VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PORTING OF REDIRFS ON OTHER OS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MIROSLAV ZELENÝ
AUTHOR

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PORTACE REDIRFS NA JINÉ OS
PORTING OF REDIRFS ON OTHER OS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MIROSLAV ZELENÝ
AUTHOR

VEDOUCÍ PRÁCE Ing. TOMÁŠ KAŠPÁREK
SUPERVISOR

BRNO 2007

Abstrakt

Tato diplomová práce popisuje proces portace RedirFS na operační systém FreeBSD. Zdrojovým operačním systémem je Linux. Tyto systémy jsou popsány z pohledu VFS, čímž jsou určeny rozdíly a potřebné modifikace. Poté následuje implementace.

Klíčová slova

RedirFS, Linux, FreeBSD, operační systém, jádro, modul, soubor, adresář, symbolický odkaz, souborový systém, VFS, superblock, inode, dentry, vnode, cache, operace.

Abstract

This thesis describes process of RedirFS portation to FreeBSD operating system. Source operating system is Linux. Those two operating systems are described from VFS view to determine differences and needed modifications. Then follows implementation of port.

Keywords

RedirFS, Linux, FreeBSD, operating system, kernel, module, file, directory, symbolic link, file system, VFS, superblock, inode, dentry, vnode, cache, operation.

Citace

Miroslav Zelený: Portace RedirFS na jiné OS, diplomová práce, Brno, FIT VUT v Brně, 2007

Portace RedirFS na jiné OS

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Kašpárka.

....
Bc. Miroslav Zelený
22. května 2007

Contents

Chapter 1

Prologue

Security is one of requirements of every system. Good data access design is basic, but not enough part of security. Every implementation contains some bugs and is based on some aspects. One of them is human operation. In this case, system break is not impossible. Viruses, worms and others dangerous programs can be in system activated. Then file access control is solid, because antivirus program can detect stored or loaded dangerous code in real time.

Chapter 2 describes Linux operating system and its structure. Contains description of Linux driver implementation and VFS operations and objects.

Chapter 3 contains informations about important parts of FreeBSD operating system and its kernel, alike chapter 2 about Linux. Used version of FreeBSD is 6.0 in this thesis.

Chapter 4 presents RedirFS framework, its goals and implementation. Further is described RedirFS filter interface.

Chapter 5 summarize differences between source and destination operating systems and needed modifications. Also describes solutions of those problems in other projects. Main part of this chapter describes implementation on destination platform.

Chapter 2

Linux

2.1 Introduce to Linux

Linux is an open source operating system kernel created by Finland's student Linus Torvalds, who was interested in Minix. In 1991 made this small unix-like system Linus inspired to develop his own system for Intel 80386 microprocessor. After time was Linux ported to AMD x86-64, ARM, Alpha, CRIS, DEC/VAX, H8/300, Hitachi SuperH, HP PA-RISC, IBM S/390, Intel IA-64, MIPS, Motorola 68000, PowerPC, SPARC, UltraSPARC and v850 platforms. Very stable Linux version 1.0 was released in year 1994. From version 2.0 are supported multiprocessor computer systems. In the course of time grow Linux in worldwide project. Although Linux is unix-like operating system, it is writen from begin. Linux implements Unix API, defined by POSIX. This fact makes Linux in some degree compatible with other unix-like systems and brings relief in application porting.

Linux was achieved because it is noncommercial project and is distributed for free under GNU General Public Licence version 2.0. Important step of Linux was association with project GNU (GNU's Not UNIX), which goal was complete noncommercial unix-like operating system with sort of applications. In GNU project was developed microkernel HURD. But its development was complicated and with not very good results. By porting GNU applications to Linux arise new platform called GNU/Linux. When arrived project XFree86 - free implementation of XWindow and grafic user interfaces was GNU/Linux usage expanded to desktop. Because kernel and applications was distributed separately, build of complete system was difficult. Then appeared linux distributions, which contained installation program, build kernel and applications. Most popular are Debian, RedHat, Suse, Slackware and in last time Mandriva. Last stable tree of Linux kernel is 2.6 and is used in most of distributions.

2.2 Kernel and device drivers

This section describes device drivers problem solution in Linux. Kernels of lots of unix-like operating systems are monolithic. This means, that kernel is one static binary file. Concept of Linux kernel allows some sections separate out and store in independent modules. These modules is possible dynamically in runtime to load from file system to operating memory and to remove from memory. A Linux system make a difference between two process run modes. One of them is called user-mode and the other kernel-mode. These modes are implemented in hardware. In processors Intel 80386 and compatible is Ring 0 used as user-mode and Ring 3 is used as kernel-mode. Processor modes Ring 1 and Ring 2 are not used. Process in user-mode has not direct access to hardware. That is why are device drivers implemented in kernel, which run in kernel-mode with direct access to hardware.

Devices drivers is possible to build as Linux kernel modules. But it is not possible in all way. Because modules are stored on file system, all drivers needed to system start and to mount file system containing modules, have to be statically compiled in linux kernel. Only then are modules accessible for loading to memory. Namely disk interface driver and file system driver. Devices can be character or block oriented. Character-oriented devices are accessible sequentially, byte after byte. Drivers of these devices contain no buffer. For example RS232 interface, sound devices or software device *null*. Block-oriented devices have different data access. Reading and writing proceed by transfers multiple of block sizes. In some operating systems is possible transfer of other size than multiple of block size. In this case are used buffers for access. Linux belong to these systems. Typical block-oriented device is hard disk.

2.3 Special files

To access device by application in user-space, operating system has to provide some interface. Unix-like systems including Linux solve this problem by special files stored on file system. These special files allow work with devices as with regular files and apply to them standard system calls. Exception in Linux and some others operating systems is network device.

Special files suppling devices have major and minor number. Major number determines type of device and identifies corresponding device driver. In Linux kernel tree 2.6 is this number 12-bit long. There is 4096 types of devices. Minor number determines concrete device of relevant type. Length of this number is 20 bits. In older Linux kernel trees were major and minor number 8-bit long. If special file from user-space is accessed, then kernel use corresponding device driver for serve system call. So special file name plays no role. Special file is possible to create on disk by command *mknod*. First argument is name of file. Second argument indicate type of device: *b* for block-oriented devices, *c* for character-oriented devices. Next arguments are major and minor numbers.

2.4 Kernel modules

Style of kernel module writing and compilation is in Linux kernel tree 2.6 other than in older trees of Linux kernel. For developing Linux kernel module are typically used header files *linux/module.h*, *linux/config.h* and *linux/init.h*. These files we can found in kernel source in directory *include*. Because we write driver for kernel-space, we cann't use libraries designed for user-space. General functions are declared in header file *linux/kernel.h*. For example often used function *printk*, equivalent to *printf* from user-space.

Modul contains init routine called by loading module and cleaning routine called by removing module from operating memory. Initialization function looks like this:

static int __init name_of_init_routine (void) and provide linking other functions to kernel. Return value represent result of initialization. Non-zero value means failure. In this case is kernel module removed from memory. Cleaning function:

static void __exit name_of_cleanup_routine (void)

returns used sources. These two basic functions are given by macros

module_init (name_of_init_routine) and *module_exit (name_of_cleanup_routine).*

In *Makefile* dedicated for module compiling is definition of module name required (*obj-m := modul.o*). We can run compilation by command

make -C $(KDIR) SUBDIRS=$(PWD) modules

where variable *KDIR* substitutes kernel source path and variable *PWD* substitutes actual directory of module source. This command cause settings of paths needed for compilation and realize compilation. Created module has *".ko"* extension. We can load this module to operating memory by *insmod* command. If module is present in reserved directory for kernel modules, then it is possible to use *modprobe* command. We can remove loaded module from operating memory by the help of *rmmod* command and show list of loaded modules via *lsmod* command. Linux kernel modules can we found in tree structure of directories beginning with directory named as linux kernel version (e.g. "2.6.18") and stored in */lib/modules* directory.

2.5 File access

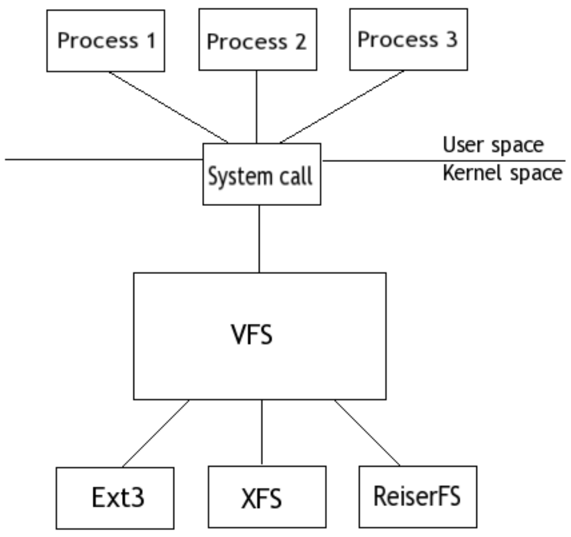
Unix-like systems have all files stored in one tree of directories beginning with root directory */*. Access to discs and other file systems is assured by mounting. Desired disc can be connected to some existing directory of file system. All mounted discs create just one directory tree. For disc mounting is used command *mount* and for unmounting command *umount*. In system have not to be mounted only discs and other media, but also virtual file systems. For example *procfs* mounted to directory */proc* and used for system information serve. Type of file system is determined by mounting. On the type depends used file system driver. For real file systems is possible to use auto detection of file system via identification numbers stored in table of disc allocation and in head of file system.

2.6 Structure of file system

Every disc file system has some defined data organization. Beside alone data is needed to store informations about files and directory structure. Unix-like systems maintain to use hard links, which allows access to one file with more names and from more directories. Its why are so-called i-node used, which are identified by number. I-node is structure containing all properties about file except name. Among others contains informations about data storing and number of files using this i-node. Directories are represented as files, but have directory flag set. Data of directory are names of files and numbers of i-nodes. Symbolic links, pipes and special files are marked by correspondent flag.

2.7 Linux VFS

First version of Linux allowed only one file system. The one was file system of Minix. In present time allows Linux to use lots of file system types. For example *ext2*, *ext3*, *xfs* or *vfat*. Because every file system has different structure, properties, types of items and attributes of items, is abstraction layer in kernel needed. For this purpose is VFS (Virtual File Switch) destined. This layer is found between system call and file system driver.



Picture 2.1: *The Linux VFS.*

There are defined objects and operations, which form VFS model. Main objects are:

- **superblock** containing informations about mounted file system.
- **inode** representing file.
- **dentry** as an access point to directories.
- **file** representing opened file in context of user process.

2.7.1 Superblock

This structure is created by file system driver by mounting. If the file system is not virtual, then informations are reading from disc. Structure *super_block* is defined in file *include/linux/fs.h* much like some next objects. Important items are:

- **struct super_operations s_op** contains functions for work eith this object.
- **unsigned long s_flags** - file system mount flags.
- **unsigned long s_magic** - identification number of file system.
- **struct dentry *s_root** is entry for root directory of mounted file system.

Structure *super_operations* contains pointers to lowest level functions of VFS. There are for example functions for reading and writing superblock and for i-nodes handling.

2.7.2 Inode

This *inode* object contains all information needed for file and directory serving. It is representation of i-node from file system in memory, in case of unix file system. For example file system *vfat* do not contains i-nodes. In this case coherency between files and i-nodes is secured by file system driver. Main items of *inode* structure are:

- **unsigned long i_ino** is identification number of i-node.
- **atomic_t i_count** - number of references.
- **kdev_t i_rdev** - number of device.
- **struct inode_operations *i_op** is structure of pointers to i-node serve functions.
- **struct file_operations *i_fop** like previous, but for files.
- **struct super_block *i_sb** - reference to superblock of file system owning this i-node.

Then are in structure all file properties stored in i-node on disc. The most interesting is structure *inode_operations*. Nonzero values of included functions mean error:

- **int (*create) (struct inode *, struct dentry *, int, struct nameidata *)** - for creation new file given by inode argument in directory. It has to alloc i-node, set its items and connect to dentry.
- **struct dentry * (*lookup) (struct inode *, struct dentry *, struct nameidata *)** - serve to i-node search given by name contained in dentry.
- **int (*link) (struct dentry *, struct inode *, struct dentry *)** - goal of this function is to create hard link. It is called by systel call *link*.
- **int (*unlink) (struct inode *, struct dentry *)** - called by systel call *unlink*.
- **int (*symlink) (struct inode *, struct dentry *, const char *)** - create symbolic link by system call *symlink*.
- **int (*mkdir) (struct inode *, struct dentry *, int)** - like as function *create* for directory.
- **int (*rmdir) (struct inode *, struct dentry *)** - sence of this function is to remove directory (given by dentry) from directory given by inode.
- **int (*mknod) (struct inode *, struct dentry *, int, dev_t)** - this function creates item in directory. It is called for pipes and special files of devices. Function can be called from another function and so used for creation of regular file or directory. Type of file and relevant major number are given by arguments.

Here is list of other functions: *rename*, *readlink*, *follow_link*, *truncate*, *permission*, *setattr*, *getattr*, *setxattr*, *getxattr*, *listxattr*, *removexattr*.

2.7.3 Dentry

Object *dentry* is second of main VFS objects. Its definition can we find in file *include/linux/dcache.h* of kernel source and is used as access point for searching or walking path. Dentry can be associated with one i-node. But i-node can by associated with more dentry. This fact reflects logic of hard links, which are in VFS realized. Heavy items of dentry are:

- **atomic_t d_count** count of dentry references in user space.
- **struct inode *d_inode** is i-node associated with this object.
- **struct dentry_operations d_op** - *dentry* object operations.
- **struct dentry *d_parent** points to parent directory *dentry* object.
- **struct qstr d_name** name of directory item.

Dentry can be in tree states:

- **used** - object has associated some i-node (*d_inode* is not *NULL*) and item *d_count* is positive number.
- **unused** - object has associated some i-node (*d_inode* is not *NULL*), but item *d_count* is zero.
- **negative** - object has not associated any i-node (*d_inode* is *NULL*).

VFS uses dentry cache. It preserves repeating of directory walking, what is time difficult. Informations about dentry caching are next items of this object. For directory walking is used function *lookup* of object *inode*. Function serving system calls executes *lookup* function of directory i-node. It will find i-node of this item and if i-node exists, then associate them with dentry.

There are functions for serve dentry operations. Usually these functions are not in drivers implemented and standard kernel functions are used. Defined structure *dentry_operations* has items:

- **int (*d_revalidate)(struct dentry *, struct nameidata *)** - verification of validity.
- **int (*d_hash) (struct dentry *, struct qstr *)** - creates hash value for dentry.
- **int (*d_compare) (struct dentry *, struct qstr *, struct qstr *)** - compares names of files. It is used for example in *vfat* file system driver. There are file names case insensitive.
- **int (*d_delete)(struct dentry *)** - called when *d_count* decreases to zero.
- **void (*d_release)(struct dentry *)** - called by VFS by taking dentry out of memory.
- **void (*d_iput)(struct dentry *, struct inode *)** - called when dentry loses i-node association.

2.7.4 File

This VFS object is data representation of file for user processes. Object is created in memory when file opening by system call *open* is succeed. System call *close* deletes this object. Object *file* points to object *dentry* and then points to *inode*. Heavy properties of *file* object are:

- **struct dentry *f_dentry** - reference to object *dentry*.
- **struct file_operations *f_op** - table of file operations. By *file* object creation VFS sets its items to values given by item *i_fop* of associated i-node.
- **atomic_t f_count** means count of references of this object.
- **unsigned int f_flags** - flags of file opening.
- **mode_t f_mode** - mode of file opening.

By requests for file operations are calling functions given by structure *file_operations*. File operations for files and device drivers makes unified interface. Negative return value means error. Heaviest of them are continues:

- **loff_t (*llseek) (struct file *, loff_t, int)** - changes position of cursor in file. The new position is returned.
- **ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)** - read data from device (from application view) and returns count of readed characters.
- **ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)** - write data to device.
- **int (*readdir) (struct file *, void *, filldir_t)** - is designed for search of directory context and is used by file systems.
- **int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)** - for device control by applications. Device drivers use it often.
- **int (*open) (struct inode *, struct file *)** - called by file opening.
- **int (*release) (struct inode *, struct file *)** - called by file closing.

2.7.5 File lookup

File name lookup in Linux kernel is implemented in function

*int fastcall path_lookup(const char *name, unsigned int flags, struct nameidata *nd).*

First argument *name* is pathname of wanted file system item. Options given by *flags* can be followings:

- **LOOKUP_FOLLOW** - follow links
- **LOOKUP_DIRECTORY** - directory required

- **LOOKUP_CONTINUE** - ending slashes will be accepted even for nonexistent files
- **LOOKUP_PARENT** - path has more parts
- **LOOKUP_NOALT** - *dcache_lock* will be locked after *path_lookup* call
- **LOOKUP_REVAL** - revalidate dentry cache

Next options tell, what will be with wanted file done:

- **LOOKUP_OPEN**
- **LOOKUP_CREATE**
- **LOOKUP_ACCESS**
- **LOOKUP_CHDIR**

The last parameter *nd* of function *path_lookup* is a pointer to *nameidata* structure. Into this structure is stored lookup result:

```
struct nameidata {
        struct dentry *dentry;
        struct vfsmount *mnt;
        struct qstr last;
        unsigned int flags;
        int last_type;
        unsigned depth;
        char *saved_names[MAX_NESTED_LINKS + 1];
        union {
                struct open_intent open;
        } intent;
};
```

Item *dentry* is a pointer to *dentry* object corresponding to wanted file pathname. Second interesting item *mnt* points to structure representing mount point, where is wanted path placed in. Array *saved_names* contains parsed pathname.

Return value of *path_lookup* is zero if succeed. In other case this function returns any error value.

For *nameidata* structure release is in Linux defined *void path_release(struct nameidata *nd)* function.

Chapter 3

FreeBSD

3.1 About FreeBSD

This operating system is derived from BSD developed in Berkeley at the University of California. BSD is one of two main trees of unix systems. FreeBSD supports alpha, amd64, i386, ia64, pc98 and sparc64 platforms. Is distributed as a complete operating system containing kernel with drivers and user applications. It is an open source project started in 1993 and released under the terms of BSD and next licences. FreeBSD has a hard position in server solutions.

3.2 Kernel

The FreeBSD kernel is the part of operating system running in protected mode of processor and makes layer between hardware and user-space processes, similarly to Linux kernel. Basic parts of kernel provide processes, file system, communications and system startup. Source code of kernel is in /usr/src/sys directory. FreeBSD kernel supports kernel modules.

3.3 Kernel modules

FreeBSD contains Dynamic Kernel Linker (KLD). It allows to load and remove kernel modules in run-time. It makes development of device drivers easy without constantly rebooting. Commands for KLD operations are following:

- **kldstat** - prints all loaded modules
- **kldload** - loads kernel module to system
- **kldunload** - remove kernel module from system

By contrast to Linux, FreeBSD kernel modules are more integrated to kernel design. Makefile for kernel module is simple:

```
SRCS=example.c
KMOD=example

.include <bsd.kmod.mk>
```

Variable *SRCS* is a list of source files needed for compilation. Files are in the list separated by space. Other variable *KMOD* is a name of destination file.

Modules with different functions has different implementations. Function is given by used macro for module specification. Possible macros are DECLARE_MODULE, DEV_MODULE, DRIVER_MODULE and SYSCALL_MODULE declared in sys/module.h header file. DECLARE_MODULE macro is general and and is used by others. Full declaration is DECLARE_MODULE(name, moduledata_t data, sub, order) and its argument are:

- **name** - name of module used in SYSINIT macro
- **data** - pointer to the moduledata_t structure; contains the official name of the module and pointer to the event callback function
- **sub** - possible values are given by system_sub_id enumeration
- **order** - initialisation order of subsytem KLD; valid values are in sysinit_elem_order enumeration

Event callback function is used for events given by parameter:

- **MOD_LOAD** - module is loading to system
- **MOD_UNLOAD** - module is removing from system
- **MOD_SHUTDOWN** - kernel panic

3.4 VFS

Basic principle of FreeBSD VFS is similar to Linux VFS. One of differences is in buffering. FreeBSD can work with blocks of different sizes and file system driver is more complicated. In FreeBSD terminology is inode used only for disk inode. Structure in memory representing this object is called vnode.

3.4.1 Vnode object

This object contains informations independent on concrete file system. Main items are:

- **enum vtype v_type** - type of vnode specify regular file, directory, etc.
- **struct vop_vector *v_op** - structure of vnode operations
- **int v_usecount** - count of vnode using
- **struct vnode *v_dd** - pointer to .. vnode

Next items have functions:

- pointer to private data of file system
- vnode locking

- mutex protecting flags and counters
- name cache
- connecting to next structures of VFS

Enumeration type *vtype* can specify these file types:

- **VNON** - no type specified
- **VREG** - regular file
- **VDIR** - directory
- **VBLK** - block device special file
- **VCHR** - character device special file
- **VLNK** - symbolic link
- **VSOCK** - named socked
- **VFIFO** - fifo
- **VBAD** - bad type

Structure vop_vector contains pointers to functions serving different operations of vnode and pointer to default vop_vector called vop_default used when given operation function is not set. Interesting functions are following:

- **vop_open** - called by file opening
- **vop_close** - called by file closing
- **vop_create** - called when the file is creating
- **vop_getattr** - get file attributes
- **vop_link** - serve inode referencing
- **vop_lookup** - filename lookup used for directory
- **vop_mkdir** - directory is creating
- **vop_mknod** - called when other file types are creating
- **vop_read** - reading data from file
- **vop_readdir** - reading directory content
- **vop_remove** - called when file is removing
- **vop_rename** - serve renaming of file
- **vop_rmdir** - called when directory is removing

- **vop_getattr** - set file attributes
- **vop_symlink** - create a symbolic link
- **vop_write** - writing to file

Each of these functions takes a special structure as parameter. It contains specific arguments for given operation.

Name of file is not stored in structure *vnode*. It is placed to vnode cache in conjunction with given vnode. Detailed description of vnode cache is not destination of this chapter and will be described later.

Structure *vnode* is defined in *sys/vnode.h* header file. This file requires automatically produced *vnode_if.h* file to be included. To ensure *vnode_if.h* automatically generation is needed to add name of this file to *SRCS* variable defined in *Makefile*. In conjunction with *vnode_if.h* will be other needed files generated. Their names are *vnode_if_typedef.h* and *vnode_if_newproto.h*.

3.4.2 File object

File object of FreeBSD is very conformable to Linux *file* object. It is defined as *struct file* containing these important items:

- **LIST_ENTRY(file) f_list** - linking of active file list
- **short f_type** - type of file descriptor
- **void *f_data** - specific data of given descriptor type
- **u_int f_flag** - file open flags
- **struct mtx *f_mtxp** - lock
- **struct fileops *f_ops** - file operations vector
- **int f_count** - reference counter
- **struct vnode *f_vnode** - corresponding vnode (or NULL)
- **off_t f_offset** - reader/writer position

Options stored in *f_flag* can be followings:

- **O_RDONLY** - read only
- **O_WRONLY** - write only
- **O_RDWR** - read and write
- **O_ACCMODE** - mask for *O_WRONLY* and *O_RDWR* flags
- **O_SHLOCK** - shared file lock
- **O_EXLOCK** - exclusive file lock

- **O_ASYNC** - send signal pgrp when data ready
- **O_FSYNC** - synchronous writing
- **O_SYNC** - equal to *O_FSYNC*
- **O_NOFOLLOW** - do not follow symbolic links
- **O_CREAT** - create file if not exists
- **O_TRUNC** - set length to zero
- **O_EXCL** - do not open existing file
- **O_NOCTTY** - no control terminal
- **O_DIRECT** - bypass buffer cache if possible

Vector of file operations defined by *struct fileops* type contains pointers to functions serving given operations:

- **fo_rdwr_t *fo_read**
- **fo_rdwr_t *fo_write**
- **fo_ioctl_t *fo_ioctl**
- **fo_poll_t *fo_poll**
- **fo_kqfilter_t *fo_kqfilter**
- **fo_stat_t *fo_stat**
- **fo_close_t *fo_close**

Structure *struct fileops* contains item *fo_flags* of type *fo_flags_t* moreover. This item can carry these flags:

- **DFLAG_PASSABLE** - can be used by sockets
- **DFLAG_SEEKABLE** - seekable access

3.4.3 File lookup

User-space processes differentiate files by pathname, but kernel works with vnode object. So transformation from pathname to vnode has to be implemented. Kernel walk vnodes from root directory or current directory by the help of lookup functions. Result of translation is vnode of requested file.

For file name lookup is structure *nameidata* defined. There have to be set some input items and called lookup function storing result into output items of structure *nameidata*. Declared *nameidata* variable has to be initialized via function

*void NDINIT(struct nameidata *ndp, u_long op, u_long flags, enum uio_seg segflg, const char *namep, struct thread *td).*

Argument *ndp* is a pointer to *nameidata* structure, what has to be initialized. Posibble operations given by option *op* are *LOOKUP*, *CREATE*, *DELETE* and *RENAME*. Operation flags *flags* can be followings:

- **LOCKLEAF** - lock found vnode
- **LOCKPARENT** - parent vnode will be locked
- **WANTPARENT** - parent vnode will be unlocked
- **NOCACHE** - search only in cache
- **FOLLOW** - resolve symbolic links
- **LOCKSHARED** - shared lock leaf
- **NOFOLLOW** - do not resolve symbolic links

Parameter *segflg* tells about placement of path name in memory given by *namep*. Possible options are *UIO_USERSPACE* and *UIO_SYSSPACE*. First means, that memory is located in user space, the other option means kernel space. Last parameter *td* is a pointer to thread. In context of this thread will be locked structures during file lookup. Usually is used current thread represented by kernel macro *curthread*.

File name lookup is implemented in function *int namei(struct nameidata *ndp)*. The one argument is pointer to initialized *nameidata* structure. In accordance to items given via *nameidata* initialisation, *namei* function tries to find vnode object pertaining to path. Result is stored to following items of *nameidata*:

- **struct vnode *ni_vp** - vnode of found file system item
- **struct vnode *ni_dvp** - vnode of intermediate directory

Return value of *namei* function is 0 if succeed, otherwise any of defined FreeBSD kernel errors.

More complex is function *int lookup(struct nameidata *ndp)*. In compare to *namei* can do *CREATE*, *DELETE* and *RENAME* operations given during *nameidata* initialization. Function *lookup* can resolve symbolic links. Before call can be set items of *nameidata*:

- **struct vnode *ni_startdir** - start directory vnode of search
- **struct vnode *ni_rootdir** - root directory vnode
- **struct vnode *ni_topdir** - logical top directory vnode
- **struct componentname ni_cnd** - contains informations passed through the VOP interface

Unused *nameidata* structure would be released via function *void NDFREE(struct nameidata *ndp, const u_int flags)* before its memory release, because during *NDINIT* call is this structure chained to kernel structure. Interesting options given by *flags* can be followings:

- **NDF_NO_STARTDIR_RELE** - start directory vnode will not be released
- **NDF_NO_VP_RELE** - releases found vnode
- **NDF_NO_VP_PUT** - applies put function to vnode counter of use
- **NDF_NO_VP_UNLOCK** - unlocks found vnode

Chapter 4

RedirFS

4.1 About project

RedirFS project means REDIRecting File System and arise in 2005 as thesis of Ing. Frantisek Hrbata, student FIT VUT Brno. Purpose of RedirFS was to solve deficiencies of existing similar projects allowing access control. These projects often change kernel source code. RedirFS project was developed for Linux kernel tree 2.6. This chapter shows basic informations about ResdirFS. More informations can be found on home website of project http://www.redirfs.org.

4.2 Goals of RedirFS

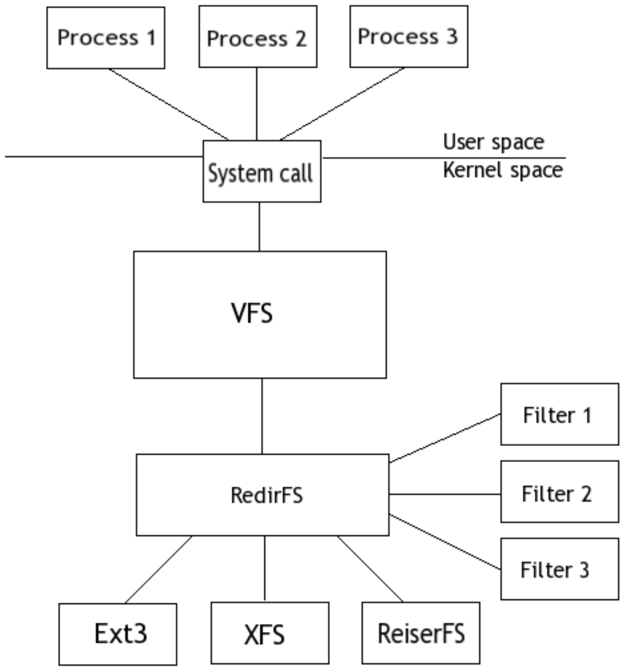
Here are main goals of RedirFS given before development:

- general, fast and flexible framework
- open source solution
- no Linux kernel modifications
- implementation as kernel module independent on kernel version
- callback interface allowing registration and unregistration of selected file operations for third-party filters
- pre and post callbacks
- selection of interested directory subtrees for filters
- calling filters in given order
- reaction on return value from filters

4.3 Implementation

Idea of RedirFS is to offer interface to third-party filters, implemented as kernel modules. Filter can register to the RedirFS framework and get handler used for following interaction. During registration is priority set. This determine order of several filters, in what have to be called. Filter can dynamically set, modify or remove callback functions and set attached directory subtrees. Next possible function of filter is to activate or deactivate itself.

RedirFS defines several objects. Path object represents start directory of subtree attached by filter. Those path objects are linked in tree. Every filter is represented by the filter object attached to path objects. Next object of RedirFS framework is rinode object. It is different object to VFS inode, but determine corresponding path object to the VFS inode.



Picture 4.1: *RedirFS.*

4.4 Download and installation

The actual source code of this project can be downloaded via *svn co http://redirfs.org/svn/redirfs* command. Note this is designed for Linux kernel tree version 2.6. After extracting is possible to build kernel module by the help of *make* command. Created Linux kernel module redirfs.ko can loaded to system via *insmod redirfs.ko.* Latest version used for this thesis was downloaded in 21. April 2007. There some problems with Linux kernel compatibility occurred. This version of RedirFS use some data structures implemented from Linux kernel version 2.6.17. When in use Linux kernel version 2.6.21, the following problem arisen.

Structure *path* of RedirFS framework conflicts with *path* structure of Linux kernel, which is newly defined in used *linux/namei.h* header file.

4.5 RedirFS filter

All needed declarations for RedirFS filter writing are stored in header file *redirfs.h* of RedirFS project. This file has to be included in RedirFS filter source code, what is implemented as a kernel module.

Filter handler

This variable is initialized via filter registering and is used for others RedirFS functions calling. Data type of this hadler has to be *rfs_filter*. Return value of RedirFS public functions can be:

- **RFS_ERR_OK** - no error occurred
- **RFS_ERR_INVAL** - invalid value
- **RFS_ERR_NOMEM** - memory allocation error
- **RFS_ERR_NOENT** - no entry point
- **RFS_ERR_NAMETOOLONG** - too long name of path
- **RFS_ERR_EXIST** - file exists
- **RFS_ERR_NOTDIR** - destination of path is not a directory

Filter registration and unregistration

Each filter has to be registered in RedirFS. Prototype of registration function is

*enum rfs_err rfs_register_filter(rfs_filter *filter, struct rfs_filter_info *filter_info)*

and is usually called by kernel module initialization. Structure filter_info sets filter parameters:

```
struct rfs_filter_info {
        const char *name;
        int priority;
        int active;
};
```

Item *name* is name of filter a is used for */proc* informations. Priority determines order of filters. Lower number is higher priority. Last item *active* sign filter activation.

Unregistering function is then *enum rfs_err rfs_unregister_filter(rfs_filter filter)*.

Selecting attached directories

RedirFS framework allows to select what directories have to be attached to filter. This feature provide following function:

*enum rfs_err rfs_set_path(rfs_filter filter, struct rfs_path_info *path_info)*

.

Structure *rfs_path_info* contains informations about path:

```
struct rfs_path_info {
        const char *path;
        int flags;
};
```

Path is a string of destination directory. Flags can be following:

- **RFS_PATH_SINGLE** - operation will be applied only to this directory
- **RFS_PATH_SUBTREE** - operation will be applied to this directory and all subdirectories
- **RFS_PATH_INCLUDE** - filter touches this directory
- **RFS_PATH_EXCLUDE** - filter don't touch this directory

Setting callback function

Following function sets or replaces pre and post callback functions for filter:

*enum rfs_err rfs_set_operations(rfs_filter filter, struct rfs_op_info *op_info).*

Operations are defined by:

```
struct rfs_op_info {
        enum rfs_op_id op_id;
        enum rfs_retv (*pre_cb)(rfs_context, struct rfs_args *);
        enum rfs_retv (*post_cb)(rfs_context, struct rfs_args *);
};
```

Item *op_id* is identification of captured operation, e.g. RFS_DIR_IOP_CREATE or RFS_REG_FOP_READ. Full list of operations is available in *redirfs.h* header file. Two next items of structure are functions called before and after operation call. Note parametr *op_info* of function *rfs_set_operations* is an array of *rfs_op_info* structures. The item *op_id* of last structure of this array has to be *RFS_OP_END*.

Callback functions accept two parameters. Currently unused *rfs_context* and pointer to structure *rfs_args* carrying arguments to given operation. The *rfs_args* is an union type and for each operation contains special structure type. This makes a good abstraction and then type of callback functions can be unified. Callback functions do not receive information of type *rfs_op_id*. This makes callback function unusable for more types of operation serving. The exception is the case when *rfs_args* union is not used.

Activating and deactivating filter

After previous initializations can be filter activated and deactivated if needed:

enum rfs_err rfs_activate_filter(rfs_filter filter),

enum rfs_err rfs_deactivate_filter(rfs_filter filter).

Chapter 5

Portation

RedirFS use lots of internal Linux kernel functions and structures. Basic concept could be kept and platform dependent sections replaced. Now lets have a look at differences.

5.1 RedirFS source code overview

Internal RedirFS documentation is in the making at this time. So this section describes structure and the parting of the source code. Then will be possible to determine needed modifications and to establish procedure of porting.

Header files

- **debug.h** - defines debug macros only
- **redir.h** - contains private object structures and interconnect functions heads (included in all C-modules of RedirFS)
- **redirfs.h** - public data types and heads of public functions needed by RedirFS filter modules

C modules

Source code of RedirFS framework contains these C modules:

- **chain.c**
- **filter.c**
- **path.c**
- **proc.c**
- **rdentry.c**
- **redir.c**
- **rfile.c**

- **rinode.c**
- **ops.c**

Main module is the *redir.c*, where is a kernel module initialization. File *redir.c* contains operation replacement moreover. Module *proc* is used for */proc* file system informations and defines needed functions. Other modules implement object operations. Each of these module names corresponds to given object type.

5.1.1 Private objects

All of these objects are dynamically created and destroyed in memory. There are linked lists used. Objects of different type can be joined of course. This fact requires to keep reference count and to destroy object when is not referenced already.

filter

Object *filter* represents registered RedirFS filter in memory and stores states of filter.

path

Objects of type *path* create tree structure corresponding to destination directory placement. In the tree are objects representing registered paths from RedirFS filters only. Linking of *path* objects is the most complicated, because there are linked objects on the same level and every *path* contains a list of subpaths.

rfile, rdentry and rinode

This objects represent a touched VFS objects by RedirFS framework. Every of them is pointing to associated *path* and *chain* objects.

chain

The *chain* is a connecting object. It can be referenced in more *path*, *rfile*, *rdentry* or *rinode* objects and references all filters have to be applied on given referencing object.

ops

This object keepts informations about RedirFS operations. It is referenced by *path*, *rfile* and *rdentry* objects.

5.2 Linked lists

Structures in RedirFS are dynamically allocated and linked to lists. The most complex is structure *path* where are linked directories on same level and subdirectories.

5.2.1 Linux kernel solution

Because C language has not direct support for those lists, Linux kernel provide concept, what make list creation easy. There is several types of lists and here is the basic double linked type of list described. Others are not in RedirFS framework used. Via defined macro LIST_HEAD(name) can be head of list declared. Node of these lists can be every structure containing item of type *struct list_head*. This structure *list_head* has items needed to connection to another elements of list. Placement in list element is not important. List has to be initialized by calling *INIT_LIST_HEAD(*list_head)* with list pointer as parameter.

Add item to list is possible with help of functions *void list_add(struct list_head *new, struct list_head *head)* or *void list_add_tail(struct list_head *new, struct list_head *head)* where *new* is pointer to *list_head* item from adding structure and parameter *head* is pointer to head of requested list. Function *list_add* inserts item to begin of list and function *list_add_tail* adds item to end of list. Nodes of list can be removed via *void list_del(struct list_head *entry)* function having *list_head* structure of element as parameter. In this case head of list is not important.

Sometimes is needed to move element from any list to other list. In Linux kernel is declared function *void list_move(struct list_head *list, struct list_head *head)* make this possible. Argument *list* is element from source list and *head* is destination list, where has to be element placed.

Function *int list_empty(const struct list_head *head)* offers to check, whether list is empty. Returns logical one if is.

To access items of list is declared macro *list_entry(ptr, type, member)* returning pointer to this item. Argument *ptr* is poiter to *list_head* structure named *member* from any structure of type *type* linked to list. Macro *list_entry* is generally used in connection with some of several macros creating cycles to walk a list. Basic macro is *list_for_each(pos, head)*. First parameter *pos* is pointer to *list_head* structure, which is set to pointing to *list_head* of actual entry in every through of cycle. And *head* is as usual head of list. Next important macro is named *list_for_each_safe(pos, n, head)*. Offers the same function as *list_for_each*, but is possible to remove item inside the cycle. New parameter *n* is of type *list_head* pointer as *pos*. But in this case *n* is only temporal variable used for private stuff of *list_for_each_safe*.

Another macros to create cycle walking a list are *list_for_each_entry(pos, head, member)* and *list_for_each_entry_safe(pos, n, head, member)*. Parameter *pos* is not set to *list_head* item of element structure, however directly to element structure. Arguments *head* and *member* similar to *list_entry*. Macro *list_for_each_entry_safe* contains parameter *n* as *list_for_each_safe*.

5.2.2 FreeBSD kernel solution

FreeBSD kernel supports linked lists too. Needed types, functions and macros are declared in *sys/queue.h* header file. There are offer tree types of a list:

- List - supports adding new elements at the head of the list or after any element in the list. Removed can be any element in the list. List can be walk forward only.

- Tail queue - in contrast to previous type is possible to insert element at the end of a list. Due this is implementation more difficult than list implementation.
- Circular queue - the most complex type of list. in addition to tail queue entries can be added before any entry in the list and support backward walking of the list.

List

List head structure type is defined by macro *LIST_HEAD(HEADNAME, TYPE)* where *HEADNAME* is name of struct and *TYPE* is type of linked structures. In contrast to Linux, this macro do not defines variable of list, however its type only. By referencing *HEADNAME* can be defined next list heads of this type. Linked structure has to contain item, its type is defined by macro *LIST_ENTRY(TYPE)*, where *TYPE* is type of these linked structure. List head has to be initialized by macro *LIST_INIT(LIST_HEAD *head)*.

Item to list can be inserted by calling macros

*LIST_INSERT_HEAD(LIST_HEAD *head, TYPE *elm, LIST_ENTRY NAME)* and

*LIST_INSERT_AFTER(LIST_ENTRY *listelm, TYPE *elm, LIST_ENTRY NAME)*

where *head* is pointer to head of the list, *listelm* is pointer to element after it has to be new element added, *elm* points to inserted element and *NAME* is name of *LIST_ENTRY* structure in linked structure type. Removal macro is defined as

*LIST_REMOVE(TYPE *elm, LIST_ENTRY NAME)*

where argument *elm* is pointer to touched element and sense of *NAME* is similar to previous functions.

There are no macros for walking a list. Cycles have to be implemented manually through the items of *LIST_HEAD* and *LIST_ENTRY* structures. This pass for next two list types too.

Tail queue

Head of tail queue is defined via macro *TAILQ_HEAD(HEADNAME, TYPE)* and item to elements linking by *TAILQ_ENTRY(TYPE)*. Initialization do *TAILQ_INIT(TAILQ_HEAD *head)*. Arguments are identical to list macros.

Insertion of elements make possible macros

*TAILQ_INSERT_HEAD(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME)*,

*TAILQ_INSERT_TAIL(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME)* and

*TAILQ_INSERT_AFTER(TAILQ_HEAD *head, TYPE *listelm, TYPE *elm, TAILQ_ENTRY NAME)*

These macros are similar to previous list type. Difference is in referencing *head* by macro *TAILQ_INSERT_HEAD* and added *TAILQ_INSERT_TAIL* inserting element at the end of a list. Removal macro is called

*TAILQ_REMOVE(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME).*

Circular queue

Similar to previous types, macro for head definition is *CIRCLEQ_HEAD(HEADNAME, TYPE)*, macro for elements linking is defined as *CIRCLEQ_ENTRY(TYPE)* and initialization macro is named *CIRCLEQ_INIT(CIRCLEQ_HEAD *head)*.

For inserting elements to circular queue are defined macros

*CIRCLEQ_INSERT_HEAD(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME)*,

*CIRCLEQ_INSERT_TAIL(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME)*,

*CIRCLEQ_INSERT_AFTER(CIRCLEQ_HEAD *head, TYPE *listelm, TYPE *elm, CIRCLEQ_ENTRY NAME)*

and new *CIRCLEQ_INSERT_BEFORE(CIRCLEQ_HEAD *head, TYPE *listelm, TYPE *elm, CIRCLEQ_ENTRY NAME)*

to add element before any existing element of the list. Macro

*CIRCLEQ_REMOVE(CIRCLEQ_HEAD *head, TYPE *elm, CIRCLEQ_ENTRY NAME)*

removes element from the list.

5.3 Locking

5.3.1 Linux mechanism

Because Linux supports symetric multiprocessoring and several processors can call some kernel functions at the same time. It could happened, that several processes want to modify some internal structure at once. So it is necessary to prevent this situation. Linux kernel architecture offers code locking of course.

Spinlock

Spinlock is a simple mechanism to protect code block against more processes entry. Any process can lock it. When other process tries to lock spinlock, will be waiting until spinlock is unlocked. Spinlock is represented by structure *spinlock_t*. Locking and unlocking is possible by the help of macros *spin_lock(lock)* and *spin_unlock(lock)*.

When can be any spinlock locked from interrupt, is needed to foreclose following situation. A spinnlock is locked and some interrupt routine tries to lock it againg. Then will be waiting without end and system will be frozen. To handle this case have to be used macros *spin_lock_irq(lock)* and *spin_unlock_irq(lock)*. These macros will disable processor interrupts before spinlock locking. Next possible macros are *spin_lock_irqsave(lock, flags)* and *spin_unlock_irqrestore(lock, flags)* storing and restoring state of interrupt disabling to and from variable *flags* of type *unsigned long*.

Next type of Linux lock is rw-lock defined by structure *rwlock_t*. Rw-lock can be locked for reading and for writing. For *rwlock_t* operations are macros *read_lock(lock)*, *read_unlock(lock)*, *write_lock(lock)* and *write_unlock(lock)* declared. Similarly to spinlock are irq save macros declared too:

read_lock_irq(lock), *write_lock_irq(lock)*,

read_unlock_irq(lock), *write_unlock_irq(lock)*,

read_lock_irqsave(lock, flags), *write_lock_irqsave(lock, flags)*,

read_unlock_irqrestore(lock, flags) and *write_unlock_irqrestore(lock, flags)*.

Mutex

This locking system uses wait queues. It makes possible to make process sleeping while is waiting. Mutex is defined by structure *mutex*:

```
struct mutex {
        atomic_t count;
        spinlock_t wait_lock;
        struct list_head wait_list;
};
```

Mutex can hold three states in item *count*:

- 1 - mutex is unlocked
- 0 - mutex is locked
- negative value - mutex is locked and some waiters exist

Declaration with initialization can be made via macro *DEFINE_MUTEX(name)*. For initialization at run time is defined *mutex_init(mutex)* macro. Then can be mutex locked and unlocked.

Locking functions are followings:

- **void fastcall mutex_lock(struct mutex *lock)** - locks the mutex or inserts the calling process to waiting queue when is locked
- **int fastcall mutex_lock_interruptible(struct mutex *lock)** - this function uses interruptible sleep and returns 0 when attempt to lock is obtained or returns -*EINTR* when process got signal
- **int fastcall mutex_trylock(struct mutex *lock)** - tries to lock mutex without waiting and returns 1 if succeed, otherwise returns 0

Mutex has to be unlocked via function *void mutex_unlock(struct mutex *lock)*. State of mutex can process get through a call *int mutex_is_locked(struct mutex *lock)* function returning 1 if mutex is locked, otherwise 0.

5.3.2 FreeBSD mechanism

FreeBSD kernel offers universal structure *mtx* providing locking. This structure and helping functions are defined in *sys/systm.h* header file. Initialization function is defined as *void mtx_init(struct mtx *m, const char *name, const char *type, int opts)*. Firts argument is the mutex. Argument *name* is used for debug and *type* for lock ordering. If *type* is set to *NULL* then *name* holds this function. Last argument *opts* must to be set to either *MTX_DEF* or *MTX_SPIN*. Variables *name* and *type* have to be valid till *void mtx_destroy(struct mtx *m)* call. Function *mtx_destroy* destroys the mutex, which can not be used anymore.

In context as Linux kernel spinlock (lock without process sleeping), where is mutex initialized with *opts = MTX_SPIN*, are macros *mtx_lock_spin(m)* and *mtx_unlock_spin(m)* used. These macros automatically disable processor interrupts. To save and restore interrupt flags have to be used macros *mtx_lock_spin_flags(m, opts)* and *mtx_unlock_spin_flags(m, opts)*.

Follows description of work with mutex initialized with option *MTX_DEF*. There are macros *mtx_lock(m)*, *mtx_unlock(m)*, *mtx_lock_flags(m, opts)* and *mtx_unlock_flags(m, opts)* used. To try lock mutex is possible with help of macro *mtx_trylock(m)*. It returns 0 on failure, otherwise non-zero value. Similar function with interrupt flags saving is defined as *mtx_trylock_flags(m, opts)*.

FreeBSD kernel defines several helpful macros:

- **mtx_initialized(m)** - if mutex is initialized, then returns non-zero value
- **mtx_owned(m)** - if current thread is owner of locked mutex, then returns non-zero value
- **mtx_recursed(m)** - if given mutex is presently recursed, then returns non-zero value
- **mtx_name(m)** - returns name of mutex defined by initialization

5.4 Integer types and atomic operations

Linux types

Numeric types of Linux kernel are defined in header file *linux/types.h*. The type *size_t* is generally used for memory size representing. Effect of this type is independence from used architecture. Types often used as memory offset are *off_t* and *loff_t*. The second is longer than the first. On i386 architecture is *off_t* defined as *long* and *loff_t* as long long. On others platforms can be size of these types different, but in present are definitions equal.

Linux kernel defines some types representing unsigned variations of basic C language data types. These types are platform dependent of course: *u_char*, *u_short*, *u_int*, *u_long*, *unchar*, *ushort*, *uint* and *ulong*. First four are named in context of BSD unix tree compatibility and other four in accordance to UNIX System V.

Common signed integer types independent on architecture are followings:

- **int8_t**

- **int16_t**
- **int32_t**
- **int64_t**

and unsigned variations respecting both unix trees are:

- **u_int8_t**, *uint8_t*
- **u_int16_t**, *uint16_t*
- **u_int32_t**, *uint32_t*
- **u_int64_t**, *uint64_t*

Linux atomic operations

In multiprocess environment with current access of more processes or threads to any variable is required to make this access safe. For atomic operations would be using of lock mechanism in code unnecessarily too time-consuming and complicated. Solution is in implementation these operations so that will not be divided by process scheduler. Linux kernel offers special type and functions in header file *asm/atomic.h.* Name of this type is *atomic_t.* To assign initial value to *atomic_t* variable is possible through the use of macro *ATOMIC_INIT(i)* where *i* is value what has to be assigned. Possible operations under *atomic_t* type are in following list:

- macro *atomic_read(v)* - offers value of requested *atomic_t* variable
- macro *atomic_set(v,i)* - sets value *i* to *atomic_t* variable
- function *void atomic_add(int i, atomic_t *v)* - adds value of *i* to *atomic_t* variable
- function *void atomic_sub(int i, atomic_t *v)* - subtracts value of *i* from *atomic_t* variable
- function *int atomic_sub_and_test(int i, atomic_t *v)* - as *atomic_sub*, but returns true if new value of *v* is zero
- function *void atomic_inc(atomic_t *v)* - increments *atomic_t* variable
- function *void atomic_dec(atomic_t *v)* - decrements *atomic_t* variable
- function *int atomic_dec_and_test(atomic_t *v)* - decrements *v* and returns true if new value of *v* is zero
- function *int atomic_inc_and_test(atomic_t *v)* - increments *v* and returns true if new value of *v* is zero
- function *int atomic_add_negative(int i, atomic_t *v)* - as *atomic_add*, but returns true if new value of *v* is negative

- function *int atomic_add_return(int i, atomic_t *v)* - as *atomic_add*, but returns new value of *v*
- function *int atomic_sub_return(int i, atomic_t *v)* - as *atomic_sub*, but returns new value of *v*
- macro *atomic_cmpxchg(v, old, new)* - compare and exchange - if *v* is equal to *old* then assign *new* to *v*; returns initial value of *v*
- macro *atomic_xchg(v, new)* - exchange values of *v* and *new*; returns new value of *v*
- macro *atomic_add_unless(v, a, u)* - if *v* is equal to value of *u* then returns false; otherwise adds *a* to *v* and returns true
- macro *atomic_inc_not_zero(v)* - if *v* is equal to 0 then returns false; otherwise increments *v* and returns true
- macro *atomic_inc_return(v)* - as *atomic_inc*, but returns new value of *v*
- macro *atomic_dec_return(v)* - as *atomic_dec*, but returns new value of *v*

How you see, there is lots of defined atomic operations. But is not true, that all of these macros and functions are safe for current access of more processes. If it is needed, all functions returning *int* value and last five macros require explicit lock.

FreeBSD types

In file *sys/types.h* of FreeBSD kernel headers is data type *size_t* defined too. The type *off_t*, known from Linux kernel, is hardly defined as signed 64bit number. Type *loff_t* is not here available. But FreeBSD offers unsigned alternate of *off_t* called *uoff_t*. Native unsigned variations of common C types are *u_char*, *u_short*, *u_int* and *u_long*. Moreover are defined types *ushort* and *uint* for UNIX System V compatibility.

Tightly long number variables independent on used architecture are followings:

- **int8_t**, **uint8_t**
- **int16_t**, **uint16_t**
- **int32_t**, **uint32_t**
- **int64_t**, **uint64_t**

These variables are identical to types of Linux kernel. UNIX system V alternatives *u_int8_t*, *u_int16_t*, *u_int32_t* and *u_int64_t* are in FreeBSD defined too, but are deprecated.

FreeBSD atomic operations

FreeBSD has no special data type for atomic subsystem. There are defined functions and macros in *machine/atomic.h* header file. Atomic operations work with standard C types and previously described tightly long number types. For all architectures are defined operations handling pseudotypes represented in name of operations:

- **int** - int type of C language
- **long** - long type of C language
- **ptr** - variable of pointer length
- **32** - int32_t type
- **64** - int64_t type

Several architectures can support all or any operations for pseudotypes:

- **char** - char type of C language
- **short** - short type of C language
- **8** - int8_t type
- **16** - int16_t type

Now follows list of possible operations. For example is used pseudotype *int*. Variations containing *_acq* and *_rel* will be described later:

- *void atomic_add_int(int *p, int v)* - adds *v* to **p*
- *void atomic_add_acq_int(int *p, int v)*
- *void atomic_add_rel_int(int *p, int v)*
- *void atomic_clear_int(int *p, int v)* - clears bits of **p*, where in mask *v* is 1
- *void atomic_clear_acq_int(int *p, int v)*
- *void atomic_clear_rel_int(int *p, int v)*
- *int atomic_cmpset_int(int *dst, int old, int new)* - if **dst* is equal to *old* then sets **dst* to *new* and returns 1 else returns 0
- *int atomic_cmpset_acq_int(int *dst, int old, int new)*
- *int atomic_cmpset_rel_int(int *dst, int old, int new)*
- *int atomic_load_acq_int(int *p)* - returns value of **p*
- *int atomic_readandclear_int(int *p)* - returns value of **p* and clears it
- *void atomic_set_int(int *p, int v)* - sets bits of **p* where in mask *v* is 1
- *void atomic_set_acq_int(int *p, int v)*
- *void atomic_set_rel_int(int *p, int v)*
- *void atomic_subtract_int(int *p, int v)* - subtracts *v* from **p*
- *void atomic_subtract_acq_int(int *p, int v)*

- *void atomic_subtract_rel_int(int *p, int v)*
- *void atomic_store_rel_int(int *p, int v)* - sets **p* to value of *v*

The *_acq* in name of operation means memory barrier use. In this case compiler will not optimize the operation. Also processor will not reorder data accesses. Operations with *_rel* are safe from SMP data access problems. Operations having no equivalent with *_acq* or *_rel* are save in every case.

5.5 Errors

Linux kernel defines error constants in header file *linux/errno.h.* Interesting of them are error constants used in RedirFS framework:

- **EINVAL** - invalid argument
- **ENOMEM** - not enought of memory
- **ENOENT** - file or directory not found
- **ENAMETOOLONG** - file name is too long
- **EEXIST** - file exists
- **ENOTDIR** - item is not a directory

Error constants are often returned by functions instead of standard value if some error occure. To handle errors offers *linux/err.h* header file some suitable macros and functions. First macro is *MAX_ERRNO*, what tells what is maximum value of error constants. Now is possible to determine what value is error and what value is not. In used Linux kernel is *MAX_ERRNO* defined as 4095. Macro *IS_ERR_VALUE(x)* can tell whether value *x* is an error constant. If is then returns logical 1, in other case returns 0. Error constants can be stored in pointer. To retype error constant to pointer and reversely are these functions defined:

- **void *ERR_PTR(long error)** - converts error value to pointer
- **long PTR_ERR(const void *ptr)** - determines error value from pointer

Last stuff from *linux/err.h* is function *long IS_ERR(const void *ptr).* It has similar task as *IS_ERR_VALUE(x).* But in this case takes a pointer as parameter.

FreeBSD kernel contains header file *sys/errno.h* where are error constants defined. Names and values of these constants can be different from Linux. But relevant is the fact, that interesting error names are identical. FreeBSD offers macro *ELAST* consistent with maximum value of error constants.

5.6 Dynamic memory allocation

5.6.1 Linux kernel functions

Header file *linux/slab.h* of Linux kernel offers operations for dynamic memory allocation. Analog function to *malloc* known from user-space is defined as *void *kmalloc(size_t size, gfp_t flags)*. First argument is a size of requested block of memory. Then argument *flags* can contain defined flags of allocation:

- **GFP_BUFFER** - if no free memory is available then do not try to free other pages
- **GFP_ATOMIC** - if no free memory is available then do not wait; this allocation must be used in interrupt
- **GFP_KERNEL** - standard allocation of memory - tries to free some pages if needed
- **GFP_USER** - equal to *GFP_KERNEL* on the present
- **GFP_NOBUFFER** - do not try to shrink the buffer cache of kernel
- **GFP_NFS** - similarly to *GFP_KERNEL*, but recommended for lower number of the requested pages (will be faster)
- **GFP_DMA** - dedicated for allocation of DMA memory and will repeatedly try to get requested memory

To free allocated memory has to be used *void kfree(const void *)* function, where is a pointer to allocated memory as parameter required.

Sometimes is needed to allocate a very big area in memory. In this case would block allocated in virtual memory. Header file *linux/vmalloc.h* declares prototypes of functions:

- **void *vmalloc(unsigned long size)** - allocates *size* of bytes in virtual memory and returns pointer to allocated memory or *NULL* if failed
- **void vfree(void *addr)** - free given virtual memory

Cache subsystem

The same header file defines a common cache subsystem with handle type *kmem_cache_t*. Use of cache subsystem makes operations faster, because works with objects of the same size. Function

*kmem_cache_t *kmem_cache_create(const char *name, size_t size, size_t align, unsigned long flags, void (*ctor)(void *, kmem_cache_t *, unsigned long), void (*dtor)(void *, kmem_cache_t *, unsigned long))*

creates a cache. Parameter *name* is a pointer to string used for */proc* informations. It has to be valid until cache is destroyed. Next parameter *size* determines memory size of objects witch will be caching. Alignment of objects in memory can be set via *align* parameter. Options given in *flags* can be these:

- **SLAB_DEBUG_FREE** - expensive checks on objects free

- **SLAB_DEBUG_INITIAL** - call constructor (as verifier)
- **SLAB_RED_ZONE** - buffer overrun checks through the medium of so-called red zones (watched neighbouring memory)
- **SLAB_POISON** - cause catching references to uninitialized memory
- **SLAB_HWCACHE_ALIGN** - implicit hardware objects align
- **SLAB_CACHE_DMA** - use *GFP_DMA* flag for memory allocation
- **SLAB_MUST_HWCACHE_ALIGN** - hardware objects align will be forced
- **SLAB_STORE_USER** - stores last owner information (for debug)
- **SLAB_RECLAIM_ACCOUNT** - track pages allocated to indicate what is reclaimable later
- **SLAB_PANIC** - kernel panic when cache creation fails
- **SLAB_DESTROY_BY_RCU** - use RCU for page freeing
- **SLAB_MEM_SPREAD** - spread some memory over cpuset

First three flags are working when *SLAB_DEBUG_SUPPORT* is in kernel defined. Last two arguments of *kmem_cache_create* are pointers to constructor and destructor of objects witch will be stored in cache. These functions do not have to be declared and *kmem_cache_create* can be called with *NULL* in their place. Both of them take a pointer to corresponding *kmem_cache_t* handler as the first parameter. In the other parameter can constructor get following flags (destructor can't get any):

- **SLAB_CTOR_CONSTRUCTOR** - call of constructor - it makes to use the same function for constructor and destructor possible
- **SLAB_CTOR_ATOMIC** - constructor can't sleep
- **SLAB_CTOR_VERIFY** - this call of constructor is just a verify call

Objects in this cache can be created by the help of function

*void *kmem_cache_alloc(kmem_cache_t *cachep, gfp_t flags)*

taking a pointer to corresponding *kmem_cache_t* handler as the first parameter. Parameter *flags* can contain earlier described options as *kmalloc* function. Opposite function to *kmem_cache_alloc* is

*void kmem_cache_free(kmem_cache_t *cachep, void *objp)*

. Except pointer to *kmem_cache_t* gets a pointer to object has to be freed.

Last important function from the cache subsystem is *int kmem_cache_destroy(kmem_cache_t *cachep)* destroying given cache.

5.6.2 FreeBSD alternatives

Dynamic memory allocation is accessed through the medium of *sys/malloc.h* include. There is a *malloc* function defined. This function takes other parameters in compare to *malloc* defined in user-space. Head of this function is

*void *malloc(unsigned long size, struct malloc_type *type, int flags).*

The *size* parameter is a number of bytes have to be allocated. Second parameter is a pointer to structure *malloc_type*. This structure contains debug informations. To define varible of this type FreeBSD offers macro *MALLOC_DEFINE(type, shortdesc, longdesc)*, where *type* is name of variable has to be defined. Parameters *shortdesc* and *longdesc* are description strings. To use so defined *malloc_type* variable in other C modules is possible by the help of macro *MALLOC_DECLARE(type)* (*extern* modifier inside). Parameter *flags* of function *malloc* can set these options:

- **M_NOWAIT** - do not wait for free memory
- **M_WAITOK** - can wait for free memory
- **M_ZERO** - clear memory after allocation
- **M_NOVM** - don't ask virtual memory manager for pages
- **M_USE_RESERVE** - can use reserved memory for allocation (this option is deprecated)

Already unused memory can be freed through the use of function *void free(void *addr, struct malloc_type *type)*. Argument *addr* is a pointer to memory has to be freed and *type* is the *malloc_type* variable used in the course of memory allocation.

In compare to Linux kernel, FreeBSD offers memory reallocation. Function

*void *realloc(void *addr, unsigned long size, struct malloc_type *type, int flags)*

requests actual allocated memory pointer given in first parameter *addr*. Other parameters are identical to parameters of *malloc* function. Return value is a pointer to new memory block which may differ from previous pointer. If error occurred then returns *NULL*. The one variation of *void realloc* is

*void *reallocf(void *addr, unsigned long size, struct malloc_type *type, int flags).*

Function *reallocf* frees old memory when new memory of requested size can't be allocated.

Helping macros *MALLOC* and *FREE* are evident from the kernel source:

```
#define MALLOC(space, cast, size, type, flags) \
    ((space) = (cast)malloc((u_long)(size), (type), (flags)))
#define FREE(addr, type) free((addr), (type))
```

Zones

FreeBSD defines so-called zones in header file *vm/uma.h*. It can be considered by alternative to Linux kernel cache subsystem. Collected type is usually defined as structure. It is important to place at the beginning of this structure two pointers to this type. These pointers are used internally for objects collecting. Example follows:

```
struct example {
        struct example *pointer1;
        struct example *pointer2;
        // own items of collected structure
};
```

Handler type of zones is named *uma_zone_t*. Similar to *kmem_cache_create* known from Linux kernel is function

*uma_zone_t uma_zcreate(char *name, int size, uma_ctor ctor, uma_dtor dtor, uma_init uminit, uma_fini fini, int align, u_int16_t flags).*

Parameter *name* is a name of zone used for debug informations. The *size* determines size of collected structures. The *align* parameter tells objects alignment fit in memory. Rather *align* is a determinative mask of possible starting addresses. For example: to ensure object being aligned to four-address blocks, the *align* has to be 3. Alignment fit templates are:

- **UMA_ALIGN_PTR** (sizeof(void *) - 1)
- **UMA_ALIGN_LONG** (sizeof(long) - 1)
- **UMA_ALIGN_INT** (sizeof(int) - 1)
- **UMA_ALIGN_SHORT** (sizeof(short) - 1)
- **UMA_ALIGN_CHAR** (sizeof(char) - 1)
- **UMA_ALIGN_CACHE** (16 - 1)

Interesting options given in *flags* can be followings:

- **UMA_ZONE_ZINIT** - initialization with zeroizing
- **UMA_ZONE_STATIC** - static size of zone
- **UMA_ZONE_OFFPAGE** - forces the slab structure allocation off of the real memory
- **UMA_ZONE_MTXCLASS** - define a new lock class
- **UMA_ZONE_HASH** - use a hash table
- **UMA_ZONE_SECONDARY** - defines secondary zone
- **UMA_ZONE_REFCNT** - allocates reference counts in slabs

Still undescribed parameters of *uma_zcreate* are functions of types:

*int (*uma_ctor)(void *mem, int size, void *arg, int flags)*

*void (*uma_dtor)(void *mem, int size, void *arg)*

*int (*uma_init)(void *mem, int size, int flags)*

*void (*uma_fini)(void *mem, int size)*

All of these types of functions take a pointer to allocated (freed) memory of object in argument *mem* and size of this memory block in argument *size*. The type *uma_ctor* is a constructor called after object memory allocation and *uma_dtor* is a destructor called before object memory release. New in compare to Linux cache subsystem are initializer *uma_init* and discard function *uma_fini*. Functions *uma_ctor* and *uma_init* return error value on failure, otherwise return 0. Sense of *arg* and *flags* arguments will be described later. It is not needed to declare these functions and is possible to use *NULL* instead of them.

New object can be allocated in zone *zone* by the help of functions

*void *uma_zalloc(uma_zone_t zone, int flags)*

or *void *uma_zalloc_arg(uma_zone_t zone, void *arg, int flags).*

Address of new allocated object will be returned. Options given in *flags* are identical to options accepted in *malloc* function. It makes possible to return *NULL* if flag *M_WAITOK* is not set. These flags will be passed to *uma_ctor* and *uma_init* functions. Argument *arg* of *uma_zalloc_arg* will be passed to *uma_ctor* function. The *uma_zalloc* uses *NULL* instead of *arg*. Initializer is called when object is cached and constructor after memory allocation.

To release object from zone is possible via functions

*void uma_zfree(uma_zone_t zone, void *item)*

or *void uma_zfree_arg(uma_zone_t zone, void *item, void *arg).*

The *uma_zfree_arg* will pass *arg* and *uma_zfree* will pass *NULL* to *uma_dtor* function. Destructor is called before memory release and discard function is called before remove object from cache.

5.7 File name lookup

Current version of RedirFS framework developed for Linux use this fragment of code to look file name up:

path_lookup(path_name, LOOKUP_FOLLOW, &nd)

Varible *path_name* is a full path of looked directory and *nd* is a *nameidata* structure.

Returned value is non-zero value if requested file can not be found. As the start point of operation replacement is *dentry* used. This object is stored in the *nameidata* structure.

How it is already described, to look file name up can be *namei* function on FreeBSD used. Result of this function is *vnode* object stored in *nameidata*. Note that *namei* function needs to obtain initialized *nameidata* structure. So FreeBSD variation is this:

```
NDINIT(&nd,LOOKUP,FOLLOW|WANTPARENT,UIO_SYSSPACE,path_name,curthread)
namei(&nd)
```

Return value is similar to *path_lookup*. Start point of replacement is *vnode* object in this case.

On both systems *nameidata* structure has to be released when is not necessary. Therebefore has to be reference count of start point increased. Otherwise could happen that this object will be released from memory. Later can be reference count of needless start point decreased.

Release of *nameidata* on Linux:

```
path_release(&nd)
```

Alternative on FreeBSD:

```
NDFREE(&nd,0)
```

Start point increase on Linux:

```
path->p_dentry = dget(nd.dentry)
```

Alternative on FreeBSD:

```
path->p_vnode = nd.ni_vp
vref(path->p_vnode)
```

Start point decrease on Linux:

```
dput(path->p_dentry)
```

Alternative on FreeBSD:

```
vrele(path->p_vnode)
```

5.8 Replacement of VFS object operations

5.8.1 RedirFS solution on Linux review

The RedirFS framework captures file system access by the help of VFS object replacement. Injured objects are *vnode*, *dentry* and *file*. Possible RedirFS operations reflect operations of these enumerated objects. This makes RedirFS dependent on defined VFS objects and their operations.

Linux kernel uses VFS objects caches. RedirFS has to replace operations of already cached objects and newly created objects. Operations of existing *dentry* objects are replaced through the dentry cache walking. At the same time are replaced operations of *inode* objects referenced by cached *dentry* objects. Corresponding *file* objects can not be directly discovered from *dentry* or *inode* objects, because more *file* objects can reference one inode and the *inode* structure do not contain a list of these *file* objects. Files are listed in *super_block* structures. From *dentry* object can be detected corresponding *super_block* object. There can be found requested files. Current version of RedirFS does not walk this list and offers newly opened files to serve only.

Operations of lately created *inode* and *dentry* objects are replaced in calls of existing *inode* operations, that are for new object creation used. File operations are replaced differently. Structure *inode* contains a pointer to *file_operations* structure and during *file* object initialization is this pointer to *file* structure copied. To replace operations of newly created *file* object is needed to set *file_operations* of every *inode* object. This can be made along with *inode* operations replacement.

Together with *inode* and *dentry* operations replacement are RedirFS objects *rinode* and *rdentry* created. RedirFS object *rfile* is created when any file is opened. These RedirFS objects are released when corresponding *inode* or *dentry* object leaves cache.

5.8.2 FreeBSD possibilities

The VFS of FreeBSD operating system cointains objects *vnode* and *file*. These objects have a vector of operations. In this time is not operations replacement implemented. Lets try to find a method how to replace these operations of all instances. From file name lookup is a *vnode* object available.

Vnode operations replacement

FreeBSD kernel has a name cache, what is implemented as zones. There are stored existing vnodes with corresponding file names. The name cache is composed of two zones named *cache_zone_small* and *cache_zone_large*. The first is dedicated to short file names and the other is dedicated to long file names. Common structure for both zones is defined:

```
struct namecache {
        LIST_ENTRY(namecache) nc_hash; /* hash chain */
        LIST_ENTRY(namecache) nc_src; /* source vnode list */
        TAILQ_ENTRY(namecache) nc_dst; /* destination vnode list */
        struct vnode *nc_dvp; /* vnode of parent of name */
        struct vnode *nc_vp; /* vnode the name refers to */
        u_char nc_flag; /* flag bits */
        u_char nc_nlen; /* length of name */
        char nc_name[0]; /* segment name */
};
```

This structure is not a complete data type, what is cached in name cache zones. Last item of *namecache* structure is file name. Note in *namecache* is not any byte reserved for the file name. To name cache zones are stored memory blocks longer then *namecache* structure. But the *namecache* structure is placed at the beginning of these blocks. Lenghts of cached blocks are defined by macros *CACHE_ZONE_SMALL* and *CACHE_ZONE_LARGE*.

```
#define CACHE_ZONE_SMALL (sizeof(struct namecache) + CACHE_PATH_CUTOFF)
#define CACHE_ZONE_LARGE (sizeof(struct namecache) + NAME_MAX)
```

Used FreeBSD operating system version defines *CACHE_PATH_CUTOFF* as 32 and *NAME_MAX* as 255. Being of these two name cache zones makes caching faster and more effective. FreeBSD defines macros common for both zones. Macro *cache_alloc(len)* allocates and returns a name cache item. Used zone depends on required file name lenght given by *len* parameter. To release an item is possible by the help of macro *cache_free(ncp)*. The *ncp* argument is a pointer to item has to be freed.

Internal structure of zones is very complicated and to walk cache via this structure is not a good idea. Zones are not designed for this stuff. There is possible to use procedure similar to walking in file name lookup used. Structure *vnode* contains several items used for name caching:

- **LIST_HEAD(, namecache) v_cache_src** - cache entries from this *vnode*
- **TAILQ_HEAD(, namecache) v_cache_dst** - cache entries to this *vnode*
- **struct vnode *v_dd** - parent directory *vnode*

By the help of this can RedirFS framework to walk existing vnodes and to replace vnode operations.

Operations of latterly created vnodes can be replaced during parent vnode operation call. This policy is used in RedirFS Linux version for newly created *inode* objects. The alternative to this could be using of callback functions of name cache zones. The second possibility presents higher complications risk.

RedirFS object named *rvnode* corresponding to *vnode* object has to be created closely before operations replacement. VFS objects on FreeBSD have no operation called when VFS object is released. So *rvnode* has to be released by the other way. Here callback functions of name cache zone has to be used.

File operations replacement

Open files are linked in a list defined by *LIST_HEAD(filelist, file).* To replace operations of existing files is possible to walk this list and to compare referenced *vnode* object with required *vnode.*

Object *vnode* does not contain an operations vector for files in contrast to *inode* object defined in Linux kernel. Latterly created *file* object can be detected in *vop_open* operation of parent directory vnode call. Then *vop_close* operation call can be used for detection of *file* object removing. At the same time can be created and removed corresponding RedirFS object named *rfile.*

5.9 Testing

Currently is ported full logic of RedirFS objects independent on operating system. It is possible to use all public functions of RedirFS, but no operations to replace are defined.

5.9.1 RedirFS kernel module

Compilation

The RedirFS framework can be compiled from source code directory via *make* command. Result of compilation is *redirfs.ko* file. This is FreeBSD kernel module.

Loading and unloading module

To load compiled RedirFS module to kernel offers command:

```
kldload ./redirfs.ko
```

By the help of command *kldstat* is possible to check if the module is loaded. User can get more informations from command *dmesg.* In case of *kldload* error is detailed statement there. Command

```
kldunload ./redirfs.ko
```

removes the module from memory. If is this module used by another module then it will stay in memory.

5.9.2 Testing filter

RedirFS filters are implemented as kernel modules too. Created testing filter is named *testflt.* Similarly to RedirFS kernel module can be compiled by force of command *make.* Compiled kernel module is named *testflt.ko.*

This module depends on RedirFS kernel module *redirfs.ko.* It means that the *testflt.ko* can be load only when *redirfs.ko* is already loaded into kernel. This can be made via call of command *kldload ./testflt.ko.* The opposite command is *kldunload ./testflt.ko.*

Test objectives

When is the *testflt.ko* loaded into kernel, following actions are performed:

- filter registration
- setting of */bin* path
- setting of */bin/sh* path
- setting of */binx* path
- call of function registering operations
- filter activation
- setting of */usr/bin* path

During module unloading are performed these operations:

- filter deactivation
- filter unregistration

Test output

The *testflt.ko* module calls RedirFS functions and for each of them types description and evaluation of returned value. After module loading and unloading the *dmesg* command types:

```
testflt: load module
testflt: register filter
testflt: ok
testflt: set path "/bin"
testflt: ok
testflt: set path "/bin/sh"
testflt: error -20
testflt: set path "/binx"
testflt: error -2
testflt: set operations
testflt: ok
testflt: activate filter
testflt: ok
testflt: set path "/usr/bin"
testflt: ok
testflt: unload module
testflt: deactivate filter
testflt: ok
testflt: unregister filter
testflt: ok
```

Test evaluation

- filter registration ... **ok**
- setting of */bin* path ... **ok**
- setting of */bin/sh* path ... **error -20**
- setting of */binx* path ... **error -2**
- call of function registering operations ... **ok**
- filter activation ... **ok**
- setting of */usr/bin* path ... **ok**
- filter deactivation ... **ok**
- filter unregistration ... **ok**

Major part of calls was performed without error. Problematic are several calls of path setting. The *testflt.ko* module uses these paths:

- **/bin** - existing directory
- **/bin/sh** - existing regular file
- **/binx** - nonexistent file system item
- **/usr/bin** - existing nested directory

In case of regular file is error *-20* returned. It corresponds to *RFS_ERR_NOTDIR*. Error of nonexistent file system item is value *-2* which is represented by *RFS_ERR_NOENT*. Result of test is positive.

Chapter 6

Epilogue

This thesis describes VFS of source and destination operating systems of port. There are described VFS objects containing operations which can be replaced. Then description of RedirFS framework follows. The RedirFS project is very interesting from security view. This framework defines operations reflecting operations of *inode*, *dentry* and *file* objects used in VFS of Linux. FreeBSD has *vnode* and *file* objects. It is not possible to keep defined RedirFS operations fully independent on used operating system. There can be found any equivalences, but structures of operation arguments contain operating system specific data types.

Currently are ported common RedirFS objects usable on both operating systems. Temporarily is used dynamic linked list implementation of Linux, because logics of Linux and FreeBSD lists are very different. Now can be implemented RedirFS objects new on FreeBSD port.

Personal gains are skills in Linux and FreeBSD kernel development. There was a lot of exploration of both kernel source codes. The kernel development is far harder than development in user-space. Every fault can cause operating system crash and that is why debugging is not so easy. Development of Linux kernel is faster and documentation used to be not up to date. FreeBSD is better documented and source code contains more comments. It was not easy to find any FreeBSD equivalents to Linux conceptions, because are very differently named. Next complication was missing documentation of new RedirFS framework conception.

Bibliography

[1] Robert Love. *Linux Kernel Development.* Sams, 2004. ISBN 0-672-32512-8.

[2] Neil Matthew, Richard Stones. *Linux programujeme profesionálně.* Computer press, 2001. ISBN 80-7226-532-6.

[3] Marshall Kirk McKusick, George V. Neville-Neil. *Design and Implementation of the FreeBSD Operating System.* Addison-Wesley, 2005. ISBN 0-201-70245-2.

[4] František Hrbata. Callback Framework for VSF layer [Thesis]. FIT VUT Brno.

[5] WWW sites. The Linux kernel.
http://www.win.tue.nl/~aeb/linux/lk/lk.html.

[6] WWW sites. Source code and documentation of Linux Kernel.
http://www.kernel.org/pub/linux/kernel/v2.6/.

[7] WWW sites. Seriál Porovnání systémů Linux a FreeBSD.
http://www.root.cz/serialy/porovnani-systemu-linux-a-freebsd/.

[8] WWW sites. FreeBSD manual pages.
http://www.freebsd.org/docs/man.html.