NetBSD Internals

(2006/01/28)

The NetBSD Developers

NetBSD Internals

by The NetBSD Developers

Published 2006/01/28 09:34:18

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 (0x00011338 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 file* 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 subsystem as the virtual file system (VFS) is can be difficult. The chapter starts giving an overview on both the vnode (Section 2.1) and the VFS (Section 2.2) layers as well as on the existing file systems; they should be read in this order. These three sections ought to provide a general outline on the whole subsystem, making the reader able to read and understand existing code, should he need to.

Later on, it describes all other details related to file systems implementation and continues to extend the explanations given in the layers' overview (but please note that the information is not duplicated, so a read of the overview sections is "a must"). These sections may be read in any order, as they are highly hyperlinked to ease navigation and structured, more or less, as a reference guide.

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 vnode layer 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 2.1.1.

- 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 2.1.3.

2.1.1 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 2.7.4).

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.

2.1.2 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 2.1.4).

Table 2-1. vnode operations summary

Operation	Description	See also
`VOP_LOOKUP`	Performs a path name lookup.	See Section 2.9.
`VOP_CREATE`	Creates a new file.	See Section 2.10.1.
`VOP_MKNOD`	Creates a new special file (a device or a named pipe).	See Section 2.13.
`VOP_LINK`	Creates a new hard link for a file.	See Section 2.10.2.
`VOP_RENAME`	Renames a file.	See Section 2.10.4.
`VOP_REMOVE`	Removes a file.	See Section 2.10.3.
`VOP_OPEN`	Opens a file.	
`VOP_CLOSE`	Closes a file.	
`VOP_ACCESS`	Checks access permissions on a file.	See Section 2.10.8.
`VOP_GETATTR`	Gets a file's attributes.	See Section 2.10.6.1.
`VOP_SETATTR`	Sets a file's attributes.	See Section 2.10.6.2.

Operation	Description	See also
VOP_READ	Reads a chunk of data from a file.	See Section 2.10.5.4.
VOP_WRITE	Writes a chunk of data to a file.	See Section 2.10.5.4.
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	XXX	
VOP_MMAP	Maps a file on a memory region.	See Section 2.10.5.3.
VOP_FSYNC	Synchronizes the file with on-disk contents.	
VOP_SEEK	XXX	
VOP_MKDIR	Creates a new directory.	See Section 2.12.1.
VOP_RMDIR	Removes a directory.	See Section 2.12.2.
VOP_READDIR	Reads directory entries from a directory.	See Section 2.12.3.
VOP_SYMLINK	Creates a new symbolic link for a file.	See Section 2.11.1.
VOP_READLINK	Reads the contents of a symbolic link.	See Section 2.11.2.
VOP_TRUNCATE	Truncates a file.	See Section 2.10.6.2.
VOP_UPDATE	Updates a file's times.	See Section 2.10.7.
VOP_ABORTOP	Aborts an in-progress operation.	
VOP_INACTIVE	Marks the vnode as inactive.	See Section 2.7.1.
VOP_RECLAIM	Reclaims the vnode.	See Section 2.7.1.
VOP_LOCK	Locks the vnode.	See Section 2.7.5.
VOP_UNLOCK	Unlocks the vnode.	See Section 2.7.5.
VOP_ISLOCKED	Checks whether the vnode is locked or not.	See Section 2.7.5.
VOP_BMAP	Maps a logical block number to a physical block number.	See Section 2.10.5.5.
VOP_STRATEGY	Performs a file transfer between the file system's backing store and memory.	See Section 2.10.5.5.
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 2.10.5.2.

Operation	Description	See also
VOP_PUTPAGES	Writes memory pages to the file.	See Section 2.10.5.2.

2.1.3 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 2.1.2). 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, being each entry de description of a single operation. The purpose of this vector is to define a mapping from logical operations such as vop_open or read ro 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 whishes. 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 stablished 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.

2.1.4 Executing vnode opearations

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.

2.2 VFS layer 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.

2.2.1 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.

2.2.2 VFS operations

A file system driver exposes a well-known interface to the kernel by means of a set of public opeations. 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 2.2.3).

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 2.5.
`fs_start`	Makes the file system operational.	Must be defined.	
`fs_unmount`	Unmounts an instance of the file system.	Must be defined.	See Section 2.5.
`fs_root`	Gets the file system root vnode.	Must be defined.	See Section 2.8.
`fs_quotactl`	Queries or modifies space quotas.	Must be defined.	
`fs_statvfs`	Gets file system statistics.	Must be defined.	See Section 2.6.
`fs_sync`	Flushes file system buffers.	Must be defined.	
`fs_vget`	Gets a vnode from a file identifier.	Must be defined.	See Section 2.7.3.
`fs_fhtovp`	Converts a NFS file handle to a vnode.	Must be defined.	See Section 2.14.

Operation	Description	Considerations	See also
`fs_vptofh`	Converts a vnode to a NFS file handle.	Must be defined.	See Section 2.14.
`fs_init`	Initializes the file system driver.	Must be defined.	See Section 2.4.
`fs_reinit`	Reinitializes the file system driver.	May be undefined (i.e., null).	See Section 2.4.
`fs_done`	Finalizes the file system driver.	Must be defined.	See Section 2.4.
`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 numer is bumped.

2.2.3 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 2.2.2 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 2.1.3, 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 2.13 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.

2.3 File systems overview

2.3.1 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

2.3.2 Network file systems

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

2.3.3 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

2.3.4 Layered file systems

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

2.3.5 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 2.3.4).
- specfs: Implements all operations used to deal with special files in a file system.

2.4 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.

Furthremore, 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_DEFINE(M_EGFSMNT, "egfs mount", "egfs mount structures");

void
egfs_init(void)
{

#ifdef _LKM
        malloc_type_attach(M_EGFSMNT);
#endif

        ...
}

void
egfs_done(void)
{

        ...

#ifdef _LKM
        malloc_type_detach(M_EGFSMNT);
#endif
}
```

2.5 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.

2.5.1 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-tunnable 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;

        ...
}
```

2.5.2 The mount utility

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

2.5.3 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 2.5.1) 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 2.5.3.1.

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

   This is further detailed in Section 2.5.3.2.

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 2.5.3.3.

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 2.5.3.4.

2.5.3.1 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));
}
```

2.5.3.2 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;
```

2.5.3.3 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.

2.5.3.4 Setting up a new mount point

If neither `MNT_GETARGS` nor `MNT_UPDATE` were set in `mp->mnt_data` 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(9) 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);
```

2.5.4 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;
```

2.6 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(9) 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`. Detailes 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;
```

2.7 vnode management

2.7.1 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 live 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 `VOP_INACTIVE` on it and, when this happens, it becomes part of 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 shutted down (e.g., due to a revoke(2)).

Note that the `kern.maxvnodes` sysctl(2) node specifies how many vnodes can be kept active at a time.

2.7.2 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).

2.7.3 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 2.9.2. 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 2.14.

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 physicall file. ach 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.

2.7.4 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 2.7.1).

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;
}
```

2.7.5 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`.

2.8 The root vnode

As described in Section 2.9, 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);
```

2.9 Path name resolution procedure

XXX Write an introduction.

2.9.1 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`.

2.9.2 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 2.9.1); 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 callling 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).

2.9.3 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).

2.10 File management

XXX: Write an introduction.

2.10.1 Creation of regular files

XXX: To be written. Describe vop_create.

2.10.2 Creation of hard links

XXX: To be written. Describe vop_link.

2.10.3 Removal of a file

XXX: To be written. Describe vop_remove.

2.10.4 Rename of a file

XXX: To be written. Describe vop_rename.

2.10.5 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.

2.10.5.1 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_t n, 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.

2.10.5.2 Getting and putting pages

As seen in Section 2.10.5.1, 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 2.10.5.5.

2.10.5.3 Memory-mapping a file

Thanks to the particular UBC implementation in NetBSD (see Section 2.10.5.2), 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.

XXX: Should describe where mmap is really handled.

2.10.5.4 The read and write operations

Thanks to the particular UBC implementation in NetBSD (see Section 2.10.5.2), 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;
```

```
}
```

2.10.5.5 Reading and writing pages

As seen in Section 2.10.5.2, 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 needs it read to bring the requested page to 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.

2.10.6 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 2.10.7 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 ficticious 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.

2.10.6.1 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;
}
```

2.10.6.2 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;
}
```

2.10.7 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 asynchronoous 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 whishes; such a situation may be when it is mounted synchronously, as it will be updating the times as soon as the changes happen.

2.10.8 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 possibilites. 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);
}
```

2.11 Symbolic link management

2.11.1 Creation of symbolic links

XXX: To be written. Describe vop_symlink.

2.11.2 Read of symbolic link's contents

XXX: To be written. Describe vop_readlink.

2.12 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;
};
```

2.12.1 Creation of directories

XXX: To be written. Describe vop_mkdir.

2.12.2 Removal of directories

XXX: To be written. Describe vop_rmdir.

2.12.3 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.

2.13 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 taks 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 2.2.3.

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`.

2.14 NFS support

XXX: To be written. Describe vop_fhtovp and vfs_vptofh.

2.15 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 2.7.2.

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 1.1 2006/01/28 09:34:18 jmmv 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 1.1 2006/01/28 09:34:18 jmmv 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 2.4.

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 2.5.

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 2.7.5.
20. Implement the `vop_reclaim` and `vop_inactive` operations to correctly destroy vnodes.

    See Section 2.7.4.
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 2.7.3.

    ```
    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 2.10.6.
25. Implement the `egfs_access` operation.

    See Section 2.10.8.
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 2.10.8.
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 2.9.
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 2.12.3.

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

    See Section 2.9.3.

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 2.9.3 and Section 2.7.4.

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 2.10.5.

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

    See Section 2.10.5.2.

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

    See Section 2.10.5.5.

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 2.11.

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

    See Section 2.13.

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

    See Section 2.14.

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
Regression testing

Regression testing is an important part of software development. Unfortunately, NetBSD does not have a consistent regression testing framework. Each subsystem defines its own set of tests in whichever form it wishes to stress test itself.

This chapter provides some guidelines on how to test different parts of the system, but please do keep in mind that the whole regression testing framework ought to be replaced with something better.

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

3.1 Testing file systems

Testing file systems this is specially important because they work within kernel space; any unexpected failure is often fatal and renders the whole system unusable. Also, because there are literally hunderds of minor details to test, none of them should suffer regressions.

The tests for a given file system are stored inside a directory named after it, placed under `src/regress/sys/fs/`. For example, the tmpfs test suite lives in `src/regress/sys/fs/tmpfs/`. Generally, this directory is accompanied by a `Makefile` whose `regress` targets executes all the tests automatically.

The author of this text suggests you to add individual and independent tests for each feature you want to check, and within these, add as many subtests as you need to ensure that the whole feature works. For example, if you wanted to verify the mkdir vnode operation, you'd write a `t_mkdir` script that checks its functionality through the mkdir(1) command. This script could check that directories can be created, that they cannot be overwritten, that their link count is updated correctly, etc.

tmpfs comes with a good set of generic tests that can be reused for other file systems.

Appendix A.
Acknowledgements

A.1 Authors

- Julio M. Merino Vidal (../../People/Pages/jmmv.html) wrote most of Chapter 2 and small bits of Chapter 1 and Chapter 3. These chapters were the foundation of this book.

  The initial versions of these chapters were as part of the tmpfs' development, 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.

A.2 License

WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Appendix B.
Bibliography

Bibliography

[4.4BSD] 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*.