OpenBSD Filesystem Howto

$Revision: 1.13 $

Sebastien Bombal
Laurent Corbes

$Date: 2004/03/07 15:48:42 $.

$Id: filesystem-howto.texi,v 1.13 2004/03/07 15:48:42 caf Exp $

Table of Contents

1 Introduction

This howto is designed to be a guide for helping someone who wanted to integrate or design a new filesystem into OpenBSD. It is based on personal experiences in the OpenBSD kernel development. Because it is not done by the OpenBSD team. This document may contains errors and future code of OpenBSD may differ from the one presented. This work is based over a 3.4 version from November 2003.

Before reading this Howto you must at least have the minimum knowledge about a filesystem, what it intend to do, . . . This will normally be the case because if you read this Howto you certainly want to introduce the support for a new filesystem in OpenBSD.

If you have any questions about this document feel free to email us, Laurent Corbes <caf@glot.net> and Sebastien Bombal <acide@bombal.org>.

The master copy of this document is located at http://glot.net/openbsd-filesystem-howto/.

1.1 Introducing OpenBSD

The OpenBSD project produces a FREE, multi-platform 4.4BSD-based UNIX-like operating system. Our efforts emphasize portability, standardization, correctness, proactive security and integrated cryptography. OpenBSD supports binary emulation of most programs from SVR4 (Solaris), FreeBSD, Linux, BSD/OS, SunOS and HP-UX.

OpenBSD is a server oriented Os intend to high security level users. With is own great fire-walling tool, pf as a very reliable Os gateway. It is also great for running untrusted programs like user cgis in a web server. It contains stack protection features that make the gain of privileges using buffer overflows very difficult.

1.2 Credits

All these people helped us (directly or not) to produce this howto.

- OpenBSD team
- EPITA SRS 2003 specialisation students
- Marc Espie <espie@openbsd.org>

2 Internal filesystem structures

This chapter will describe the internal structures you must know if you plan to add a new filesystem into OpenBSD kernel. As you know OpenBSD is based upon netBSD now. The filesystem interface has not yet evolved since netBSD fork so it can be considered very old. The interface is an implementation of the vfs interface definition. If you don't know vfs and others filesystems interfaces we recommend that you read Toward a Compatible Filesystem Interface that explain how this interface was designed and why. It's a documentation done by Berkeley BSD team comparing Unix filesystem interfaces and explaining why they used vfs into 4.4BSD.

2.1 OpenBSD vfs interface

The goal of vfs is to have a very clear separation between an independent filesystem and -dependent data structures and operations. This is done by using two layers, an independent filesystem one and an independent one. Surprising ! When adding a new filesystem in a vfs interface we simply needed to do the lower layer, the filesystem dependent routines. This low level code is then plugged into the vfs upper layer. The advantage is that in the lower layer you only need to code basic filesystem operations (check filesystem superbloc informations, write a file, read a file, ...). All the very complicated functions, in particular namei and lookup, does not needed to be rewrite.

The code of the upper layer is located in all the vfs_* files in kern/. the important structures definitions are in sys/namei.h and sys/vnode*.h. Just take a look at these files if you want to know how the kernel does the abstraction needed. A Compatible Filesystem Interface will also describes these structures that are the base of the vfs interface. The most important thing to remember is that the upper level only manipulates vnodes (virtual inodes) instead of inodes and blocks in the lower layer. The two layers communicates using vectors of operations that defined each filesystem. This is the main concept of vfs.

We only needed to know how work the interface of vfs toward a filesystem. Taking a look to sys/vnode.h, we can see a list of filesystem types. These tags are used in the vnode struct for knowing the type of filesystem we must use in the lower layer :

```
/*
 * Vnode tag types.
 * These are for the benefit of external programs only (e.g., pstat)
 * and should NEVER be inspected by the kernel.
 *
 * Note that v_tag is actually used to tell MFS from FFS, and EXT2FS from
 * the rest, so don't believe the above comment!
 */
enum vtagtype   {
  VT_NON, VT_UFS, VT_NFS, VT_MFS, VT_MSDOSFS, VT_LFS, VT_LOFS, VT_FDESC,
  VT_PORTAL, VT_NULL, VT_UMAP, VT_KERNFS, VT_PROCFS, VT_AFS, VT_ISOFS,
  VT_UNION, VT_ADOSFS, VT_EXT2FS, VT_NCPFS, VT_VFS, VT_XFS
```

```
};
...
struct vnode {
...
        enum    vtagtype v_tag;                 /* type of underlying
        data */
...
};
```

The next file is sys/malloc.h. It contains the memory type information, you must provide a type for your new fs that will be used later :

```
/*
 * Types of memory to be allocated
 */
#define M_FREE          0       /* should be on free list */
...
#define M_EXT2FSNODE    72      /* EXT2FS vnode private part */
...

#define INITKMEMNAMES { \
        "free",         /* 0 M_FREE */ \
...
        "EXT2FS node",  /* 72 M_EXT2FSNODE */ \
...
}
```

The last file for the interface is kern/vfs_conf.c. That file contains the global configuration of the vfs interface lower layer. The first thing is including external function definitions After that the array of functions pointers are built for the use of the filesystem. These structs are the vfsops (vfs operations; operations needed to initialise, mount, umount, sync, ...) and the vnodeops (virtual inode operations; operations that access inodes of the filesystem from the vnode given by the system like read, write, open, readlink, ...).

```
#ifdef FFS
#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ffs/ffs_extern.h>
#endif

#ifdef EXT2FS
#include <ufs/ext2fs/ext2fs_extern.h>
#endif
...

/*
 * Set up the filesystem operations for vnodes.
 * The types are defined in mount.h.
 */
```

```
#ifdef FFS
extern  struct vfsops ffs_vfsops;
#endif
...

/*
 * Set up the filesystem operations for vnodes.
 */
static struct vfsconf vfsconflist[] = {

        /* Fast Filesystem */
#ifdef FFS
        { &ffs_vfsops, MOUNT_FFS, 1, 0, MNT_LOCAL, ffs_mountroot, NULL },
#endif
...
}

/*
 * vfs_opv_descs enumerates the list of vnode classes, each with it's own
 * vnode operation vector.  It is consulted at system boot to build operation
 * vectors.  It is NULL terminated.
 */
extern struct vnodeopv_desc ffs_vnodeop_opv_desc;
extern struct vnodeopv_desc ffs_specop_opv_desc;
extern struct vnodeopv_desc ffs_fifoop_opv_desc;
...
struct vnodeopv_desc *vfs_opv_descs[] = {
#ifdef FFS
        &ffs_vnodeop_opv_desc,
        &ffs_specop_opv_desc,
#ifdef FIFO
        &ffs_fifoop_opv_desc,
#endif
#endif
```

As you can see the upper-lower layers interface is very clear and easy to understand. This was one of the goals of the vfs interface and, in the case of OpenBSD, it's a good implementation.

3 Implementation

Now it's time to do the job. We will make the minimal lower layer code needed to handle a new filesystem. We will not code the filesystem for you but just do the templates needed for the interface and explain what the vfs upper code expect for these functions.

3.1 lower layer structures

Let's first explain the structures you will have to use. As you have see, you must have a vector of vfs operations. This vector is defined in sys/mount.h. This is a bit surprising at first because it's clearly not the place where it would logically be found, but the definition is used in the code of mount utilities. So I think they have put this code here for facilities instead of sys/vfsops.h for example. The structure is as follows. Comments included are not in the original file but here to explain when these functions are called.

```
/*
 * Operations supported on mounted file system.
 */

struct vfsops {
// this is simply the function called when the filesystem is mount
// by the sys_mount syscall
        int     (*vfs_mount)(struct mount *mp, const char *path,
                                    void *data,
                                    struct nameidata *ndp, struct proc
                                    *p);
// nothing to do in most cases but was designed to make the filesystem
// operational
        int     (*vfs_start)(struct mount *mp, int flags,
                                    struct proc *p);
// inverse of mount, called by sys_umount
        int     (*vfs_unmount)(struct mount *mp, int mntflags,
                                    struct proc *p);
// you must return the root inode of the filesystem
        int     (*vfs_root)(struct mount *mp, struct vnode **vpp);
// here you must do operations in relation with the quota
// make quota on, off, setting quotas, getquota, ...
        int     (*vfs_quotactl)(struct mount *mp, int cmds, uid_t uid,
                                    caddr_t arg, struct proc *p);
// get the filesystem statistics
        int     (*vfs_statfs)(struct mount *mp, struct statfs *sbp,
                                    struct proc *p);
// sync the filesystem, called by sys_sync
// copy the memory buffers to the storage master
// this operation must be done when halting computer
```

```
        int     (*vfs_sync)(struct mount *mp, int waitfor,
                                    struct ucred *cred, struct proc
                                    *p);
// look up at a dinode number and find the proper vnode
        int     (*vfs_vget)(struct mount *mp, ino_t ino,
                                    struct vnode **vpp);
// file handle to vnode
// must return the vnode corresponding to the file handle
        int     (*vfs_fhtovp)(struct mount *mp, struct fid *fhp,
                                     struct vnode **vpp);
// vnode pointer to file handle
// the invert
        int     (*vfs_vptofh)(struct vnode *vp, struct fid *fhp);
// initialisation of the filesystem
// called at boot up. you must put here global initialisations of the
// filesystem if needed.
        int     (*vfs_init)(struct vfsconf *);
// if you have sysctl controling the filesystem operations
// or working modes the add them here
        int     (*vfs_sysctl)(int *, u_int, void *, size_t *, void *,
                                     size_t, struct proc *);
// in case of an exported filesystem (probably nfs) you check here if
// the remote user has the rights and allow the filesystem export
        int     (*vfs_checkexp)(struct mount *mp, struct mbuf *nam,
                                    int *extflagsp,
                                    struct ucred **credanonp);
// manage acls on filesystem (extended attributes)
// this function enables, starts, stops and disables extended attributes
// features of the filesystem
        int     (*vfs_extattrctl)(struct mount *mp, int cmd,
                                    struct vnode *filename_vp,
                                    int attrnamespace, const char *attrname,
                                    struct proc *p);
};
```

Even if the comments are clear we will explain what happens and when these functions are useful.

Let's began with the beginning, vfs_init() is called at boot up and is designed to be used for global filesystem initialisations. The work done here is global to all mounted filesystem and you must not put code related to a particular mount instance of you filesystem here. Like in the ufs implementation you can ensure that the work is done only one time by putting a static int and checking it's value at the beginning. The ufs after initialise global hash structures and call ufs_quota_init() that does the same job for quota subsystem.

After that the normal use of a filesystem is to mount a device. The sys_mount() function basically does 3 things; verify that the wanted mount device is a block device and that it is not already mounted (OpenBSD does not support multiple mounts of the same device);

initialise per mount point structures; verify that the pointed device is a good filesystem (reading superbloc, verify magic numbers, verify that the filesystem is clean, additional work depending of your filesystem and assure the next operations that are a correct filesystem). Once you have properly returned vfs upper layer code supposes that you have all the information needed to read and write to your filesystem.

Others functions are very easy to write with the above comments most of the time. If you don't understand what you need to do the more complete filesystem model in OpenBSD is ufs/ffs. In most cases it is very difficult to understand because it is a huge piece of code. I personally prefer the ext2fs implementation.

The other type of operation is the vnops (vnode operations). All these operations are related to vnode. The declaration of these vnops is a little bit more tricky than vfsops because for speed it's an association structure composed of two pointers. The operations are described in the file vnode.h. The first structure is vnodeopv_desc (vnode operation vector description) :

```
/*
 * This structure is used to configure the new vnodeops vector.
 */
struct vnodeopv_entry_desc {
        struct vnodeop_desc *opve_op;   /* which operation this is */
        int (*opve_impl)(void *);       /* code implementing this
                                           operation */
};
struct vnodeopv_desc {
                        /* ptr to the ptr to the vector where op should go */
        int (***opv_desc_vector_p)(void *);
        struct vnodeopv_entry_desc *opv_desc_ops;   /* null terminated
                                                       list */
};
```

This structure is composed of a struct that lists all the operations we can do on vnodes. The list is an association of a vnodeop_desc (that describes one type of vnode operation) and a function pointer to the operation associated. This vector must be filled by each filesystem and all the operation supported must be filled in.

```
/*
 * This structure describes the vnode operation taking place.
 */
struct vnodeop_desc {
        int     vdesc_offset;           /* offset in vector--first for
                                           speed */
        char    *vdesc_name;            /* a readable name for debugging */
        int     vdesc_flags;            /* VDESC_* flags */

        /*
         * These ops are used by bypass routines to map and locate arguments.
         * Creds and procs are not needed in bypass routines, but sometimes
```

```
         * they are useful to (for example) to transport layers.
         * Nameidata is useful because it has a cred in it.
         */
        int     *vdesc_vp_offsets;      /* list ended by VDESC_NO_OFFSET */
        int     vdesc_vpp_offset;       /* return vpp location */
        int     vdesc_cred_offset;      /* cred location, if any */
        int     vdesc_proc_offset;      /* proc location, if any */
        int     vdesc_componentname_offset; /* if any */
        /*
         * Finally, we've got a list of private data (about each operation)
         * for each transport layer.  (Support to manage this list is not
         * yet part of BSD.)
         */
        caddr_t *vdesc_transports;
};
```

All the base operations needed are already defined in vnode_if.h and vnode_if.c as these are common for all the filesystem. Only the associated pointer is different depending on the implementation. This example shows the description of the lookup operation :

```
int vop_lookup_vp_offsets[] = {
        VOPARG_OFFSETOF(struct vop_lookup_args,a_dvp),
        VDESC_NO_OFFSET
};
struct vnodeop_desc vop_lookup_desc = {
        0,
        "vop_lookup",
        0,
        vop_lookup_vp_offsets,
        VOPARG_OFFSETOF(struct vop_lookup_args, a_vpp),
        VDESC_NO_OFFSET,
        VDESC_NO_OFFSET,
        VOPARG_OFFSETOF(struct vop_lookup_args, a_cnp),
        NULL,
};
```

As all these common operations are already defined you don't really need to understand how they are defined. But if you make a really new filesystem with new concepts and vnode operations you must create the corresponding syscalls, vop_, and integrate them in upper layer vfs code. Therefore the important part of vnops declarations is the struct the system will put to your operations. Because all the operations take a void*, it's necessary to get a different structure for all the functions. The following example is the corresponding structure for lookup:

```
struct vop_lookup_args {
        struct vnodeop_desc *a_desc;
        struct vnode *a_dvp;
        struct vnode **a_vpp;
        struct componentname *a_cnp;
```

```
};
```

You now have all the concepts and internal structures needed to make your new filesystem code.

The description of all vnops can be easily found in ufs,ffs code so it's not necessary to put it here.

3.2 lower layer code

This section will guide you through your filesystem code general structure based on the existing filesystem. The easiest way to create good global filesystem environment is to copy existing arrangements. Your new filesystem should probably go into the ufs/myfs/ directory if it's an Hard Drive filesystem.

The first important file is myfs_extern.h. It's the file that will be included in the vfs interface so it must contains all the definitions of vfsops, vnops, and others global declarations. These declarations must be inner __BEGIN_DECLS and __END_DECLS macros. It must also contain the declaration as extern of the vnops pointer types. example :

```
__BEGIN_DECLS

/* myfs_vfsops.c */
int myfs_mountroot(void);
int myfs_mount(struct mount *, const char *, void *,
                    struct nameidata *, struct proc *);
int myfs_reload(struct mount *, struct ucred *, struct proc *);
...
/* myfs_vnops.c */
int myfs_create(void *);
int myfs_mknod(void *);
int myfs_open(void *);
...
__END_DECLS

extern int (**myfs_vnodeop_p)(void *);
```

Next comes the vfsops implementation. Most of the time, this is done in myfs_vfsops.c but if you have vfsops that needs a lot of code you can put it into another file like myfs_quota.c for quota subsystem for example. The first thing to do, is to build the vfsops vector corresponding to what functions you want to call for each corresponding operation. Take care to add the function you want at the right place otherwise it will never be called at the correct moment.

```
// vfs interface operations
struct vfsops myfs_vfsops = {
        myfs_mount,
        myfs_start,
        myfs_unmount,
```

```
        myfs_root,
        myfs_quotactl,
        myfs_statfs,
        myfs_sync,
        myfs_vget,
        myfs_fhtovp,
        myfs_vptofh,
        myfs_init,
        myfs_sysctl,
        myfs_check_export,
        vfs_stdextattrctl
};
```

Once you have the implementation of each vfs operations. There's no particular problem for these function once you respect the functions declarations you have done using the correct structs in parameters. Be careful that the current kernel code is written in K&R C so the parameters are not written as usual :

```
int
myfs_foo(arg, bar)
        struct vfs_arg* arg;
        int bar;
{
...
```

If you don't want to implement a particular vfs operation that is not necessary, don't hesitate to return an EOPNOTSUPP error so the error will be reported to the user. If on the other hand, the system want you to make a silly thing that will corrupt your filesystem simply do a kernel panic. Even if it's not smart, this way you will not have a corrupted filesystem and you will be able to easily see the debugging of the kernel and what happened. This way, if you think it's a kernel bug, you will be able to contact OpenBSD core team that will help you finding and debugging the problem.

The last part is the vnops. The work is usually done in myfs_vnops.c. Unlike the vfsops part, we will begin by the implementation of the operations. All the functions have the same signature (int func (void *)) so be careful to cast the void * to the corresponding struct before using it. If you don't do that the kernel will remind you to do it. The above example is the standard coding style in the OpenBSD kernel. Take care to use ap-> and not v-> in future treatments.

```
int
myfs_foo(v)
        void *v;
{
        struct vop_foo_args /* {
                struct vnode *a_dvp;
                int a;
        } */ *ap = v;
```

```
        printf (``arg->a=%i'', ap->a);
...
```

I recommend that you simply make empty functions that always return EOPNOTSUPP and print "hello myfs_foo" for the moment. This way you will have a trace of what is done each time and you will be able to make the vnops vector quickly. It is done in two steps, the first step is to make the operations vector, while the second has the operation, vector and the double pointer needed to have a complete description.

```
/* Global vfs vnode operations for myfs. */
int (**myfs_vnodeop_p)(void *);
struct vnodeopv_entry_desc myfs_vnodeop_entries[] = {
        { &vop_default_desc, vn_default_error },
        { &vop_lookup_desc, myfs_lookup },      /* lookup */
        { &vop_create_desc, myfs_create },      /* create */
        { &vop_mknod_desc, myfs_mknod },                /* mknod */
        { &vop_open_desc, myfs_open },          /* open */
        { &vop_close_desc, myfs_close },                        /* close */
        { &vop_access_desc, myfs_access },      /* access */
        { &vop_getattr_desc, myfs_getattr },    /* getattr */
        { &vop_setattr_desc, myfs_setattr },    /* setattr */
        { &vop_read_desc, myfs_read },          /* read */
        { &vop_write_desc, myfs_write },                /* write */
...
        { NULL, NULL }
};
struct vnodeopv_desc myfs_vnodeop_opv_desc =
        { &myfs_vnodeop_p, myfs_vnodeop_entries };
```

This is the myfs_vnodeop_opv_desc that will be used by the vfs interface. For vnops if you do not want to support an operation you simply do not need to include it in the available entries. The vfs will only call entries put in the descs vector. Otherwise it will return a EINVALL error.

3.3 vfs interface integration

We will now plug your new filesystem into the vfs kernel interface. This will be simply to put the corresponding information in files seen at the beginning. The first step will be to add our new vnode type into sys/vnode.h. In the vtagtype enum we will add our tag type at the end.

```
enum vtagtype      {
          VT_NON, VT_UFS, VT_NFS, VT_MFS, VT_MSDOSFS, VT_LFS, VT_LOFS, VT_FDESC,
          VT_PORTAL, VT_NULL, VT_UMAP, VT_KERNFS, VT_PROCFS, VT_AFS, VT_ISOFS,
          VT_UNION, VT_ADOSFS, VT_EXT2FS, VT_NCPFS, VT_VFS, VT_XFS, VT_MYFS
};
```

Next sys/mount.h must be modified to be able to mount your filesystem. This is done by adding a proper define for the filesystem. This define is important because it's this name that will appear to users of the filesystem. After that you must make a new mount_myfs

program based on mount to be able to mount in user land the filesystem. This program will simply be a wrapper around mount that specify the options supported by the filesystem and the type of it (in our case MOUNT_MYFS).

```
#define MOUNT_XFS       "xfs"           /* xfs */
#define MOUNT_MYFS      "myfs"          /* MYFS The future best
                                           filesystem in the world */
```

The next define must be done in sys/malloc.h. It will represent the type of node to malloc when the system or you want to malloc a myfs vnode. It's not necessary because you can for example, use your proper tags but it's the easiest way to do it. You must use a value not yet used by another filesystem. For example, 73 should be fine.

```
/*
 * Types of memory to be allocated
 */
#define M_FREE          0       /* should be on free list */
...
#define M_MYFSNODE      73      /* myfs vnode private part */
```

Now the last part of the integration, the vfs interface configuration. As seeing before the vfs configuration is done in kern/vfs_conf.c. The first thing to do is to include the myfs_extern.h where you have all the definitions. Take care that all the following codes must be inner #ifdef MYFS blocks, because you only want to compile this part of the kernel and not the entire myfs code. Otherwise if you include this code without defining MYFS in the kernel configuration, your kernel won't compile because of unresolved symbols.

```
...
#ifdef MYFS
# include <ufs/myfs/myfs_extern.h>
#endif
...

/*
 * Set up the filesystem operations for vnodes.
 * The types are defined in mount.h.
 */
#ifdef MYFS
 extern struct vfsops myfs_vfsops;
#endif
...

/*
 * Set up the filesystem operations for vnodes.
 */
static struct vfsconf vfsconflist[] = {
...
        /* my filesystem */
#ifdef MYFS
```

```
        { &myfs_vfsops, MOUNT_MYFS, 18, 0, MNT_LOCAL, myfs_mountroot, NULL },
#endif
...
};

/*
 * vfs_opv_descs enumerates the list of vnode classes, each with it's own
 * vnode operation vector.  It is consulted at system boot to build operation
 * vectors.  It is NULL terminated.
 */

extern struct vnodeopv_desc myfs_vnodeop_opv_desc;
...
struct vnodeopv_desc *vfs_opv_descs[] = {
        &sync_vnodeop_opv_desc,

#ifdef SRSFS
        &srsfs_vnodeop_opv_desc,
#endif

        NULL
};
```

This is simple. All you need to do is to insert your structures into the needed vectors. Beware to not forget a part or the system will crash at the first access of your filesystem. All this part is explained in the comments, this will normally not cause any problems.

4 Conclusion

OK, you have survived all this horrible work. You should be able to have a working filesystem structure. I hope this howto has helped you and permit to not waste time understanding by hand the OpenBSD vfs interface. We are pretty sure that there are few mistakes so if you find one please email us so the error will be corrected. This document doesn't explain big parts of the OpenBSD kernel but you should already know how to add files to be compiled.