to accomplish this, each file system provides the `fs_root` hook which returns a vnode representing its root node. The prototype for this function is:

```
int fs_root(struct mount *mp, struct vnode **vpp);
```

2.10 Path name resolution procedure

XXX Write an introduction.

2.10.1 Path name components

A path name component is a non-divisible part of a complete path name — one that does not contain the slash (`/`) character. Any path name that includes one or more slashes in it can be divided in two or more different atoms.

Path name components are represented by struct componentname objects (defined in `src/sys/sys/namei.h`), heavily used by several vnode operations. The following are its most important fields:

- `cn_flags`: A bitfield that describes the element. Of special interest is the `HASBUF` flag, which indicates that this object holds a valid path name buffer (see the `cn_pnbuf` field below).
- `cn_pnbuf`: A pointer to the buffer holding the complete path name. This is only valid if the `cn_flags` bitfield has the `HASBUF` flag.

  In most situations, this buffer is automatically allocated and deallocated by the system, but this is not always true. Sometimes, it is necessary to free it in some of the vnode operations themselves; vnodeops(9) gives more details about this.
- `cn_nameptr`: A pointer within `cn_pnbuf` that specifies the start of the path name component described by this object. Must *always* be used in conjunction with `cn_namelen`.
- `cn_namelen`: The length of this path name component, starting at `cn_nameptr`.

2.10.2 The lookup algorithm

To *resolve a path name* (or to *lookup a path name*) means to get a vnode that uniquely represents based on a previously specified path name, be it absolute or relative.

The NetBSD kernel uses a two-level iterative algorithm to resolve path names. The first level is file system independent and is carried on by the namei(9) function, while the second one relies on internal file system details and is achieved through the lookup vnode operation.

The following list illustrates the lookup algorithm. Lots of details have been left out from it to make things simpler; namei(9) and vnodeops(9) contain all the missing information:

XXX: <wrstuden> I think you simplified the description too much. You left out lookup(), and ascribe certain actions to namei() when they are performed by lookup(). While I like your attempt to keep it simple, I think both namei() and lookup() need describing. lookup() takes a path name and turns it into a vnode, and namei() takes the result and handles symbolic link resolution.

XXX: <jmmv> I currently don't know very much about the internals of lookup() and namei(), so I've left the simplified description in the document, temporarily.

1. `namei` constructs a `cnp` path name component (of type struct componentname as described in Section 2.10.1); its buffer holds the complete path name to look for. The component pointers are adjusted to describe the path name's first component.
2. The `namei` operation gets the vnode for the lookup's starting point (always a directory). For absolute path names, this is the root directory's vnode. For relative path names, it is the current working directory's vnode, as seen by the calling userland process.

   This vnode is generally called `dvp`, standing for *directory vnode pointer*.
3. `namei` calls the vnode lookup operation on the `dvp` vnode, telling it which is the component it has to resolve (`cnp`) starting from the given directory.
4. If the component exists in the directory, the vnode lookup operation must return a vnode for its respective entry.

   However, if the component does not exist in the directory, the lookup will fail returning an appropriate error code. There are several other error conditions that have to be reported, all of them appropriately described in vnodeops(9).
5. `namei` updates `dvp` to point to the returned vnode and advances `cnp` to the next component, only if there are more components to look for. In that case, the procedure continues from 3.

   In case there are no more components to look for, `namei` returns the vnode of the last entry it located.

There are several reasons behind this two-level lookup mechanism, but they have been left over for simplicity. XXX: The 4.4BSD book gives them all; we should either link to it or explain these here in our own words (preferably the latter).

2.10.3 Lookup hints

One of the arguments passed to the lookup algorithm is a hint that specifies the kind of lookup to execute. This hint specifies whether the lookup is for a file creation (`CREATE`), a deletion (`DELETE`) or a name change (`RENAME`). The file system uses these hints to speed up the corresponding operation — generally to cache some values that will be used while processing the real operation later on.

For example, consider the unlink(2) system call whose purpose is to delete the given file name. This operation issues a lookup to ensure that the file exists and to get a vnode for it. This way, it is able to call the vnode's remove operation. So far, so good. Now, the operation itself has to delete the file, but removing a file means, among other things, detaching it from the directory containing it. How can the remove operation access the directory entry that pointed to the file being removed? Obviously, it can do another lookup and traverse a potentially long directory. But is this really needed?

Remember that unlink(2) first got a vnode for the entry to be removed. This implied doing a lookup, which traversed the file's parent directory looking for its entry. The algorithm reached the entry once, so there is no need to repeat the process once we are in the vnode operation itself.

In the above situation, the second lookup is avoided by caching the affected directory entry while the lookup operation is executed. This is only done when the `DELETE` hint is given.

The same situation arises with file creations (because new entries may be overwrite previously deleted entries in on-disk file systems) or name changes (because the operation needs to modify the associated directory entry).

2.11 File management

XXX: Write an introduction.

2.11.1 Creation of regular files

XXX: To be written. Describe vop_create.

2.11.2 Creation of hard links

XXX: To be written. Describe vop_link.

2.11.3 Removal of a file

XXX: To be written. Describe vop_remove.

2.11.4 Rename of a file

XXX: To be written. Describe vop_rename.

2.11.5 Reading and writing

vnodes have an operation to read data from them (`vop_read`) and one to write data to them (`vop_write`) both called by their respective system calls, read(2) and write(2). The read operation receives an offset from which the read starts, a number that specifies the number of bytes to read (length) and a buffer into which the data will be stored. Similarly, the write operation receives an offset from which the write starts, the number of bytes to write and a buffer from which the data is read.

There is also the mmap(2) system call which maps a file into memory and provides userland direct access to the mapped memory region.

2.11.5.1 uio objects

The struct uio type describes a data transfer between two different buffers. One of them is stored within the uio object while the other one is external (often living in userland space). These objects are created when a new data transfer starts and are alive until the transfer finishes completely; in other words, they identify a specific transfer.

The following is a description of the most important fields in struct uio (the ones needed for basic understanding on how it works). For a complete list, see uiomove(9).

- `uio_offset`: The offset within the file from which the transfer starts. If the transfer is a read, the offset must be within the file size limits; if it is a write, it can extend beyond the end of the file — in which case the file is extended.
- `uio_resid` (also known as the *residual count*): Number of bytes remaining to be transferred for this object.
- A set of pointers to buffers into/from which the data will be read/written. These are not used directly and hence their names have been left out.
- A flag that indicates if data should be read from or written to the buffers described by the uio object.

This may be easier to understand by discussing a little example. Consider the following userland program:

```
char buffer[1024];
lseek(fd, 100, SEEK_SET);
read(fd, buffer, 1024);
```

The read(2) system call constructs an uio object containing an offset of 100 bytes and a residual count of 1024 bytes, making the uio's buffers point to `buffer` and marking them as the data's target. If this was a write operation, the uio object's buffers could be the data's source.

In order to simplify uio object management, the kernel provides the uiomove(9) function, whose signature is:

```
int uiomove(void *buf, size_tn, struct uio *uio);
```

This function copies up to `n` bytes between the kernel buffer pointed to by `buf` into the addresses described by the `uio` instance. If the transfer is successful, the uio object is updated so that `uio_resid` is decremented by the amount of data copied, `uio_offset` is increased by the same amount and the internal buffer pointers are updated accordingly. This eases calling `uiomove` repeatedly (e.g., from within a loop) until the transfer is complete.

2.11.5.2 Getting and putting pages

As seen in Section 2.11.5.1, data transfers are described by a high-level object that does not take into account any detail of the underlying file system. More specifically, they are not tied to any specific on-disk block organization. (Remember that most on-disk file systems store data scattered across the disk (due to fragmentation); therefore, the transfers have to be broken up into pieces to read or write the data from the appropriate disk blocks.)

Breaking the transfer into pieces, requesting them to the disk and handling the results is a (very) complex operation. Fortunately, the UVM memory subsystem (see Section 1.1) simplifies the whole task. Each vnode has a struct uvm_object (as described in Section 1.1.1) associated to it, backed by a vnode.

The vnode backs up the uobj through its `vop_getpages` and `vop_putpages` operations. As these two operations are very generic (from the point of view of managing memory pages), genfs provides two generic functions to implement them. These are `genfs_getpages` and `genfs_putpages`, which will usually suit the needs of any on-disk file system. How they deal with specific file system details is something detailed in Section 2.11.5.5.

2.11.5.3 Memory-mapping a file

Thanks to the particular UBC implementation in NetBSD (see Section 2.11.5.2), a file can be trivially mapped into memory. The mmap(2) system call is used to achieve this and the kernel handles it independently from the file system.

The `VOP_MMAP` method is used to only inform the file system that the vnode is about to be memory-mapped and ask the file system if it allows the mapping to happen.

After the file is memory-mapped, file system I/O is handled by UVM through the vnode pager and ends up in `vop_getpages` and `vop_putpages`. In a sense this is very much like regular reading and writing, but instead of explicitly calling `vop_read` and `vop_write`, which then use `uiomove`, the memory window is accessed directly.

2.11.5.4 The read and write operations

Thanks to the particular UBC implementation in NetBSD (see Section 2.11.5.2), the vnode's read and write operations (`vop_read` and `vop_write` respectively) are very simple because they only deal with virtual memory. Basically, all they need to do is memory-map the affected part of the file and then issue a simple memory copy operation.

As an example, consider the following sample read code:

```
int
egfs_read(void *v)
{
        struct vnode *vp = ((struct vop_read_args *)v)->a_vp;
        struct uio *uio = ((struct vop_read_args *)v)->a_uio;

        int error;
        struct egfs_node *node;

        node = (struct egfs_node *)vp->v_data;

        if (uio->uio_offset < 0)
                return EINVAL;

        if (uio->uio_resid == 0 || uio->uio_offset >= node->en_size)
                return 0;

        if (vp->v_type == VREG) {
                error = 0;
```

```
                while (uio->uio_resid > 0 && error == 0) {
                        int flags;
                        off_t len;
                        void *win;

                        len = MIN(uio->uio_resid, node->en_size -
                            uio->uio_offset);
                        if (len == 0)
                                break;

                        win = ubc_alloc(&vp->v_uobj, uio->uio_offset,
                            &len, UBC_READ);
                        error = uiomove(win, len, uio);
                        flags = UBC_WANT_UNMAP(vp) ? UBC_UNMAP : 0;
                        ubc_release(win, flags);
                }
        } else {
                ... left out for simplicity (if needed) ...
        }

        return error;
}
```

2.11.5.5 Reading and writing pages

As seen in Section 2.11.5.2, the `genfs_getpages` and `genfs_putpages` functions are enough for most on-disk file systems. But if they are abstract, how do they deal with the specific details of each file system? E.g., if the system wants to fetch the third page of the `/foo/bar` file, how does it know which on-disk blocks it must read to bring the requested page into memory? Where does the real transfer take place?

The mapping between memory pages and disk blocks is done by the vnode's bmap operation, `vop_bmap`, called by the paging functions. This receives the file's logical block number to be accessed and converts it to the internal, file system specific block number.

Once bmap returns the physical block number to be accessed, the generic page handling functions check whether the block is already in memory or not. If it is not, a transfer is done by using the vnode's strategy operation (`vop_strategy`).

More information about these operations can be found in the vnodeops(9) manual page.

2.11.6 Attributes management

Within the NetBSD kernel, a file has a set of standard and well-known attributes associated to it. These are:

- A type: specifies whether the file is a regular file (`VREG`), a directory (`VDIR`), a symbolic link (`VLNK`), a special device (`VCHR` or `VBLK`), a named pipe (`VFIFO`) or a socket (`VSOCK`). The constants mentioned here are the vnode types, which do not necessarily match the internal type representation of a file within a file system.

- An ownership: that is, a user id and a group id.
- An access mode.
- A set of flags: these include the immutable flag, the append-only flag, the archived flag, the opaque flag and the nodump flag. See chflags(2) for more information.
- A hard link count.
- A set of times: these include the birth time, the change time, the access time and the modification time. See Section 2.11.7 for more details.
- A size: the exact size of the file, in bytes.
- A device number: in case of a special device (character or block ones), its number is also stored.

The NetBSD kernel uses the struct vattr type (detailed in vattr(9)) to handle all these attributes all in a compact way. Based on this set, each file system typically supports these attributes in its node representation structure (unless they are fictitious and faked when accessed). For example, FFS could store them in inodes, while FAT could save only some of them and fake the others at run time (such as the ownership).

A struct vattr instance is initialized by using the `VATTR_NULL` macro, which sets its vnode type to `VNON` and all of its other fields to `VNOVAL`, indicating that they have no valid values. After using this macro, it is the responsibility of the caller to set all the fields it wants to the correct values. The consumer of the object shall not use those fields whose value is unset (`VNOVAL`).

It is interesting to note that there are no vnode operations that match the regular system calls used to set the file ownership, its mode, etc. Instead, nodes provide two operations that act on the whole attribute set: `vop_getattrs` to read them and `vop_setattrs` to set them. The rest of this section describes them.

2.11.6.1 Getting file attributes

The `vop_getattr` vnode operation fetches all the standard attributes from a given vnode. All it does is fill the given `struct vattr` structure with the correct values. For example:

```
int
egfs_getattr(void *v)
{
        struct vnode *vp = ((struct vop_getattr_args *)v)->a_vp;
        struct vattr *vap = ((struct vop_getattr_args *)v)->a_vap;

        struct egfs_node *node;

        node = (struct egfs_node *)vp->v_data;

        VATTR_NULL(vap);

        switch (node->en_type) {
        case EGFS_NODE_DIR:
                vap->va_type = VDIR;
                break;
        case ...:
```

```
        ...
        }
        vap->va_mode = node->en_mode;
        vap->va_uid = node->en_uid;
        vap->va_gid = node->en_gid;
        vap->va_nlink = node->en_nlink;
        vap->va_flags = node->en_flags;
        vap->va_size = node->en_size;
        ... continue filling values ...

        return 0;
}
```

2.11.6.2 Setting file attributes

Similarly to the `vop_getattr` operation, `vop_setattr` sets a subset of file attributes at once. Only those attributes which are not `VNOVAL` are changed. Furthermore, the operation ensures that the caller is not trying to set unsettable values; for example, one cannot set (i.e., change) the file type.

Of special interest is that the file's size can be changed as an attribute. In other words, this operation is the entry point for file truncation calls and it is its responsibility to call `vop_truncate` when appropriate. The system never calls the vnode's truncate operation directly.

A little sketch:

```
int
egfs_setattr(void *v)
{
        struct vnode *vp = ((struct vop_setattr_args *)v)->a_vp;
        struct vattr *vap = ((struct vop_setattr_args *)v)->a_vap;
        struct ucred *cred = ((struct vop_setattr_args *)v)->a_cred;
        struct proc *p = ((struct vop_setattr_args *)v)->a_p;

        /* Do not allow setting unsettable values. */
        if (vap->va_type != VNON || vap->va_nlink != VNOVAL || ...)
                return EINVAL;

        if (vap->va_flags != VNOVAL) {
                ... set node flags here ...
                if error, return it
        }

        if (vap->va_size != VNOVAL) {
                ... verify file type ...
                error = VOP_TRUNCATE(vp, size, 0, cred, p);
                if error, return it
        }

        ... etcetera ...

        return 0;
}
```

2.11.7 Time management

Each node has four times associated to it, all of them represented by struct timespec objects. These times are:

- Birth time: the time the file was born. Cannot be changed after the file is created.
- Access time: the time the file was last accessed.
- Change time: the time the file's node was last changed. For example, if a new hard link for an existing file is created, its change time is updated.
- Modification time: the time the file's contents were last modified.

Given that these times reflect the last accesses to the underlying files, they need to be modified extremely often. If this was done synchronously, it could impose a big performance penalty on files accessed repeatedly. This is why time updates are done in a delayed manner.

Nodes usually have a set of flags (which are only kept in memory, never written to disk) that indicate their status to let asynchronous actions know what to do. These flags are used, among other things, to indicate that a file's times have to be updated. They are set as soon as the file is changed but the times are not really modified until the vnode's update operation (`vop_update`) is called; see vnodeops(9) for more details on this.

`vop_update` is called asynchronously by the kernel from time to time. However, a file system may opt to execute it on purpose as it wishes; such a situation may be when it is mounted synchronously, as it will be updating the times as soon as the changes happen.

2.11.8 Access control

The file system is in charge of ensuring that a request is valid or not, permission-wise. This is done with the vnode's access operation (`vop_access`), which receives the caller's credentials and the requested access mode. The operation then checks if these are compatible with the current attributes of the file being accessed.

The operation generally follows this structure:

1. If the file system is mounted read only, and the caller wants to write to a directory, to a link or to a regular file, then access must be denied.
2. If the file is immutable and the caller wants to write to it, access is denied.
3. At last, vaccess(9) is used to check all remaining access possibilities. This simplifies a lot the code of this operation.

For example:

```
int
egfs_access(void *v)
{
        struct vnode *vp = ((struct vop_access_args *)v)->a_vp;
        int mode = ((struct vop_access_args *)v)->a_mode;
        struct ucred *cred = ((struct vop_access_args *)v)->a_cred;

        struct egfs_node *node;
```

```
        node = (struct egfs_node *)vp->v_data;

        if (vp->v_type == VDIR || vp->v_type == VLNK || vp->v_type == VREG)
                if (mode & VWRITE &&
                    vp->v_mount->mnt_flag & MNT_RDONLY)
                        return EROFS;
        }

        if (mode & VWRITE && mode->tn_flags & IMMUTABLE)
                return EPERM;

        return vaccess(vp->v_type, node->en_mode, node->en_uid,
            node->en_gid, mode, cred);
}
```

2.12 Symbolic link management

2.12.1 Creation of symbolic links

XXX: To be written. Describe vop_symlink.

2.12.2 Read of symbolic link's contents

XXX: To be written. Describe vop_readlink.

2.13 Directory management

A directory maps file names to file system nodes. The internal representation of a directory depends heavily on the file system, but the vnode layer provides an abstract way to access them. This includes the `vop_lookup`, `vop_mkdir`, `vop_rmdir` and `vop_readdir` operations.

For the rest of this section, assume that the following simple struct egfs_dirent describes a directory entry:

```
struct egfs_dirent {
        char ed_name[MAXNAMLEN];
        int ed_namelen;
        off_t ed_fileid;
};
```

2.13.1 Creation of directories

XXX: To be written. Describe vop_mkdir.

2.13.2 Removal of directories

XXX: To be written. Describe vop_rmdir.

2.13.3 Reading directories

The `vop_readdir` operation reads the contents of directory in a file system independent way. Remember that the regular read operation can also be used for this purpose, though all it returns is the exact contents of the directory; this cannot be used by programs that aim to be portable (not to mention that some file systems do not support this functionality).

This operation returns a struct dirent object (as seen in dirent(5)) for each directory entry it reads from the offset it was given up to that offset plus the transfer length. Because it must read entire objects, the offset must always be aligned to a physical directory entry boundary; otherwise, the function shall return an error. This is not always true, though: some file systems have variable-sized entries and they use another metric to determine which entry to read (such as its ordering index).

It is important to note that the size of the resulting struct dirent objects is variable: it depends on the name stored in them. Therefore, the code first constructs these objects (settings all its fields by hand) and then uses the `_DIRENT_SIZE` macro to calculate its size, later assigned to the `d_reclen` field. For example:

```
struct egfs_dirent de;
struct egfs_node *node;
struct dirent d;

... read a directory entry from disk into de ...
... make node point to the de.ed_fileid node ...

switch (node->ed_type) {
case EGFS_NODE_DIR:
        d.d_type = DT_DIR;
case ...:
...
}

d.d_namlen = de.ed_namelen;
(void)memcpy(d.d_name, de.ed_name, de.ed_namelen);
d.d_name[de.ed_namelen] = '\0';
d.d_reclen = _DIRENT_SIZE(&d);
```

With this in mind, the operation also ensures that the offset is correct, locates the first entry to return and loops until it has exhausted the transmission's length. The following illustrates the process:

```
int
egfs_readdir(void *v)
{
        struct vnode *vp = ((struct vop_readdir_args *)v)->a_vp;
        struct uio *uio = ((struct vop_readdir_args *)v)->a_uio;
        int *eofflag = ((struct vop_readdir_args *)v)->a_eofflag;

        int entry_counter;
        int error;
```

```
        off_t startoff;
        struct egfs_dirent de;
        struct egfs_node *dnode;
        struct egfs_node *node;

        if (vp->v_type != VDIR)
                return ENOTDIR;

        if (uio->uio_offset % sizeof(struct egfs_dirent) > 0)
                return EINVAL;

        dnode = (struct egfs_node *)vp->v_data;

        ... read the first directory entry into de ...
        ... make node point to the de.ed_fileid node ...

        entry_counter = 0;
        startoff = uio->uio_offset;
        do {
                struct dirent d;

                ... construct d from de ...

                error = uiomove(&d, d.d_reclen, uio);

                entry_counter++;
                ... read the next directory entry into de ...
                ... make node point to the de.ed_fileid node ...
        } while (error == 0 && uio->uio_resid > 0
            && de is valid)


        /* Important: Update transfer offset to match on-disk
         * directory entries, not virtual ones. */
        uio->uio_offset = entry_counter * sizeof(egfs_dirent);

        if (eofflag != NULL)
                *eofflag = (de is invalid?);

        return error;
}
```

File systems that support NFS take some extra steps in this function. See vnodeops(9) for more details. XXX: Cookies and the eof flag should really be explained here.

2.14 Special nodes

File system that support named pipes and/or special devices implement the vnode's mknod operation (`vop_mknod`) in order to create them. This is extremely similar to `vop_create`. However, it takes some extra steps because named pipes and special devices are not like regular files: their contents are not stored in the file system and they have specific access methods. Therefore, they cannot use the file system's regular vnode operations vector.

In other words: the file system defines two additional vnode operations vectors: one for named pipes and one for special devices. Fortunately, this task is easy because the virtual fifofs (`src/sys/miscfs/fifofs`) and specfs (`src/sys/miscfs/specfs`) file systems provide generic vnode operations. In general, these vectors use all the generic operations except for a few functions.

Because the on-disk file system has to update the node's times when accessing these special files, some operations are implemented on a file system basis and later call the generic operations implemented in fifofs and specfs. This basically means that those file systems implement their own `vop_close`, `vop_read` and `vop_write` operations for named pipes and for special devices.

As a little example of such an operation:

```
int
egfs_fifo_read(void *v)
{
        struct vnode *vp = ((struct vop_read_args *)v)->a_vp;

        ((struct egfs_node *)vp->v_data)->tn_status |= TMPFS_NODE_ACCESSED;
        return VOCALL(fifo_vnodeop_p, VOFFSET(vop_read), v);
}
```

Remember that these two additional operations vectors are added to the vnode operations description structure; otherwise, they will are not initialized and therefore will not work. See Section 2.3.3.

For more sample code, consult `src/sys/fs/tmpfs/fifofs_vnops.c`, `src/sys/fs/tmpfs/fifofs_vnops.h`, `src/sys/fs/tmpfs/specfs_vnops.c` and `src/sys/fs/tmpfs/specfs_vnops.h`.

2.15 NFS support

XXX: To be written. Describe vop_fhtovp and vfs_vptofh.

2.16 Step by step file system writing

1. Create the `src/sys/fs/egfs` directory.
2. Create a minimal `src/sys/fs/egfs/files.egfs` file:

   ```
   deffs fs_egfs.h EGFS
   file fs/egfs/egfs_vfsops.c egfs
   file fs/egfs/egfs_vnops.c egfs
   ```

3. Modify `src/sys/conf/files` to include `files.egfs`. I.e., add the following line:

   ```
   include "fs/egfs/files.egfs"
   ```

4. Define the file system's name in `src/sys/sys/mount.h`. I.e., add the following line:

   ```
   #define MOUNT_EGFS "egfs"
   ```

5. Define the file system's vnode tag type.

   See Section 2.8.2.

6. Add the file system's magic number in the Linux compatibility layer, `src/sys/compat/linux/common/linux_misc.c` and `src/sys/compat/linux/common/linux_misc.h`, if applicable. Fallback to the default number if there is nothing appropriate for the file system.

7. Create a minimal `src/sys/fs/egfs/egfs_vnops.c` file that contains stubs for all vnode operations.

```
#include <sys/cdefs.h>
__KERNEL_RCSID(0, "$NetBSD: chap-file-system.xml,v 1.6 2017/01/10 11:30:31 ryoon Exp $")

#include <sys/param.h>
#include <sys/vnode.h>

#include <miscfs/genfs/genfs.h>

#define egfs_lookup genfs_eopnotsupp
#define egfs_create genfs_eopnotsupp
#define egfs_mknod genfs_eopnotsupp
#define egfs_open genfs_eopnotsupp
#define egfs_close genfs_eopnotsupp
#define egfs_access genfs_eopnotsupp
#define egfs_getattr genfs_eopnotsupp
#define egfs_setattr genfs_eopnotsupp
#define egfs_read genfs_eopnotsupp
#define egfs_write genfs_eopnotsupp
#define egfs_fcntl genfs_eopnotsupp
#define egfs_ioctl genfs_eopnotsupp
#define egfs_poll genfs_eopnotsupp
#define egfs_kqfilter genfs_eopnotsupp
#define egfs_revoke genfs_eopnotsupp
#define egfs_mmap genfs_eopnotsupp
#define egfs_fsync genfs_eopnotsupp
#define egfs_seek genfs_eopnotsupp
#define egfs_remove genfs_eopnotsupp
#define egfs_link genfs_eopnotsupp
#define egfs_rename genfs_eopnotsupp
#define egfs_mkdir genfs_eopnotsupp
#define egfs_rmdir genfs_eopnotsupp
#define egfs_symlink genfs_eopnotsupp
#define egfs_readdir genfs_eopnotsupp
#define egfs_readlink genfs_eopnotsupp
#define egfs_abortop genfs_eopnotsupp
#define egfs_inactive genfs_eopnotsupp
#define egfs_reclaim genfs_eopnotsupp
#define egfs_lock genfs_eopnotsupp
#define egfs_unlock genfs_eopnotsupp
#define egfs_bmap genfs_eopnotsupp
#define egfs_strategy genfs_eopnotsupp
#define egfs_print genfs_eopnotsupp
```

```
#define egfs_pathconf genfs_eopnotsupp
#define egfs_islocked genfs_eopnotsupp
#define egfs_advlock genfs_eopnotsupp
#define egfs_blkatoff genfs_eopnotsupp
#define egfs_valloc genfs_eopnotsupp
#define egfs_reallocblks genfs_eopnotsupp
#define egfs_vfree genfs_eopnotsupp
#define egfs_truncate genfs_eopnotsupp
#define egfs_update genfs_eopnotsupp
#define egfs_bwrite genfs_eopnotsupp
#define egfs_getpages genfs_eopnotsupp
#define egfs_putpages genfs_eopnotsupp

int (**egfs_vnodeop_p)(void *);
const struct vnodeopv_entry_desc egfs_vnodeop_entries[] = {
        { &vop_default_desc, vn_default_error },
        { &vop_lookup_desc, egfs_lookup },
        { &vop_create_desc, egfs_create },
        { &vop_mknod_desc, egfs_mknod },
        { &vop_open_desc, egfs_open },
        { &vop_close_desc, egfs_close },
        { &vop_access_desc, egfs_access },
        { &vop_getattr_desc, egfs_getattr },
        { &vop_setattr_desc, egfs_setattr },
        { &vop_read_desc, egfs_read },
        { &vop_write_desc, egfs_write },
        { &vop_ioctl_desc, egfs_ioctl },
        { &vop_fcntl_desc, egfs_fcntl },
        { &vop_poll_desc, egfs_poll },
        { &vop_kqfilter_desc, egfs_kqfilter },
        { &vop_revoke_desc, egfs_revoke },
        { &vop_mmap_desc, egfs_mmap },
        { &vop_fsync_desc, egfs_fsync },
        { &vop_seek_desc, egfs_seek },
        { &vop_remove_desc, egfs_remove },
        { &vop_link_desc, egfs_link },
        { &vop_rename_desc, egfs_rename },
        { &vop_mkdir_desc, egfs_mkdir },
        { &vop_rmdir_desc, egfs_rmdir },
        { &vop_symlink_desc, egfs_symlink },
        { &vop_readdir_desc, egfs_readdir },
        { &vop_readlink_desc, egfs_readlink },
        { &vop_abortop_desc, egfs_abortop },
        { &vop_inactive_desc, egfs_inactive },
        { &vop_reclaim_desc, egfs_reclaim },
        { &vop_lock_desc, egfs_lock },
        { &vop_unlock_desc, egfs_unlock },
        { &vop_bmap_desc, egfs_bmap },
        { &vop_strategy_desc, egfs_strategy },
        { &vop_print_desc, egfs_print },
        { &vop_islocked_desc, egfs_islocked },
        { &vop_pathconf_desc, egfs_pathconf },
        { &vop_advlock_desc, egfs_advlock },
```

```
        { &vop_blkatoff_desc, egfs_blkatoff },
        { &vop_valloc_desc, egfs_valloc },
        { &vop_reallocblks_desc, egfs_reallocblks },
        { &vop_vfree_desc, egfs_vfree },
        { &vop_truncate_desc, egfs_truncate },
        { &vop_update_desc, egfs_update },
        { &vop_bwrite_desc, egfs_bwrite },
        { &vop_getpages_desc, egfs_getpages },
        { &vop_putpages_desc, egfs_putpages },
        { NULL, NULL }
};
const struct vnodeopv_desc egfs_vnodeop_opv_desc =
        { &egfs_vnodeop_p, egfs_vnodeop_entries };
```

8. Create a minimal `src/sys/fs/egfs/egfs_vfsops.c` file that contains stubs for all VFS operations.

```
#include <sys/cdefs.h>
__KERNEL_RCSID(0, "$NetBSD: chap-file-system.xml,v 1.6 2017/01/10 11:30:31 ryoon Exp $")

#include <sys/param.h>
#include <sys/mount.h>

static int egfs_mount(struct mount *, const char *, void *,
    struct nameidata *, struct proc *);
static int egfs_start(struct mount *, int, struct proc *);
static int egfs_unmount(struct mount *, int, struct proc *);
static int egfs_root(struct mount *, struct vnode **);
static int egfs_quotactl(struct mount *, int, uid_t, void *,
    struct proc *);
static int egfs_vget(struct mount *, ino_t, struct vnode **);
static int egfs_fhtovp(struct mount *, struct fid *, struct vnode **);
static int egfs_vptofh(struct vnode *, struct fid *);
static int egfs_statvfs(struct mount *, struct statvfs *, struct proc *);
static int egfs_sync(struct mount *, int, struct ucred *, struct proc *);
static void egfs_init(void);
static void egfs_done(void);
static int egfs_checkexp(struct mount *, struct mbuf *, int *,
    struct ucred **);
static int egfs_snapshot(struct mount *, struct vnode *,
    struct timespec *);

extern const struct vnodeopv_desc egfs_vnodeop_opv_desc;

const struct vnodeopv_desc * const egfs_vnodeopv_descs[] = {
        &egfs_vnodeop_opv_desc,
        NULL,
};

struct vfsops egfs_vfsops = {
        MOUNT_EGFS,
        egfs_mount,
        egfs_start,
        egfs_unmount,
```

```
        egfs_root,
        egfs_quotactl,
        egfs_statvfs,
        egfs_sync,
        egfs_vget,
        egfs_fhtovp,
        egfs_vptofh,
        egfs_init,
        NULL, /* vfs_reinit: not yet (optional) */
        egfs_done,
        NULL, /* vfs_wassysctl: deprecated */
        NULL, /* vfs_mountroot: not yet (optional) */
        egfs_checkexp,
        egfs_snapshot,
        vfs_stdextattrctl,
        egfs_vnodeopv_descs
};
VFS_ATTACH(egfs_vfsops);

static int
egfs_mount(struct mount *mp, const char *path, void *data,
    struct nameidata *ndp, struct proc *p)
{

        return EOPNOTSUPP;
}

static int
egfs_start(struct mount *mp, int, struct proc *p)
{

        return EOPNOTSUPP;
}

static int
egfs_unmount(struct mount *mp, int, struct proc *p)
{

        return EOPNOTSUPP;
}

static int
egfs_root(struct mount *mp, struct vnode **vpp)
{

        return EOPNOTSUPP;
}

static int
egfs_quotactl(struct mount *mp, int cmd, uid_t uid, void *arg,
    struct proc *p)
{
```

```
        return EOPNOTSUPP;
}

static int
egfs_vget(struct mount *mp, ino_t ino, struct vnode **vpp)
{

        return EOPNOTSUPP;
}

static int
egfs_fhtovp(struct mount *mp, struct fid *fhp, struct vnode **vpp)
{

        return EOPNOTSUPP;
}

static int
egfs_vptofh(struct vnode *mp, struct fid *fhp)
{

        return EOPNOTSUPP;
}

static int
egfs_statvfs(struct mount *mp, struct statvfs *sbp, struct proc *p)
{

        return EOPNOTSUPP;
}

static int
egfs_sync(struct mount *mp, int waitfor, struct ucred *uc, struct proc *p)
{

        return EOPNOTSUPP;
}

static void
egfs_init(void)
{

        return EOPNOTSUPP;
}

static void
egfs_done(void)
{

        return EOPNOTSUPP;
}

static int
```

```
egfs_checkexp(struct mount *mp, struct mbuf *mb, int * wh,
    struct ucred **anon)
{

        return EOPNOTSUPP;
}

static int
egfs_snapshot(struct mount *mp, struct vnode *vp, struct timespec *ctime)
{

        return EOPNOTSUPP;
}
```

9. Define a new malloc type for the file system and modify the `egfs_init` and `egfs_done` hooks to attach and detach it in the LKM case.

   See Section 2.5.

10. Create the `src/sys/fs/egfs/egfs.h` file, that will define all the structures needed for our file system.

```
#if !defined(_EGFS_H_)
#  define _EGFS_H_
#else
#  error "egfs.h cannot be included multiple times."
#endif

#if defined(_KERNEL)

struct egfs_mount {
        ...
};

struct egfs_node {
        ...
};

#endif /* defined(_KERNEL) */

#define EGFS_ARGSVERSION 1
struct egfs_args {
        char *ea_fspec;

        int ea_version;

        ...
};
```

11. Create the `src/sbin/mount_egfs` directory.

12. Create a simple `src/sbin/mount_egfs/Makefile` file:

```
.include <bsd.own.mk>

PROG= mount_egfs
```

```
SRCS= mount_egfs.c
MAN= mount_egfs.8

CPPFLAGS+= -I${NETBSDSRCDIR}/sys
WARNS= 4

.include <bsd.prog.mk>
```

13. Create a simple `src/sbin/mount_egfs/mount_egfs.c` program that calls the mount(2) system call.

    XXX: Add an example or link to the corresponding section.

14. Create an empty `src/sbin/mount_egfs/mount_egfs.8` manual page. Details left out from this guide.
15. Fill in the `egfs_mount` and `egfs_unmount` functions.

    See Section 2.6.

16. Fill in the `egfs_statvfs` function. Return correct data if possible at this point or leave it for a later step.
17. Set the `vop_fsync`, `vop_bwrite` and `vop_putpages` operations to `genfs_nullop`. These need to be defined and return successfully to avoid crashes during sync(2) and mount(2). We will fill them in at a later stage.
18. Set the `vop_abortop` operation to `genfs_abortop`.
19. Set the locking operations to `genfs_lock`, `genfs_unlock` and `genfs_islocked`. You will most likely need locking, so it is better if you get it right from the beginning.

    See Section 2.8.5.

20. Implement the `vop_reclaim` and `vop_inactive` operations to correctly destroy vnodes.

    See Section 2.8.4.

21. Fill in the `egfs_sync` function. In case you do not know what do put in it, just return success (zero); otherwise, serious problems will arise because it will be impossible for the operating system to flush your file system.
22. Fill in the `egfs_root` function. Assuming you already read the file system's root node from disk (or whichever backing store you use) and have it in memory, simply allocate and lock a vnode for it.

    See Section 2.8.3.

    ```
    int
    egfs_root(struct mount *mp, struct vnode **vpp)
    {

            return egfs_alloc_vp(mp, ((struct egfs_mount *)mp)->em_root, vpp);
    }
    ```

23. Improve the mount utility to support standard options (see getmntopts(3)) and possibly some file system specific options too.
24. Implement the `egfs_getattr` and `egfs_setattr` functions operations. As a side effect, implement `egfs_update` and `egfs_sync` too. For the latter, you only need an stub that returns success for now.

See Section 2.11.6.

25. Implement the `egfs_access` operation.

    See Section 2.11.8.

26. Implement the `egfs_print` function. This is trivial, as all it has to do is dump vnode information (its attributes, mostly) on screen, but it will help with debugging.

    See Section 2.11.8.

27. Implement a simple `egfs_lookup` function that can locate any given file; be careful to conform with the locking protocol described in vnodeops(9), as this part is really tricky. At this point, you can forget about the lookup hints (`CREATE`, `DELETE` or `RENAME`); you will add them when needed.

    See Section 2.10.

28. Implement the `egfs_open` function. In the general case, this one only needs to verify that the open mode is correct against the file flags.

```
int
egfs_open(void *v)
{
        struct vnode *vp = ((struct vop_open_args *)v)->a_vp;
        int mode = ((struct vop_open_args *)v)->a_mode;

        struct egfs_node *node;

        node = (struct egfs_node *)vp->v_data;

        if (node->en_flags & APPEND &&
            mode & (FWRITE | O_APPEND)) == FWRITE)
                return EPERM;

        return 0;
}
```

29. Implement the `egfs_close` function. In the general case, this one needs to do nothing aside returning success.

30. Implement the `egfs_readdir` operation so that you can start interacting with your file system. After you add this function, you should be able to list any directory in it, and check that the files' attributes are shown correctly. And most likely, you will start seeing bugs ;-)

    See Section 2.13.3.

31. Implement the `egfs_mkdir` operation. You may need to modify the `egfs_lookup` function to honour the `CREATE` hint.

    See Section 2.10.3.

32. Implement the `egfs_rmdir` operation. You may need to modify the `egfs_lookup` function to honour the `DELETE` hint. Note that adding an operation that removes stuff from the file system is tricky; problems will certainly pop up if you have got bugs in your vnode allocation code or in the `egfs_inactive` or `egfs_reclaim` functions.

    See Section 2.10.3 and Section 2.8.4.

33. Implement the `egfs_create` operation to create regular files (`VREG`) and local sockets (`VSOCK`) .

34. Implement the `egfs_remove` operation to delete files.

35. Implement the `egfs_link` operation to create hard links. Be sure to control the file's hard link count correctly.

36. Implement the `egfs_rename` operation. This one may seem complex due to the amount of arguments it takes, but it is not so difficult to implement. Just keep in mind that it has to manage renames as well as moves and in which situation they happen.

37. Implement the `egfs_read` and `egfs_write` operations. These are quite simple thanks to the indirection provided by the vnode's UVM object.

    See Section 2.11.5.

38. Redirect the `egfs_getpages` and `egfs_putpages` to `genfs_getpages` and `genfs_putpages` respectively. Should be enough for most file systems.

    See Section 2.11.5.2.

39. Implement the `egfs_bmap` and `egfs_strategy` operations.

    See Section 2.11.5.5.

40. Implement the `egfs_truncate` operation.

41. Redirect the `egfs_fcntl`, `egfs_ioctl`, `egfs_poll`, `egfs_revoke` and `egfs_mmap` operations to their corresponding ones in genfs. Should be enough for most-filesystems; note that even FFS does this.

42. Implement the `egfs_pathconf` operation. This one is trivial, although the documentation in pathconf(2) and vnodeops(9) is a bit inconsistent.

```
int
egfs_pathconf(void *v)
{
        int name = ((struct vop_pathconf_args *)v)->a_name;
        register_t *retval = ((struct vop_pathconf_args *)v)->a_retval;

        int error;

        switch (name) {
        case _PC_LINK_MAX:
                *retval = LINK_MAX;
                break;
        case ...:
        ...
        }

        return 0;
}
```

43. Implement the `egfs_symlink` and `egfs_readlink` operations to manage symbolic links.

    See Section 2.12.

44. Implement the `egfs_mknod` operation, which adds support for named pipes and special devices.

    See Section 2.14.

45. Add NFS support. This basically means implementing the `egfs_vptofh`, `egfs_checkexp` and `egfs_fhtovp` VFS operations.

    See Section 2.15.

Notes

1. Technically speaking, a mount point needn't be a directory as you can NFS-mount regular files; the mount point could be a regular file, but this restriction is deliberately imposed because otherwise, the system could run out of name space quickly.

Chapter 3

Processes and threads

This chapter describe processes and threads in NetBSD. This includes process startup, traps and system calls, process and thread creation and termination, signal delivery, and thread scheduling.

CAUTION! This chapter is an ongoing work: it has not been reviewed yet, neither for typos, nor for technical mistakes

3.1 Process startup

3.1.1 `execve` usage

On Unix systems, new programs are started using the `execve` system call. If successful, `execve` replaces the currently-executing program by a new one. This is done within the same process, by reinitializing the whole virtual memory mapping and loading the new program binary in memory. All the process's threads (except for the calling one) are terminated, and the calling thread CPU context is reset for executing the new program startup.

Here is `execve` prototype:

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

`path` is the filesystem path to the new executable. *`argv`* and *`envp`* are two NULL-terminated string arrays that hold the new program arguments and environment variables. `execve` is responsible for copying the arrays to the new process stack.

3.1.2 Overview of in-kernel `execve` code path

Here is the top-down modular diagram for `execve` implementation in the NetBSD kernel when executing a native 32 bit ELF binary on an i386 machine:

- `src/sys/kern/kern_exec.c`: `sys_execve`
  - `src/sys/kern/kern_exec.c`: `execve1`
    - `src/sys/kern/kern_exec.c`: `check_exec`
      - `src/sys/kern/kern_verifiedexec.c`: `veriexec_verify`
      - `src/sys/kern/kern_conf.c`: `*execsw[]->es_makecmds`
        - `src/sys/kern/exec_elf32.c`: `exec_elf_makecmds`

- src/sys/kern/exec_elf32.c: exec_check_header
- src/sys/kern/exec_elf32.c: exec_read_from
- src/sys/kern/exec_conf.c: *execsw[]->u.elf_probe_func
  - src/sys/kern/exec_elf32.c: netbsd_elf_probe

- src/sys/kern/exec_elf32.c: elf_load_psection
- src/sys/kern/exec_elf32.c: elf_load_file
- src/sys/kern/exec_conf.c: *execsw[]->es_setup_stack
  - src/sys/kern/exec_subr.c: exec_setup_stack

- *fetch_element
  - src/sys/kern/kern_exec.c: execve_fetch_element

- *vcp->ev_proc
  - src/sys/kern/exec_subr.c: vmcmd_map_zero
  - src/sys/kern/exec_subr.c: vmcmd_map_pagedvn
  - src/sys/kern/exec_subr.c: vmcmd_map_readvn
  - src/sys/kern/exec_subr.c: vmcmd_readvn

- src/sys/kern/exec_conf.c: *execsw[]->es_copyargs
  - src/sys/kern/kern_exec.c: copyargs

- src/sys/kern/kern_clock.c: stopprofclock
- src/sys/kern/kern_descrip.c: fdcloseexec
- src/sys/kern/kern_sig.c: execsigs
- src/sys/kern/kern_ras.c: ras_purgeall
- src/sys/kern/exec_subr.c: doexechooks
- src/sys/sys/event.h: KNOTE
  - src/sys/kern/kern_event.c: knote

- src/sys/kern/exec_conf.c: *execsw[]->es_setregs
  - src/sys/arch/i386/i386/machdep.c: setregs

- src/sys/kern/kern_exec.c: exec_sigcode_map
- src/sys/kern/kern_exec.c: *p->p_emul->e_proc_exit (NULL)
- src/sys/kern/kern_exec.c: *p->p_emul->e_proc_exec (NULL)

execve calls execve1 with a pointer to a function called fetch_element, responsible for loading program arguments and environment variables in kernel space. The primary reason for this abstraction function is to allow fetching pointers from a 32 bit process on a 64 bit system.

execve1 uses a variable of type struct exec_package (defined in src/sys/sys/exec.h) to share information with the called functions.

The makecmds is responsible for checking if the program can be loaded, and to build a set of virtual memory commands (vmcmd's) that can be used later to setup the virtual memory space and to load the program code and data sections. The set of vmcmd's is stored in the ep_vmcmds field of the exec package. The use of these vmcmd set allows cancellation of the execution process before a commitment point.

3.1.3 Multiple executable format support with the exec switch

The exec switch is an array of structure struct execsw defined in src/sys/kern/exec_conf.c: execsw[]. The struct execsw itself is defined in src/sys/sys/exec.h.

Each entry in the exec switch is written for a given executable format and a given kernel ABI. It contains test methods to check if a binary fits the format and ABI, and the methods to load it and start it up if it does. One can find here various methods called within execve code path.

Table 3-1. struct execsw fields summary

Field name	Description
es_hdrsz	The size of the executable format header
es_makecmds	A method that checks if the program can be executed, and if it does, create the vmcmds required to setup the virtual memory space (this includes loading the executable code and data sections).
u.elf_probe_func u.ecoff_probe_func u.macho_probe_func	Executable probe method, used by the es_makecmds method to check if the binary can be executed. The u field is an union that contains probe methods for ELF, ECOFF and Mach-O formats
es_emul	The struct emul used for handling different kernel ABI. It is covered in detail in Section 3.2.2.

Field name	Description
es_prio	A priority level for this exec switch entry. This field helps choosing the test order for exec switch entries
es_arglen	XXX ?
es_copyargs	Method used to copy the new program arguments and environment function in user space
es_setregs	Machine-dependent method used to set up the initial process CPU registers
es_coredump	Method used to produce a core from the process
es_setup_stack	Method called by es_makecmds to produce a set of vmcmd for setting up the new process stack.

execve1 iterate on the exec switch entries, using the es_priority for ordering, and calls the es_makecmds method of each entry until it gets a match.

The es_makecmds will fill the exec package's ep_vmcmds field with vmcmds that will be used later for setting up the new process virtual memory space. See Section 3.1.3.2 for details about the vmcmds.

3.1.3.1 Executable format probe

The executable format probe is called by the es_makecmds method. Its job is simply to check if the executable binary can be handled by this exec switch entry. It can check a signature in the binary (e.g.: ELF note section), the name of a dynamic linker embedded in the binary, and so on.

Some probe functions feature wildcard, and will be used as last resort, with the help of the es_prio field. This is the case of the native ELF 32 bit entry, for instance.

3.1.3.2 Virtual memory space setup commands (vmcmds)

Vmcmds are stored in an array of struct exec_vmcmd (defined in src/sys/sys/exec.h) in the ep_vmcmds field of the exec package, before execve1 decides to execute or destroy them.

struct exec_vmcmd defines, in the ev_proc field, a pointer to the method that will perform the command, The other fields are used to store the method's arguments.

Four methods are available in src/sys/kern/exec_subr.c

Table 3-2. vmcmd methods

Name	Description
vmcmd_map_pagedvn	Map memory from a vnode. Appropriate for handling demand-paged text and data segments.
vmcmd_map_readvn	Read memory from a vnode. Appropriate for handling non-demand-paged text/data segments, i.e. impure objects (a la OMAGIC and NMAGIC).
vmcmd_readvn	XXX ?
vmcmd_zero	Maps a region of zero-filled memory

Vmcmd are created using `new_vmcmd`, and can be destroyed using `kill_vmcmd`.

3.1.3.3 Stack virtual memory space setup

The `es_setup_stack` field of the exec switch holds a pointer to the method in charge of generating the vmcmd for setting up the stack space. Filling the stack with arguments and environment is done later, by the `es_copyargs` method.

For native ELF binaries, the `netbsd32_elf32_copyargs` (obtained by a macro from `elf_copyargs` method in `src/sys/kern/exec_elf32.c`) is used. It calls the `copyargs` (from `src/sys/kern/kern_exec.c`) for the part of the job which is not specific to ELF.

`copyargs` has to copy back the arguments and environment string from the kernel copy (in the exec package) to the new process stack in userland. Then the arrays of pointers to the strings are reconstructed, and finally, the pointers to the array, and the argument count, are copied to the top of the stack. The new program stack pointer will be set to point to the argument count, followed by the argument array pointer, as expected by any ANSI program.

Dynamic ELF executable are special: they need a structure called the ELF auxiliary table to be copied on the stack. The table is an array of pairs of key and values for various things such as the ELF header address in user memory, the page size, or the entry point of the ELF executable

Note that when starting a dynamic ELF executable, the ELF loader (also known as the interpreter: `/usr/libexec/ld.elf_so`) is loaded with the executable by the kernel. The ELF loader is started by the kernel and is responsible for starting the executable itself afterwards.

3.1.3.4 Initial register setup

`es_setregs` is a machine dependent method responsible for setting up the initial process CPU registers. On any machine, the method will have to set the registers holding the instruction pointer, the stack pointer and the machine state. Some ports will need more work (for instance i386 will set up the segment registers, and Local Descriptor Table)

The CPU registers are stored in a struct trapframe, available from struct lwp.

3.1.3.5 Return to userland

After `execve` has finished his work, the new process is ready for running. It is available in the run queue and it will be picked up by the scheduler when appropriate.

From the scheduler point of view, starting or resuming a process execution is the same operation: returning to userland. This involves switching to the process virtual memory space, and loading the process CPU registers. By loading the machine state register with the system bit off, kernel privileges are dropped.

XXX details

3.2 Traps and system calls

When the processor encounter an exception (memory fault, division by zero, system call instruction...), it executes a trap: control is transferred to the kernel, and after some assembly routine in `locore.S`, the CPU drops in the `syscall_plain` (from `src/sys/arch/i386/i386/syscall.c` on i386) for system calls, or in the `trap` function (from `src/sys/arch/i386/i386/trap.c` on i386) for other traps.

There is also a `syscall_fancy` system call handler which is only used when the process is being traced by **ktrace**.

3.2.1 Traps

XXX write me

3.2.2 Multiple kernel ABI support with the emul switch

The struct emul is defined in `src/sys/sys/proc.h`. It defines various methods and parameters to handle system calls and traps. Each kernel ABI supported by the NetBSD kernel has its own struct emul. For instance, Linux ABI defines `emul_linux` in `src/sys/compat/linux/common/linux_exec.c`, and the native ABI defines `emul_netbsd`, in `src/sys/kern/kern_exec.c`.

The struct emul for the current ABI is obtained from the `es_emul` field of the exec switch entry that was selected by `execve`. The kernel holds a pointer to it in the process' struct proc (defined in `src/sys/sys/proc.h`).

Most importantly, the struct emul defines the system call handler function, and the system call table.

3.2.3 The syscalls.master table

Each kernel ABI have a system call table. The table maps system call numbers to functions implementing the system call in the kernel (e.g.: system call number 2 is `fork`). The convention (for native syscalls) is that the kernel function implementing syscall `foo` is called `sys_foo`. Emulation syscalls have their own conventions, like `linux_sys_` prefix for the Linux emulation. The native system call table can be found in `src/sys/kern/syscalls.master`.

This file is not written in C language. After any change, it must be processed by the `Makefile` available in the same directory. `syscalls.master` processing is controlled by the configuration found in `syscalls.conf`, and it will output several files:

Table 3-3. Files produced from `syscalls.master`

File name	Description
`syscallargs.h`	Define the system call arguments structures, used to pass data from the system call handler function to the function implementing the system call.
`syscalls.c`	An array of strings containing the names for the system calls

File name	Description
syscall.h	Preprocessor defines for each system call name and number — used in libc
sysent.c	An array containing for each system call an entry with the number of arguments, the size of the system call arguments structure, and a pointer to the function that implements the system call in the kernel

In order to avoid namespace collision, non native ABI have `syscalls.conf` defining output file names prefixed by tags (e.g: `linux_` for Linux ABI).

system call argument structures (syscallarg for short) are always used to pass arguments to functions implementing the system calls. Each system call has its own syscallarg structure. This encapsulation layer is here to hide endianness differences.

All functions implementing system calls have the same prototype:

```
int syscall(struct lwp *l, void * v, register_t *retval);
```

l is the struct lwp for the calling thread, *v* is the syscallarg structure pointer, and *retval* is a pointer to the return value. The function returns the error code (see errno(2)) or 0 if there was no error. Note that the prototype is not the same as the "declaration" in `syscalls.master`. The declaration in `syscalls.master` corresponds to the documented prototype for the system call. This is because system calls as seen from userland programs have different prototypes, but the `sys_...` kernel functions implementing them must have the same prototype to unify the interface between MD syscall handlers and MI syscall implementation. In `syscalls.master`, the declaration shows the syscall arguments as seen by userland and determines the members of the syscallarg structure, which encapsulates the syscall arguments and has one member for each one.

While generating the files listed above some substitutions on the function name are performed: the syscalls tagged as `COMPAT_XX` are prefixed by `compat_xx_`, same for the syscallarg structure name. So the actual kernel function implementing those syscalls have to be defined in a corresponding way. Example: if `syscalls.master` has a line

```
97 COMPAT_30 { int sys_socket(int domain, int type, int protocol); }
```

the actual syscall function will have this prototype:

```
int compat_30_sys_socket(struct lwp *l, void * v, register_t *retval);
```

and *v* is a pointer to struct compat_30_sys_socket_args, whose declaration is the following:

```
struct compat_30_sys_socket_args {
        syscallarg(int) domain;
        syscallarg(int) type;
        syscallarg(int) protocol;
};
```

Note the correspondence with the documented prototype of the socket(2) syscall and the declaration of `sys_socket` in `syscalls.master`. The types of syscall arguments are wrapped by `syscallarg` macro, which ensures that the structure members will be padded to a minimum size, again for unified interface between MD and MI code. That's why those members should not be accessed directly, but by the `SCARG` macro, which takes a pointer to the syscall arg structure and the argument name and extracts the argument's value. See below (#syscall_howto) for an example.

3.2.4 System call implementation in libc

The system call implementation in libc is autogenerated from the kernel implementation. As an example, let's examine the implementation of the access(2) function in libc. It can be found in the `access.S` file, which does not exist in the sources — it is autogenerated when libc is built. It uses macros defined in `src/sys/sys/syscall.h` and `src/lib/libc/arch/`*`MACHINE_ARCH`*`/SYS.h`: the `syscall.h` file contains defines which map the syscall names to syscall numbers. The syscall function names are changed by replacing the `sys_` prefix by `SYS_`. The `syscall.h` header file is also autogenerated from `src/sys/kern/syscalls.master` by running **make init_sysent.c** in `src/sys/kern`, as described above. By including `SYS.h`, we get `syscall.h` and the `RSYSCALL` macro, which accepts the syscall name, automatically adds the `SYS_` prefix, takes the corresponding number, and defines a function of the name given whose body is just the execution of the syscall itself with the right number. (The method of execution and of transfer of the syscall number and its arguments are machine dependent, but this is hidden in the `RSYSCALL` macro.)

To continue the example of access(2), `syscall.h` contains

```
#define SYS_access      33
```

so

```
RSYSCALL(access)
```

will result in defining the function `access`, which will execute the syscall with number 33. Thus, `access.S` needs to contain just:

```
#include "SYS.h"
RSYSCALL(access)
```

To automate this further, it is enough to add the name of this file to the `ASM` variable in `src/lib/libc/sys/Makefile.inc` and the file will be autogenerated with this content when libc is built.

The above is true for libc functions which correspond exactly to the kernel syscalls. It is not always the case, even if the functions are found in section 2 of the manuals. For example the wait(2), wait3(2) and waitpid(2) functions are implemented as wrappers of only one syscall, wait4(2). In such case the procedure above yields the `wait4` function and the wrappers can reference it as if it were a normal C function.

3.2.5 How to add a new system call

Let's pretend that the access(2) syscall does not exist yet and you want to add it to the kernel. How to proceed?

- add the syscall to the `src/sys/kern/syscalls.master` list:

  ```
  33      STD             { int sys_access(const char *path, int flags); }
  ```

- Run **make init_sysent.c** under `src/sys/kern`. This will update the autogenerated files: `syscallargs.h`, `syscall.h`, `init_sysent.c` and `syscalls.c`.
- Implement the kernel part of the system call, which will have the prototype:

```
int sys_access(struct lwp *l, void * v, register_t *retval);
```

  as all other syscalls. To get the syscall arguments cast `v` to a pointer to struct sys_access_args and use the `SCARG` macro to retrieve them from that structure. For example, to get the *`flags`* argument if `uap` is a pointer to struct sys_access_args obtained by casting `v`, use:

  ```
  SCARG(uap, flags)
  ```

  The type struct sys_access_args and the function `sys_access` are declared in `sys/syscallargs.h`, which is autogenerated from `src/sys/kern/syscalls.master`. Use

  ```
  #include <sys/syscallargs.h>
  ```

  to get those declarations.

  Look in `src/sys/kern/vfs_syscalls.c` for the real implementation of `sys_access`.

- Run **make includes** in `src/sys/sys`. This will copy the autogenerated include files (most importantly, `syscall.h`) to `usr/include` under `DESTDIR`, where libc build will find them in the next steps.
- Add `access.S` to the `ASM` variable in `src/lib/libc/sys/Makefile.inc`.

This is all. To test the new syscall, simply rebuild libc (`access.S` will be generated at this point) and reboot with a new kernel containing the new syscall. To make the new syscall generally useful, its prototype should be added to an appropriate header file for use by userspace programs — in the case of access(2), this is unistd.h, which is found in the NetBSD sources at `src/include/unistd.h`.

3.2.6 Versioning a system call

If the system call ABI (or even API) changes, it is necessary to implement the old syscall with the original semantics to be used by old binaries. The new version of the syscall has a different syscall number, while the original one retains the old number. This is called versioning.

The naming conventions associated with versioning are complex. If the original system call is called `foo` (and implemented by a `sys_foo` function) and it is changed after the *x.y* release, the new syscall will be named `__fooxy`, with the function implementing it being named `sys___fooxy`. The original syscall (left for compatibility) will be still declared as sys_foo in `syscalls.master`, but will be tagged as `COMPAT_XY`, so the function will be named `compat_xy_sys_foo`. We will call `sys_foo` the original version, `sys___fooxy` the new version and `compat_xy_sys_foo` the compatibility version in the procedure described below.

Now if the syscall is versioned again after version *z.q* has been released, the newest version will be called `__foozq`. The intermediate version (formerly the new version) will have to be retained for compatibility, so it will be tagged as `COMPAT_ZQ`, which will change the function name from `sys___fooxy` to `compat_zq_sys___fooxy`. The oldest version `compat_xy_sys_foo` will be unaffected by the second versioning.

HOW TO change a system call ABI or API and add a compatibility version? Let's look at a real example: versioning of the socket(2) system call after the error code in case of unsupported address family changed from EPROTONOSUPPORT to EAFNOSUPPORT between NetBSD 3.0 and 4.0.

- tag the old version (`sys_socket`) with the right `COMPAT_XY` in `syscalls.master`. In the case of `sys_socket`, it is `COMPAT_30`, because NetBSD 3.0 was the last version before the system call changed.
- add the new version at the end of `syscalls.master` (this effectively allocates a new syscall number). Name the new version as described above. In our case, it will be `sys___socket30`:

  ```
  394 STD  { int sys___socket30(int domain, int type, int protocol); }
  ```
- The function implementing the socket syscall now needs to be renamed from `sys_socket` to `sys___socket30` to match the change above. Ideally, at this moment the change which requires versioning would be made. (Though in practice it happens that a change is made and only later it is realized that it breaks compatibility and versioning is needed.)
- Implement the compatibility version, name it compat_xy_sys_... as described above. The implementation belongs under `src/sys/compat` and it shouldn't be a modified copy of the new version, because the copies would eventually diverge. Rather, it should be implemented in terms of the new version, adding the adjustments needed for compatibility (which means that it should behave exactly as the old version did).

  In our example, the compatibility version would be named `compat_30_sys_socket`. It can be found in `src/sys/compat/common/uipc_syscalls_30.c`.
- Find all references to the old syscall function in the kernel and point them to the compatibility version or to the new version as appropriate. (The kernel would not link otherwise.) For example, many of the compatibility syscalls or the `syscalls.master` tables for various emulations under `src/sys/compat` used to refer to `sys_socket`. Decision if the references should be changed to the compatibility version or to the new version depend on the behavior of the OS that we intend to emulate. E.g., FreeBSD uses the old error number, while System V uses the new one.

Now the kernel should be compilable and old statically linked binaries should work, as should binaries using the old libc. Nothing uses the new syscall yet. We have to make a new libc, which will contain both the new and the compatibility syscall:

- in `src/lib/libc/sys/Makefile.inc`, replace the name of the old syscall by the new syscall (`__socket30` in our example). When libc is rebuilt, it will contain the new function, but no programs use this internal name with underscore, so it is not useful yet. Also, we have lost the old name.
- To make newly compiled programs use the new syscall when they refer to the usual name (`socket` in our example), we add a `__RENAME(newname)` statement after the declaration of the usual name is declared. In the case of `socket`, this is `src/sys/sys/socket.h`:

  ```
  int     socket(int, int, int)
  #if !defined(__LIBC12_SOURCE__) && !defined(_STANDALONE)
  ```

```
__RENAME(__socket30)
#endif
```

Now, when a program is recompiled using this header, references to `socket` will be replaced by `__socket30`, except for compilation of standalone tools (basically bootloaders), which define `_STANDALONE`, and libc compat code itself, which defines `__LIBC12_SOURCE__`. The `__RENAME` causes the compiler to emit references to the `__socket30` symbol when `socket` is used in the source. The symbol will be then resolved by the linker to the new function (implemented by the new system call). Old binaries are unaware of this and continue to reference `socket`, which should be resolved to the old function (having the same API as before the change). We will re-add the old function in the next step.

- To make the old binaries work with the new libc, we must add the old function. We add it under `src/lib/libc/compat/sys`, implementing it using the new function. Note that we did not use the compatibility syscall in the kernel at all, so old programs will work with the new libc, even if the kernel is built without `COMPAT_30`. The compatibility syscall is there only for the old libc, which is used if the shared library was not upgraded, or internally by statically linked programs.

We are done — we have covered the cases of old binaries, old libc and new kernel (including statically linked binaries), old binaries, new libc and new kernel, and new binaries, new libc and new kernel.

3.2.7 Committing changes to syscall tables

When committing your work (either a new syscall or a new syscall version with the compatibility syscalls), you should remember to commit the source (`syscalls.master`) for the autogenerated files first, and then regenerate and commit the autogenerated files. They contain the RCS Id of the source file and this way, the RCS Id will refer to the current source version. The assembly files generated by `src/lib/libc/sys/Makefile.inc` are not kept in the repository at all, they are regenerated every time libc is built.

3.2.8 Managing 32 bit system calls on 64 bit systems

When executing 32 bit binaries on a 64 bit system, care must be taken to only use addresses below 4 GB. This is a problem at process creation, when the stack and heap are allocated, but also for each system call, where 32 bits pointers handled by the 32 bit process are manipulated by the 64 bit kernel.

For a kernel built as a 64 bit binary, a 32 bit pointer is not something that makes sense: pointers can only be 64 bit long. This is why 32 bit pointers are defined as an u_int32_t synonym called netbsd32_pointer_t (in `src/sys/compat/netbsd32/netbsd32.h`).

For `copyin` and `copyout`, true 64 bits pointers are required. They are obtained by casting the netbsd32_pointer_t through the `NETBSD32PTR64` macro.

Most of the time, implementation of a 32 bit system call is just about casting pointers and to call the 64 version of the system call. An example of such a situation can be found in `src/sys/compat/netbsd32/netbsd32_time.c`: `netbsd32_timer_delete`. Provided that the 32 bit system call argument structure pointer is called `uap`, and the 64 bit one is called `ua`, then helper macros called `NETBSD32TO64_UAP`, `NETBSD32TOP_UAP`, `NETBSD32TOX_UAP`, and `NETBSD32TOX64_UAP` can be used. Sources in `src/sys/compat/netbsd32` provide multiple examples.

3.3 Processes and threads creation

3.3.1 `fork`, `clone`, and `pthread_create` usage

XXX write me

3.3.2 Overview of `fork` code path

XXX write me

3.3.3 Overview of `pthread_create` code path

XXX write me

3.4 Processes and threads termination

3.4.1 `exit`, and `pthread_exit` usage

XXX write me

3.4.2 Overview of `exit` code path

XXX write me

3.4.3 Overview of `pthread_exit` code path

XXX write me

3.5 Signal delivery

3.5.1 Deciding what to do with a signal

XXX write me

3.5.2 The `sendsig` function

For each kernel ABI, struct emul defines a machine-dependent `sendsig` function, which is responsible for altering the process user context so that it calls a signal handler.

`sendsig` builds a stack frame containing the CPU registers before the signal handler invocation. The CPU registers are altered so that on return to userland, the process executes the signal handler and have the stack pointer set to the new stack frame.

If requested at `sigaction` call time, `sendsig` will also add a struct siginfo to the stack frame.

Finally, `sendsig` may copy a small piece of assembly code (called a "signal trampoline") to perform cleanup after handling the signal. This is detailed in the next section. Note that modern NetBSD native programs do not use a trampoline anymore: it is only used for older programs, and emulation of other operating systems.

3.5.3 Cleaning up state after signal handler execution

Once the signal handler returns, the kernel must destroy the signal handler context and restore the previous process state. This can be achieved by two ways.

First method, using the kernel-provided signal trampoline: `sendsig` have copied the signal trampoline on the stack and has prepared the stack and/or CPU registers so that the signal handler returns to the signal trampoline. The job of the signal trampoline is to call the `sigreturn` or the `setcontext` system calls, handling a pointer to the CPU registers saved on stack. This restores the CPU registers to their values before the signal handler invocation, and next time the process will return to userland, it will resume its execution where it stopped.

The native signal trampoline for i386 is called `sigcode` and can be found in `src/sys/arch/i386/i386/locore.S`. Each emulated ABI has its own signal trampoline, which can be quite close to the native one, except usually for the `sigreturn` system call number.

The second method is to use a signal trampoline provided by libc. This is how modern NetBSD native programs do. At the time the `sigaction` system call is invoked, the libc stub handle a pointer to a signal trampoline in libc, which is in charge of calling `setcontext`.

`sendsig` will use that pointer as the return address for the signal handler. This method is better than the previous one, because it removes the need for an executable stack page where the signal trampoline is stored. The trampoline is now stored in the code segment of libc. For instance, for i386, the signal trampoline is named `__sigtramp_siginfo_2` and can be found in `src/lib/libc/arch/i386/sys/__sigtramp2.S`.

3.6 Thread scheduling

3.6.1 Overview

NetBSD 5.0 introduced a new scheduling API that allows for different scheduling algorithms to be implemented and selected at compile-time. There are currently two different scheduling algorithms available: the traditional 4.4BSD-based scheduler and the more modern M2 scheduler.

NetBSD supports the three scheduling policies required by POSIX in order to support the POSIX real-time scheduling extensions:

- SCHED_OTHER: Time sharing (TS), the default on NetBSD

- SCHED_FIFO: First in, first out
- SCHED_RR: Round-robin

SCHED_FIFO and SCHED_RR are predefined scheduling policies, leaving SCHED_OTHER as an implementation-specific policy.

Currently, there are 224 different priority levels with 64 being available for the user level. Scheduling priorities are organized within the following classes:

Table 3-4. Scheduling priorities

Class	Range	# Levels	Description
Kernel (RT)	192..223	32	Software interrupts.
User (RT)	128..191	64	Real-time user threads (SCHED_FIFO and SCHED_RR policies).
Kernel threads	96..127	32	Internal kernel threads (*kthreads*), used by I/O, VM and other kernel subsystems.
Kernel	64..95	32	Kernel priority for user processes/threads, temporarily assigned when entering kernel-space and blocking.
User (TS)	0..63	64	Time-sharing range, user processes and threads (SCHED_RR policy)

Threads running with the SCHED_FIFO policy have a fixed priority, i.e. the kernel does not change their priority dynamically. A SCHED_FIFO thread runs until

- completion
- voluntary yielding the CPU
- blocking on an I/O operation or other resources (memory allocation, locks)
- preemption by a higher priority real-time thread

SCHED_RR works similar to SCHED_FIFO, except that such threads have a default time-slice of 100ms.

For the SCHED_OTHER policy, both schedulers currently use the same run queue implementation, employing multi-level feedback queues. By dynamically adjusting a thread's priority to reflect its CPU and resource utilization, this approach allows the system to be responsive even under heavy loads.

Each runnable thread is placed on one of the runqueues, according to its priority. Each thread is allowed to run on the CPU for a certain amount of time, its time-slice or quantum. Once the thread has used up its time-slice, it is placed on the back on its runqueue. When the scheduler searches for a new thread to run on the CPU, the first thread of the highest priority, non-empty runqueue is selected.

3.6.1.1 The 4.4BSD Scheduler

The 4.4BSD scheduler adjusts a thread's priority dynamically as it accumulates CPU-time. CPU utilization is incremented in `hardclock` each time the system clock ticks and the thread is found to be executing. An estimate of a thread's recent CPU utilization is stored in `l_estcpu`, which is adjusted once per second in `schedcpu` via a digital decay filter. Whenever a thread accumulates four ticks in its CPU utilization, `schedclock` invokes `resetpriority` to recalculate the process's scheduling priority.

3.6.1.2 The M2 scheduler

The M2 scheduler employs a traditional time-sharing approach similar to Unix System V Release 4 and Solaris.

3.6.2 References

The common scheduler API is implemented within the file `src/sys/kern/kern_synch.c`. Additional information can be found in csf(9). Generic run-queues are implemented in `src/sys/kern/kern_runq.c`. Detailed information about the 4.4BSD scheduler is given in [McKusick]. A description of the SVR4 scheduler is provided in [Goodheart].

Chapter 4
Networking

This chapter describes code modules related to core networking. It begins with a discussion on core networking components: the routing code, sockets and mbufs. After this attention is given to the TCP/IP suite. Information on services provided by the networking stack, including various pseudo devices, can be found from the chapter on networking services .

This chapter describes only the involved interfaces and is meant to give an overview for people wishing to work with the code. A line-by-line discussion on most parts of the networking subsystem can be found from TCP/Illustrated, Vol2. It is still mostly accurate these days, and this chapter does not attempt to rewrite what was already written.

4.1 Routing

The routing code controls the flow of all packets within the networking subsystem. It makes the decision of where a packet with a given destination address should be sent next. The current routing table can be dumped using the command **netstat -r**; going over the normal table contents may be beneficial for the following discussion.

For example, in TCP/IP networking the routing module forwards the packets to the correct addresses and is involved in choosing the network interface through which the packets should go. The decision whether to send to an intermediate gateway or directly to the target is made here. And, finally, link layer lookups are done, which in the case of Ethernet and IPv4 would be ARP.

One important concept in the routing subsystem is cloning routes. These routes are "cloned" to create a more specific route each time they are resolved; copy-on-read if it makes any sense. A natural example is an interface route to the local Ethernet. Any packet sent through this link will get a cloned route entry for ARP resolution. This enables the caching of ARP responses without adding any explicit support for it on the routing layer.

The routing code can be thought to consist of three separate entities. An overview will be given now and a more dedicated discussion can be found later.

- The routing database backend is implemented in `net/radix.c`. As hinted by the name, the internal structure is a radix tree.
- The route query and control interface is located in `net/route.c`. This module is accessed from within the kernel and parameters are passed mainly using the struct route and the struct rtentry structures.
- The routing socket interface is located in `net/rtsock.c`. It is used to control the routing table entries from userspace and is accessed by opening a socket in the protocol family PF_ROUTE.

It is good to keep in mind that routing control messages can come either from a process in the operating system (user running **/sbin/route**, **gated**, etc.) or from the network (e.g. ICMP). This explains why internally some portions are controlled and parameters passed using a fashion similar to what the routing socket uses.

4.1.1 Network Routing DB Backend

This section describes the interface to the radix tree routing database. The actual internal operation of the radix code is beyond the scope of this document and can be found, for example, in TCP/IP Illustrated Vol2. Much of the complexity of this module is due to its aspiration to be as generic as possible and adaptable to any networking domain.

A radix tree is accessed mostly through the member functions in struct radix_node_head (but there are some exceptions). These function pointers are initialized after a radix tree is created with `rn_inithead` called from each networking domain's initialization routine. The head is then stored in the global array `rt_tables` using the domain type as the indexing value.

As arguments the radix tree functions take void *'s, which point to a struct sockaddr type of data structure. But since the radix tree layer treats the arguments as opaque data and a struct sockaddr contains header data (such as the sockaddr family *`sa_family`* or, specific to IP, the port *`sin_port`*) before the actual network address, each networking domain specifies a bit offset where the network address is located. This offset is given in struct domain by `dom_rtoffset` and used when testing against the supplied void *'s. Additionally, the radix tree expects to find the length of the structure underlying the void * at the first byte of the provided argument. This also matches the (BSD) struct sockaddr layout.

Most of the interface methods are located in the struct radix_node_head accessible through `rt_tables`. Theoretically the jump through the function pointer hoop is unnecessary, since the pointers are initialized to the same values for all trees in `net/radix.c`. Not all of the members of the structure are ever used and only the ones used are described here. Included in the description are also the ones not provided through function pointers, but accessed directly. They can be differentiated by looking at the function prefix: ones inside radix_node_head start with rnh, while directly accessed ones start with rn.

Table 4-1. struct radix_node_head interfaces

name	description
rnh_addaddr	Adds an entry with the given address and netmask to the database. The storage space for the database structures is provided by the caller as an argument (usually these are part of struct rtentry, as we will see later). The route target is already prefilled into the storage space by the caller.
rnh_deladdr	Removes an address with a matching netmask from the given radix database.
rnh_matchaddr	Locate the entry in the database which provides the best match for the given address. "Best match" is defined as the match having the longest prefix of 1-bits in the netmask.
rnh_lookup	Locate the most exact entry in the database for the given address with the netmask of the entry matching the argument given.

name	description
rnh_walktree	Walks the database calling the provided function for each entry. This is useful e.g. for dumping the entire routing table or flushing routes cloned from a certain parent entry.
rn_search	Returns the leaf node found at the end of the bit comparisons. This is either a match or the leaf in the the tree that should be backtracked to find a match.
rn_refines	Compares two netmasks and checks which one has more bits set, i.e. which one is "more exact".
rn_addmask	Enters the supplied netmask into the netmask tree.

4.1.2 Routing Interface

The route interface implemented in `net/route.c` is used by the kernel proper when it requires routing services to discover where a packet should be sent to.

The argument passing to this modules revolves around, in addition to struct sockaddr and the radix structures mentioned in the previous chapter, two structures: struct route and struct rtentry.

The structure struct route contains a struct rtentry in addition to a struct sockaddr signalling the destination address for the route. The destination is decoupled from the routing information so that multiple destinations could share the same route structure; think outgoing connections routed to the gateway, for example.

The routing information itself is contained in struct rtentry defined in `net/route.h`. It consists of the following fields:

Table 4-2. struct rtentry members

type	name	description
struct radix_node [2]	*rt_nodes*	radix tree glue
struct sockaddr *	*rt_gateway*	gateway address
int	*rt_flags*	flags
u_long	*rt_use*	number of times the route was used
struct ifnet *	*rt_ifp*	pointer to the interface structure of the route target
struct ifaddr *	*rt_ifa*	Pointer to the interface address of the route target. The sockaddr behind this pointer can be for example of type struct sockaddr_in or struct sockaddr_dl.
const struct sockaddr *	*rt_genmask*	Cloning mask for struct sockaddr.
caddr_t	*rt_llinfo*	Link level structure pointer. For example, this will point to struct llinfo_arp for Ethernet.

type	name	description
struct rt_metrics	*rt_rmx*	The route metrics used by routing protocols. This includes information such as the path MTU and estimated RTT and RTT variance.
struct rtentry *	*rt_gwroute*	In the case of a gateway route, this contains (after it is resolved) the route to the gateway. For example, on IP/Ethernet, the route containing the gateway IP address will have a pointer to the gateway MAC address route here.
LIST_HEAD(, rttimer)	rt_timer	Misc. timers associated with a route. This queue is accessed through the `rt_timer`* family of functions declared in `net/route.h`.
struct rtentry *	rt_parent	The parent of a cloned route. This is used for cleanup when the parent route is removed.

Routes can be queried through two different interfaces:

```
void rtalloc(struct route *ro);
struct rtentry *rtalloc1(const struct sockaddr *dst, int report);
```

The difference is that the former is a convenience function which skips the routing lookup in case the rtentry contained within struct route already contains a route that is up. As an example, struct route is included in struct inpcb to act as a route cache. It helps especially for TCP connections, where the endpoint does not change and therefore the route remains the same.

The latter is used to lookup information from the routing database. The `report` parameter is overloaded to control two operations: whether to create a routing socket message in case there is no route available and whether to clone a route in case the resolved route is a cloning route (quite clearly both of these conditions cannot be true during the same lookup, so the only possibility for ambiguity is for the programmers).

After routing information (struct rtentry) is no longer needed, the routing entry is to be released using `rtfree`. This takes care of appropriate reference counting and releasing the underlying data structures. Notice that this does not delete a routing table entry, it merely releases a route entry created from a routing table entry.

4.1.3 Routing Sockets

Routing sockets are used for controlling the routing table. The routing socket kernel portion is implemented in the file `net/rtsock.c`.

While in BSD systems the routing code is within the kernel, the decisions on the routing table entries are controlled from outside, usually from a userspace routing daemon. The routing socket (simply a socket of type PF_ROUTE) enables the communication between the user and kernel portions. For simpler systems, such as normal desktop machines with essentially the default gateway address being the only routing information, the routing socket is used to set the gateway address. Smart routers will use this interface for programs such as **routed**.

The routing socket acts like any other socket: it is possible to write to it or read from it. Writing (from the userland perspective) is handled by `route_output`, while data is transferred to userspace using the `raw_input` call giving raw data and the routing socket addressing identifiers as the parameters.

The data passed through the routing socket is described by a structure called struct rt_msghdr, which is declared in `net/route.h`. The message type field in the header identifies the type of activity taking place, e.g. RTM_ADD means adding a new route and RTM_REDIRECT means redirecting an existing route. In the latter case, the input comes from the network e.g. in the form of an ICMP packet.

The addresses involved with the routing messages are handled in a somewhat non-obvious way within the file `net/rtsock.c`. They are passed as binary data in the message, read into a structure called struct rt_addrinfo, and used like local variables throughout the file because of preprocessor magic such as the following:

```
#define dst     info.rti_info[RTAX_DST]
```

To be compatible with the routing socket and its propagation of information, networking subsystems should support the `rtrequest` interface. It is called through the *`if_rtrequest`* member in struct ifaddr. Examples include `arp_rtrequest` for Ethernet interfaces and `llc_rtrequest` in the OSI ISO stack. Rtrequest methods handle the same requests as what are communicated via the routing socket (RTM_ADD, RTM_DELETE, ...), but the actual routing socket message layout is handled within the routing socket code.

4.2 Sockets

The Berkeley abstraction for a communication endpoint is called a socket. From the userspace perspective, the handle to a socket is an integer, no different from any other descriptor. There are differences, such as support for interface calls reserved only for sockets (`getsockopt`, `setsockopt`, etc.), but on the basic level, the interface is the same.

However, under the hood things diverge quickly. The integer descriptor value is used to lookup the kernel internal file descriptor structure, struct file, as usual. After this the socket data structure, struct socket is found from the pointer *`f_data`* in struct file.

When discussing sockets it is important to remember that they were designed as an abstraction to the underlying protocol layers.

4.2.1 Socket Data Structure

Inside the kernel, a socket's information is contained within struct socket. This structure is not defined in `sys/socket.h`, which mostly deals with the user-kernel interface, but rather in `sys/socketvar.h`.

Table 4-3. struct socket members

type	name	description
short	*so_type*	The generic socket type. Well-known examples are SOCK_STREAM and SOCK_DGRAM

type	name	description
short	*so_options*	socket options. Most of these, such as SO_REUSEADDR, can be set using `setsockopt`. Others, such as SO_ACCEPTCONN as set using other methods (in this case calling `listen`)
short	*so_linger*	Time the socket lingers on after being closed. Used if SO_LINGER is set. An example user is TCP.
short	*so_state*	internal socket state flags controlled by the kernel. Some, however, are indirectly settable by userspace, for example SS_ASYNC to deliver async I/O notifications.
const struct protosw *	*so_proto*	socket protocol handle. This is used to attach the socket to a certain protocol, for example IP/UDP. The socket protocol requests, such as PRU_USRREQ used for sending packets, are accessed through this member. See also section on socket protocol support.
short	*so_timeo*	connection timeout, not used except as a wakeup() address(?)
u_short	*so_error*	an error value. This field is used to store error values that should be returned to socket routine callers once they are executed/scheduled.
pid_t	*so_pgid*	the pgid use to identify the target for socket-related signal delivery.
u_long	*so_oobmark*	counter to the oob mark
struct sockbuf	*so_snd, so_rcv*	Socket send and receive buffers, see section on socket buffers for further information.
void (*so_upcall)	*so_upcall*	In-kernel upcall to make when a socket wakeup occurs. The canonical example is nfs, which uses sockets from inside the kernel for network request servicing.
caddr_t	*so_upcallarg*	argument to be passed `so_upcall`.
int (*so_send)	*so_send*	socket receive method. This is always currently `sosend` (which eventually leads to *so_proto*'s PR_USRREQ), but might change in the future.
int (*so_receive)	*so_receive*	socket receive method. Same holds as for *so_send*.
struct mowner *	*so_mowner*	owner of the mbuf's for the socket, used to track mbufs other than socket buffer mbufs. This can be used to debug mbuf leaks. Available only when the kernel is compiled with options MBUFTRACE.
struct uidinfo *	*so_uidinfo*	socket owner information. This is currently used to limit socket buffer size.

Additionally, *so_head*, *so_onq*, *so_q0*, *so_q*, *so_qe*, *so_qlen* and *so_qlimit* are use to queue and control incoming partial connections and handle aborts.

4.2.2 Socket Buffers

The socket buffer plays a critical role in the operation of the networking subsystem. It is used to buffer incoming data before it is read by the application and outgoing data before it can be sent to the network. As noted above, a struct socket contains two socket buffers, one for each direction. A socket buffer is described by struct sockbuf in `sys/socketvar.h`.

Table 4-4. struct sockbuf members

type	name	description
struct selinfo	*sb_sel*	Contains the information on which process (if any) wants to know about changes in the socket, for example `poll` called with POLLOUT on the socket.
struct mowner *	*sb_mowner*	Used to track owners of the socket buffer mbufs, tracking enabled by options MBUFTRACE.
u_long	*sb_cc*	counter for octets in the buffer.
u_long	*sb_hiwat*	high water mark for the socket buffer
u_long	*sb_mbcnt*	bytes allocated as mbuf memory in the socket buffer. This is the sum of regular mbufs and mbuf externals.
u_long	*sb_mbmax*	maximum amount of mbuf memory that is allowed to be allocated for the socket buffer.
long	*sb_lowat*	low watermark for socket buffer. Writing is disallowed unless there is more than the low watermark space in the socket and conversely reading is disallowed, if there is less data than the low watermark.
struct mbuf *	*sb_mb*, *sb_mbtail*, *sb_lastrecord*	mbuf chains associated with the socket buffer
int	*sb_flags*	flags for the socket buffer, such as locking and async I/O information
int	*sb_timeo*	time to wait for send space or receivable data.
u_long	*sb_overflowed*	statistics on times we had to drop data due to the socket buffer being full.

Socket buffers are manipulated by the `sb*` family of functions. Examples include `sbappend`, which appends data to the socket buffer (it assumes that relevant space checks have been made prior to calling it) and `sbdrop`, which is used to remove packets from the front of a socket buffer queue. The latter is used also by e.g. TCP for removing ACKed data from the send buffer (recall that "original" TCP requires to ACK data in-order).

4.2.3 Socket Creation

A socket is created by making the system call `socket`, which is handled inside the kernel by `sys__socket30` in `kern/uipc_syscalls.c` (`sys_socket` is reserved for compat30 ABI). First, a file descriptor structure is allocated for the socket using `fdalloc`. Then, the socket structure itself is created and initialized in `socreate` in `kern/uipc_socket.c`.

`socreate` reserves memory for the socket data structure from a pool and initializes the members that were discussed in the section Socket Data Structure. It also calls the socket's protocol's `pr_usrreq` method with the PRU_ATTACH argument. This allows to do protocol-specific initialization, such as reserve memory for protocol control blocks.

4.2.4 Socket Operation

Sockets are handled through the `so*` family if functions. Some of them map directly to system calls, such as `sobind` and `soconnect`, while others, such as `sofree` are meant for kernel internal consumption.

Socket control routines take data arguments in the form of the memory buffers (mbufs) used in the networking stack. The callers of these functions must be prepared to handle mbufs, although usually this can arranged for with a calls to `sockargs`. It should be noted, that the comment above the function takes an attitude towards this behaviour:

```
/*
 * XXX In a perfect world, we wouldn't pass around socket control
 * XXX arguments in mbufs, and this could go away.
 */
```

Note: In case a perfect world is some day being planned, the author should also be contacted, since he can contribute a whole lot of ideas for that goal.

The critical socket routines are `sosend` and `soreceive`. These are used for transmitting and receiving network data. They make sure that the socket is in the correct state for data transfer, handle buffering issues and call the socket protocol methods for doing data access. They are, as mentioned above in the socket member discussion, not called directly but rather through the *`so_send`* and *`so_receive`* members.

4.2.5 Socket Destruction

Sockets are destroyed once they are no longer useful. This is done when their reference count in the file descriptor table drops to zero. The socket is first disconnected, if it was connected at all. Then, if the socket option SO_LINGER was set, the socket lingers around until either the timer expires or the connection is closed. After this the socket is detached from its associated protocol and finally freed.

4.2.6 Protocol Support

Each protocol (e.g. TCP/IP or UDP/IP) has operations which depend on its functionality. These are controlled through the *`so_proto`* member in struct socket. While the member provides many different interfaces, the socket is interested in two: `pr_ctloutput`, which is used for control output and `pr_usrreq`, which is used for user requests. Additionally, the socket code is interested in the flags set for the protocol pointed to by *`so_proto`*. These flags are defined in `sys/protosw.h`, but examples include PR_CONNREQUIRED and PR_LISTEN, both of which the TCP protocol sets but UDP sets neither.

Control output is used to set or get parameters specific to the protocol. These are called from the kernel implementations of `setsockopt` and `getsockopt`. If the level parameter for the calls is set appropriately, the calls will trickle to the correct layer (e.g. TCP or IP) before taking action. For instance, `tcp_ctloutput` checks if the request is for itself and proceeds to query the IP layer if it discovers that the call should be passed down.

The user request method handles multiple types of different requests coming from the user. A complete list is defined in `sys/protosw.h`, but examples include PRU_BIND for binding the protocol to an address (e.g. making data received at an address:port UDP pair accepted), PRU_CONNECT for initiating a protocol level connect (e.g. TCP handshake) and PRU_SEND for sending data.

4.3 mbufs

The data structure used to pass data around in the networking code is known as an mbuf. An mbuf is described by struct mbuf, which is defined in `sys/mbuf.h`. However, it is not defined in the regular fashion, but rather through the macro MBUF_DEFINE. To understand the need for this trickery, we need to first look at the structure of an mbuf.

To accommodate for the needs of networking subsystem, an mbuf needs to provide cheap operations for prepending headers and stripping them off. Therefore an mbuf is structured as a list of constant-size struct mbufs, of which each consist of a structure header and optional secondary headers or data.

this is mostly TODO, still

4.4 IP layer

TODO

4.5 UDP

TODO

4.6 TCP

TODO

Chapter 5
Networking Services

Services built on top of the core networking functionality are described here. They include, for example, the IEEE 802.11 subsystem and the IPsec subsystem. Additionally, networking pseudo devices and their operation is described.

5.1 IEEE 802.11

The net80211 layer provides functionality required by wireless cards. It is located under `sys/net80211`. The code is meant to be shared between FreeBSD and NetBSD and therefore NetBSD-specific bits should be attempted to be kept in the source file `ieee80211_netbsd.c` (likewise, there is `ieee80211_freebsd.c` in FreeBSD).

The ieee80211 interfaces are documented in Chapter 9 of the NetBSD manual pages. This document does not attempt to duplicate information already available there.

The responsibilities of the net80211 layer are the following:

- MAC address based access control
- crypto
- input and output frame handling
- node management
- radiotap framework for bpf/tcpdump
- rate adaption
- supplementary routines such a kernel diagnostic output, conversion functions and resource management

The ieee80211 layer positions itself logically between the device driver and the ethernet module, although for transmission it is called indirectly by the device driver instead of control passing straight through it. For input, the ieee80211 layer receives packets from the device driver, strips any information useful only to wireless devices and in case of data payload proceeds to hand the Ethernet frame up to `ether_input`.

5.1.1 Device Management

The way to describe an ieee80211 device to the ieee80211 layer is by using a struct ieee80211com, declared in `sys/net80211/ieee80211_var.h`. It is used to register a device to the ieee80211 from the device driver by calling `ieee80211_ifattach`. Fields to be filled out by the caller include the underlying struct ifnet pointer, function callbacks and device capability flags. If a device is detached, the ieee80211 layer can be notified with the call `ieee80211_ifdetach`.

5.1.2 Nodes

A node represents another entity in the wireless network. It is usually a base station when operating in BSS mode, but can also represent entities in an ad-hoc network. A node is described by struct ieee80211_node, declared in `sys/net80211/ieee80211_node.h`. Examples of fields contained in the structure include the node unicast encryption key, current transmit power, the negotiated rate set and various statistics.

A list of all the nodes seen by a certain device is kept in the struct ieee80211com instance in the field *`ic_sta`* and can be manipulated with the helper functions provided in `sys/net80211/ieee80211_node.c`. The functions include, for example, methods to scan for nodes, iterate through the nodelist and functionality for maintaining the network structure.

5.1.3 Crypto Support

Crypto support enables the encryption and decryption of the network frames. It provides a framework for multiple encryption methods such as WEP and null crypto. Crypto keys are mostly managed through the ioctl interface and inside the ieee80211 layer, and the only time drivers need to worry about them is in the send routine when they must test for an encapsulation requirement and call `ieee80211_crypto_encap` if necessary.

5.2 IPSec

IPSec is collection of security-related protocols: Authentication Header (AH) and Encapsulated Security Payload (ESP). This section, however, is TODO.

5.3 Networking pseudo-devices

A networking pseudo-device does not have a physical hardware component backing the device. Pseudo devices can be roughly divided into two different categories: pseudo-devices which behave like an interface and devices which are controlled through a device node. An interface built on top of a pseudo-device acts completely the same as an interface with hardware backing the interface.

Since there is no backing hardware involved, most of these interfaces can be generated and destroyed dynamically at runtime. Interfaces that can dynamically de/allocate themselves known as cloning interfaces. They are created and destroyed by the **ifconfig** tool by using **ifconfig create** and **ifconfig destroy**, respectively. Additionally, the interface names available for cloning can be requested by **ifconfig -C**.

The list of networking pseudo-devices with short descriptions is presented below.

Table 5-1. Available networking pseudo-devices

name	description
bpfilter	Berkeley Packet filter, bpf(4). Can be used to capture network traffic matching certain pattens.
loop	The loopback network device, lo(4). All output is directed as input for the loopback interface.
npf	NetBSD Packet Filter, npf(7). Used to filter IP traffic.
ppp	Point-to-Point Protocol, ppp(4). This interface allows to create point-to-point network links.
pppoe	Point-to-Point Protocol Over Ethernet, pppoe(4). Encapsulates PPP inside Ethernet frames.
sl	Serial Line IP, sl(4). Used to transport IP over a serial connection.
strip	STarmode Radio IP, strip(4). Similar to SLIP, except uses the STRIP protocol instead of SLIP.
tap	A virtual ethernet device, tap(4). The tap[n] interface is attached to /dev/tap[n]. I/O on the device node maps to Ethernet traffic on the interface and vice versa.
tun	Tunneling network device, tun(4). Similar to tap, except that the packets handled are network layer packets instead of Ethernet frames.
gre	Encapsulating network device, gre(4). The gre device can encapsulate datagrams into IP packets in multiple formats, e.g. IP protocols 47 and 55, and tunnels the packets over an IP network.
gif	the generic tunneling interface, gif(4). gif can encapsulate and tunnel IPv{4,6} over IPv{4,6}.
faith	IPv6-to-IPv4 TCP relay interface, faith(4). Can use used, in conjunction with **faithd**, to relay TCPv6 traffic to IPv4 addresses.
stf	Six To Four tunneling interface, stf(4). Tunnels IPv6 over IPv4, RFC3056.
vlan	IEEE 802.1Q Virtual LAN, vlan(4). Supports Virtual LAN interfaces which can be attached to physical interfaces and then be used to send virtual lan tagged traffic.
bridge	Bridging device, bridge(4). The bridging device is used to attach IEEE 802 networks together on the link layer.

5.3.1 Cloning devices

In the distant past, the number of pseudo-device instances for each device type was hardcoded into the kernel configuration and fixed at compile-time. This, while the fastest method for implementation, was wasteful because it allocated resources based on a compile-decision and limiting because it required recompilation when wanting to use the n+1'th device. Cloning devices allow resource allocation and resource release dynamically at runtime.

This discussion will concentrate on cloning interfaces, i.e. cloners which are created using **ifconfig**.

Most of the work behind in cloning in interface is handled in common code in `net/if.c` in the `if_clone` family of functions. Cloning interfaces are registered and deregistered using `if_clone_attach` and `if_clone_detach`, respectively. These calls are usually made in the pseudo-device attach and detach routines. Both functions take as an argument a pointer to the if_clone structure, which identifies the cloning interface.

struct if_clone is initialized by using the IF_CLONE_INITIALIZER macro:

```
struct if_clone cloner =
    IF_CLONE_INITIALIZER("clonername", cloner_create, cloner_destroy);
```

The parameters `cloner_create` and `cloner_destroy` are pointers to functions to be called from the common code. Create is responsible for allocating resources and attaching the interface to the rest of the framework while destroy is responsible for the opposite. Of course, destroy must preserve normal system semantics and not remove resources which are still in use. This is relevant with for example the tun device, where users open the `/dev/tun[n]` when using tun interface n.

A create or destroy operation coming from userspace passes through `sys_ioctl` and `soo_ioctl` before landing at `ifioctl`. The correct struct if_clone is found by searching the names of the attached cloners, i.e. doing string comparison. After this the function pointers in the structure are used.

5.3.2 Pseudo Interface Operation

The operation and attachment to the rest of the operating system kernel for a pseudo device depend greatly on the device's functionality. The following are some examples:

- The vlan interface is always attached to a parent device when operational. It sends packets by vlan encapsulating them and enqueueing them onto the parent device packet send queue. It receives packets from `ether_input`, removes the vlan encapsulation and passes it back to the parent interface's `if_input` routine.
- The gif interface registers itself as an interface to the networking stack by using `if_attach` and `gif_output` as the output routine. This output routine is then called from the network layer output routine, `ip_output` and the output routine eventually calls `ip_output` again once the packet has been encapsulated. Input is handled by `in_gif_input`, which is called via the struct protosw input routine from the encapsulated packet input function.
- The tap interface registers itself as a network interface using `if_attach`. When a packet is sent through it, it notifies the device node listener or, if corresponding device is not open, simply drops all packets. When a packet is written to the device node, it gets injected into the networking stack through the `if_input` routine in the associated struct ifnet.

5.4 Packet Filters

In principle there are two pseudo devices involved with packet filtering: npf is involved in filtering network traffic, and bpf is an interface to capture and access raw network traffic. All will be discussed briefly from the point of view of their attachment to rest of the kernel; the packet inspection and modification engines they implement are beyond the scope of this document.

npf is implemented using pfil hooks, while bpf is implemented as a tap in all the network drivers.

5.4.1 Packet Filter Interface

pfil is a purely in-kernel interface to support packet filtering hooks. Packet filters can register hooks which should be called when packet processing taken place; in its essence pfil is a list of callbacks for certain events. In addition to being able to register a filter for incoming and outgoing packets, pfil provides support for interface attach/detach and address change notifications. pfil is described on the pfil(9) manual page and is used by NPF to hook to the packet stream for implementing firewalls and NAT.

5.4.2 NPF

NPF, found in `sys/net/npf`, is a multiplatform packet filtering device useful for creating a firewall and Network Address Translation (NAT). Operation is controlled through device nodes with userland tools. NPF is documented on multiple different manual pages: npf(7), npf.conf(5) and npfctl(8) are good starting points.

5.4.3 Berkeley Packet Filter

The Berkeley Packet Filter (bpf) (`sys/net/bpf.c`) provides link layer access to data available on the network through interfaces attached to the system. bpf is used by opening a device node, `/dev/bpf` and issuing `ioctl`'s to control the operation of the device. A popular example of a tool using bpf is **tcpdump**.

The device `/dev/bpf` is a cloning device, meaning it can be opened multiple times. It is in principle similar to a cloning interface, except bpf provides no network interface, only a method to open the same device multiple times.

To capture network traffic, a bpf device must be attached to an interface. The traffic on this interface is then passed to bpf to evaluation. For attaching an interface to an open bpf device, the ioctl BIOCSETIF is used. The interface is identified by passing a struct ifreq, which contains the interface name in ASCII encoding. This is used to find the interface from the kernel tables. bpf registers itself to the interfaces struct ifnet field *`if_bpf`* to inform the system that it is interested about traffic on this particular interface. The listener can also pass a set of filtering rules to capture only certain packets, for example ones matching a given host and port combination.

bpf captures packets by supplying a tapping interface, `bpf_tap`-functions, to link layer drivers and relying on the drivers to always pass packets to it. Drivers honor this request and commonly have code which, along both the input and output paths, does:

```
#if NBPFILTER > 0
  if (ifp->if_bpf)
   bpf_mtap(ifp->if_bpf, m0);
#endif
```

This passes the mbuf to the bpf for inspection. bpf inspects the data and decides is anyone listening to this particular interface is interested in it. The filter inspecting the data is highly optimized to minimize time spent inspecting each packet. If the filter matches, the packet is copied to await being read from the device.

The bpf tapping feature looks very much like the interfaces provided by pfil, so a valid is question is debating the necessity of both. Even though they provide similar services, their functionality is disjoint. The bpf mtap wants to access packets right off the wire without any alteration and possibly copy it for further use. Callers linking into pfil want to modify and possibly drop packets.

Appendix A. *Acknowledgments*

A.1 Authors

- Julio Merino wrote most of Chapter 2 and small bits of Chapter 1. These chapters were the foundation of this book.

  The initial versions of these chapters were written as part of tmpfs' development, which was possible thanks to Google (http://www.google.com/)'s Summer of Code (http://code.google.com/summerofcode.html) 2005 program.

  Thanks also go to William Studenmund for reviewing Chapter 2 and providing multiple valuable suggestions.
- Emmanuel Dreyfus wrote half of Chapter 3. Daniel Sieger wrote Section 3.6.
- Antti Kantee wrote Chapter 4 and Chapter 5, but was kind enough to leave room for future contributions on the subject.

A.2 License

Appendix B.
Bibliography

Bibliography

[McKusick] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman, 1996, 0201549794, Addison-Wesley Professional, *The Design and Implementation of the 4.4 BSD Operating System*.

[Goodheart] Berny Goodheart, James Cox, Michael J. Karels, 1994, Prentice Hall, *The Magic Garden Explained: The Internals of UNIX System V Release 4*.