VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

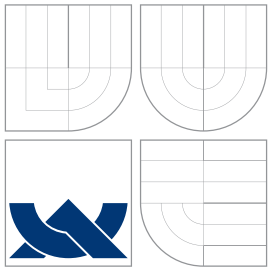
EXTENDING REDIRFS TO USERSPACE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

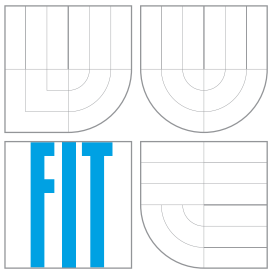
AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ PÍRKO

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍŘENÍ REDIRFS DO UŽIVATELSKÉHO REŽIMU

EXTENDING REDIRFS TO USERSPACE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ PÍRKO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ KAŠPÁREK

BRNO 2007

Zadání

1. Prostudujte tvorbu LKM a možnosti komunikace mezi jaderným a uživatelským prostorem v operačním systému GNU/Linux. Zaměřte se na znaková zařízení a Netlink sokety. Dále nastudujte možnosti rozšíření implicitní kontroly přístupu k souborovým systémům(např. kontrola per proces nebo kontrola AV softwarem).

2. Prostudujte Redirfs Framework a jeho rozhraní.

3. Navrhněte komunikační protokol mezi jaderným a uživatelským prostorem, pomocí kterého bude možné rozšířit kontrolu přístupu k souborovým systémům.

4. Implementujte systém rozšířené kontroly přístupu k souborovým systémům. Tento systém se bude skládat z LKM, který bude využívat Redirfs Framework. Dále aplikace, pomocí které bude možno definovat pravidla rozšířené kontroly. Aplikace a LKM budou komunikovat pomocí vámi navrženého protokolu.

5. Zvažte možnosti rozšíření vámi navrženého systému a Redirfs Frameworku.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Účelem této práce je určit správný způsob jak vytvořit rozšíření RedirFS, který pracuje jako modul Linuxového jádra, do uživatelského režimu a naimplementovat ho. Je zde popsán model uživatelského režimu a režimu jádra, který je použitý v Linuxu a jak spolu tyto dva režimy mohou vzájemně komunikovat. Je zde popsáno několik komunikačních mechanizmů s popisem použití. Pro všechny tyto mechanizmy jsou implementovány testy propustnosti a latence. Výsledky měření jsou předloženy a vhodný mechanizmus je vybrán. Druhá část práce je zaměřena na *redirctl* a *urfs*. První řešení představuje ovládací nástroj pro RedirFS, druhé implementaci uživatelských filtrů.

Klíčová slova

RedirFS, jádro, režim jádra, uživatelský režim, Linux, systémové volání, znakové zařízení, ioctl, Netlink, relay, relayfs, sysfs, procfs, urfs, redirctl, liburfs

Abstract

The purpose of this thesis is to consider the rigth way how to make a user space extension of RedirFS, which works as a Linux kernel module and implement it. There is described a model of user and kernel spaces used in Linux and how this two spaces can communicate with each other. There are several communication mechanisms described with a description of use. Bandwidth and latency tests for all these mechanisms are implemented. Measurement results are presented and the suitable mechanism is chosen. The second part of the thesis is focused on *redirctl* and *urfs*. The first solution represents the RedirFS control tool, the second is the implementation of user space filters.

Keywords

RedirFS, kernel, kernel space, user space, Linux, system call, syscall, character device, chardev, ioctl, Netlink, relay, relayfs, sysfs, procfs, urfs, redirctl, liburfs

Citace

Jiří Pírko: Extending RedirFS to userspace, diplomová práce, Brno, FIT VUT v Brně, 2007

Extending RedirFS to userspace

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Tomáše Kašpárka. Další informace mi poskytl Ing. František Hrbata. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Pírko
May 20, 2007

Contents

Chapter 1

Prologue

I chose this thesis because I'm interested in the Linux operating system for quite a long time. As I was working with it, it brought me a lot of fun but also a lot of stress. It even caused that I work as an embedded Linux programmer nowadays. I learned how to do many things and how to write applications, but the Linux kernel was something that I didn't know much about. This thesis gives me an opportunity to learn something that interests me.

The *RedirFS* framework is pretty usable for redirecting VFS calls at this moment. It has well defined API for making filters. Several filters based on it were already made. There is no other way to interact with this framework than from the kernel space by its API. But if there is a need to use it in the real life, a need of a user space extension is obvious. A user needs an opportunity to control filters, to set their parameters and so on. Also it could be useful to develop a filter in the user space berofe its usage in the kernel space. A debugging process will be better and a damage caused by a mistake should not endanger the whole system. There could be even filters naturally running in the user space.

The first idea was to make a user space library with its kernel module opposite to implement the whole extension. But finally, it was decided to split the solution in two. Therefore the *redirctl* control tool and the *liburfs* user filter's library arose. These are two independent solutions and each has its kernel opposite. The redirctl opposite was implanted in the RedirFS module and the kernel part of liburfs is the *urfs* module.

Both mentioned solutions need to make a communication between the user space and the kernel space. There are few communication mechanisms in the Linux kernel. For each solution, it is necessary to pick the one that fits its needs. All available mechanisms will be described. Also bandwidth and latency tests of each will be implemented and measured. The final choice will be also affected by portability issues.

Chapter 2 provides a basic idea of the user space and the kernel space. Differences of these spaces are described as well as the reason for the space division. Also the way of a system call is briefly mentioned.

Chapter 3 describes the inter-space communication methods available in the Linux kernel. For each method, the original purpose and idea is noticed, implementations of user space and kernel parts are described and advantages and disadvantages are mentioned.

Chapter 4 presents tests of all communication methods. The implementation issues of the tests, both the kernel and the user space sides, are described. Measured results are presented and discussed.

Chapter 5 choses the best method for each solution taking into account the results of tests and portability issues.

Chapter 6 describes the implementation of the redirctl control solution for RedirFS. First, there is showed how the RedirFS sources are changed to handle this solution. Then the description of the user space application is given.

Chapter 7 presents an idea of RedirFS user filters. Implementations of the kernel module and of the user library are described as well as the demo applications which use the library.

Chapter 2

Kernel Space and User Space

The Linux kernel a is central part of the operating system. It provides basic services for all parts of the system. It's responsible of a processors management, a memory management, I/O devices, an interrupt handling etc. It is bridge between applications and hardware and other applications. Applications cannot directly access devices. They have only a piece of memory and they do not know much about the outer world. They can communicate with the outer world only via the kernel.

Generally, the kernel space means a space where the kernel resides and the user space means a space where user applications reside. These two spaces are logically independent on each other. That means that the kernel stays within its space and user applications too. Applications cannot access the kernel space. In case of the kernel, previous proposition was not fully true. The kernel can access the user space via mechanism described later. Each application has its virtual memory which is usually much bigger than the system memory. That gives application the feeling that it's on the computer alone. Note that memory addresses used in one application make no sense in another.

The division of space is not only the division of memory. The kernel can call instructions which are not accessible by user applications (usually a call of this type of instruction causes an exception). These instructions are marked as *privileged*. So the kernel space also means the space where privileged instructions can be executed. In this context, the kernel space means processor's *system mode* and the user space means processor's *user mode* . Privileged instructions can be executed in the system mode only so they are accessible to the kernel only. User applications run in the user mode of the processor. Switching between these two modes is usually done by already mentioned exception, occuring during a call of privileged instruction in the user space or a timer. Both causes an interrupt which brings the processor into the system mode and lets the kernel to handle it. Back transition from the system mode to the user mode can be done by calling a special instruction.

Applications running in the user space can communicate with the rest of the world via the kernel only. This is done by a communication mechanism called *system calls*. These calls are not typically used directly by a user, but they are enveloped by functions in a user library (C library). When a user application calls one of these functions, a system call is invoked, the kernel handles it (in the kernel space) and returns a value and the control back to the application. In this regard (view of a user or a programmer) the kernel is *passive*. Kernel services are as though sub-programs of an application. An Application uses the kernel to do its job whenever it wants to. Note that an application itself (without system calls to the kernel) cannot do anything meaningful. It can compute something or do something on its assigned memory but it cannot show results or write them somewhere

or do something on connected devices. All these activities should be done by calling system calls.

Consider an example. The C library function `printf()`. It's called by a user application residing in the user space to write some text on a console. It performs some actions with the text and given variables and then it calls the function `write()`. This function does a system call by which the control is passed to the kernel. The kernel handles the system call by writing the text on the console. Now the kernel can give the processor back to the application or it can also do something itself or give the processor to another application (this is decided by a scheduler). Anyhow, the control is passed back to the application which wanted to write some text in the finite time. By giving the processor back, the application runs in the user space again.

The division of system space into these two spaces is really effective for many reasons. Especially for security reasons. The model which was described in this chapter ensures applications to not to do anything wrong. The kernel handles and checks all system calls. An application cannot evoke a system crash because the kernel never allows it to do anything what could cause it. But sometimes, system calls are implemented badly so this could appear theoretically. This is the main reason why to prompt kernel developers to implement system calls in their modules in the safe way. Because any tiny little bug in a system call can cause a system crash and other possible consequences.

The next reason is to provide a simple unified interface for a programmer. He do not need to handle many mechanisms to access hardware in the right way, to communicate with other applications or even to handle interrupts. All this is the kernel job and a programmer only takes advantages of it. He also do not need to worry about he can cause a system crash or incur a damage to other applications or cause havoc anywhere. A programmer works on his own place where everything is his problem. The rest is a kernel problem.

Chapter 3

Inter-Space Communication

In fact, as I mentioned before, there is only one way how to communicate between the kernel space and the user space (omitting exceptions and traps) and it is the usage of *system calls*. Of course, the kernel can directly access the user space on itself because it see the whole memory and can access everywhere. But the kernel does this almost always only while handling a system call. Let's call an interaction between the kernel space and the user space the *inter-space communication*.

The Linux kernel provides a set of interfaces by which applications running in the user space can interact with it. They give applications a chance to do something with hardware devices, with the system or with another application. But realize that all these communication mechanism are implemented by system calls. They must because there is no other way for an application to do something with the kernel and consequently with the rest of the system. There are several various inter-space communication mechanisms, each uses system calls as its base. Each was developed for some particular reason, for some specific sort of use and each has its benefits and also disadvantages. Let's take a look at some inter-space communication techniques which can be found in current Linux kernel (version 2.6.18).

3.1 Pure System Call

System calls (also called *syscalls*) provide a basic eventuality to perform an inter-space communication. Any other mechanisms are based on them. They are providing three main things.

The first is a hardware abstraction. For example an application does not need to know what kind of drive it's writing to. Whether it's a local disk drive, a floppy or a network drive. The application only calls appropriate system call which is referred to a driver or other kind of kernel module and it will do dirty job instead. This abstraction is not only on the hardware layer. Imagine two kernel modules. Each implements the same subset of system calls, so the API is unified to an application. But each handles calls differently. These modules could be device drivers but also for example crypting modules, each encrypting parameters into different output data.

Secondly, system calls ensure system security and stability. When the kernel handles a system call, it can check all arguments for their sense and it can even do a permission check (some system calls can be done only by root etc.). This prevents applications from doing anything wrong to the kernel, hardware devices, other running applications and the

whole system in itself. This is very important to know for a developer who is implementing a system call. All referred parameters should be checked carefully to not to cause any damage when they are used.

Thirdly, system calls create a simple single layer between an application and the rest of the world. This provides a chance to implement a multitasking and a virtual memory.

There are about 250 system calls in Linux on x86 architecture. There are differences between architectures. On some, there are some system calls missing and there may be other specific syscalls. Each system call has assigned a *syscall number*. It is unique within the system and it stands for the reference to certain system call. It stands for a syscall identificator. So when an application calls a system call, it's not done by a name but by a syscall number. Each syscall number is assigned to one and only system call. One Syscall number cannot be assigned to more system calls during the time of the kernel development process. So if during the system development a system call losts its sense and needs to be removed, a hole in the syscall number sequence appears. When application tries to call a system call by this syscall number then `sys_ni_syscall()` is executed by default instead of the original system call. It does nothing except returning -ENOSYS.

The list of all registered system calls in Linux kernel is stored in `sys_call_table`. This is architecture dependent and it is most often defined in `arch/<arch>/kernel/entry*.S` (on x86_64 arch the table is located in `include/asm-x86_64/unistd.h`). Each architecture has a slightly different set of system calls.

So system calls are accessed via application function calls. They can accept one or more arguments as an input. They should produce some output as a return value. They may (and in most cases they do) also cause one or more side effects. That is a change of system's state as a whole. But there are also syscalls that have no side effect. One of them is `getpid()` syscall.

```
amslinkage long sys_getpid(void){
  return(current->tgid);
}
```

It's common for system calls to return zero value in case of success, sometimes also positive value (like number of processed bytes and so on). If passed parameters are bad or system call itself was not processed in the right way, some negative number is returned indicating the cause of error. The example above is a kind of syscall that acts in the different way. It returns current PID in any way.

When a user space application wants to perform a system call it must somehow tell the kernel to do it. An application cannot execute a kernel code directly. If this could happen, the system stability and security would be threatened. When an application which is running in the user mode needs to go into the system mode where the kernel can handle a system call, it must give the control to the kernel. On some architectures, this is done by the special instruction - the software interrupt. On other architectures, it's done by calling any privileged instruction. Both incur an exception and the processor will switch into the system mode. Then the kernel can execute a system call handler. Of course, a syscall number must be delivered to the handler (this is also architecture dependent and it's usually done by a register). System call parameters are passed to the kernel by some registers too, as well as a return value. Then the kernel looks into the system call table and executes an appropriate function which handles the system call.

When a programmer writes an application, he doesn't need to know much about syscall numbers or even system calls at all. For example the C Library (*libc*) has functions which

implements processing of system calls. The library has a similar table of syscall numbers as the kernel (defined in `asm/unistd.h`) so it can call system calls accordingly. These functions are for example `read()`, `write()`, `fork()`, etc. But a programmer, if he wants to, can call a system call directly by a number. This is provided by a set of functions (macros to be exact) named `syscallX()`, where "X" is a number between zero and six and it corresponds to the number of parameters passed to the system call. Or eventually, the function `syscall()` can be used. It's only needed to include `sys/syscall.h` and pass a syscall number as the first parameter. The rest of parameters are used as corresponding syscall parameters and their count is also dependent on the definition of syscall.

It's easy to implement a new system call. Only a new syscall number needs to be added into the kernel table. Then a handling kernel function should be written. The function must be static part of kernel so it cannot be defined in any kernel module. Next thing to do is to register the function to the chosen syscall number. This is also architecture depended because different architectures have different interfaces. On x86_64 architecture, this is done in the same file where the syscall table resides by calling following macro:

```
__SYSCALL(syscall_NR, function)
```

Then the system call can be executed by `syscall()` function from the user space or it can be also added to the libc library. Just make sure to pass the same syscall number as defined in the kernel table.

Adding a syscall is simple but it's not recomended to do so. The main reason is, that system calls are added and removed by core kernel developers. Therefore adding a system call can cause a nonstandard interface which cannot be used worldwide (note that if anyone can add a syscall number then the big mess will arise). In most cases, there is not unavoidable to add a new system call. There are many other ways to solve the problem. It's also the only type of mechnism which needs to be hard wired inside the kernel binary, so there is no chance to make this using modules.

3.2 Proc Filesystem

The proc filesystem (*procfs*) is not a regular filesystem with data placed on some hard drive or something. This is a virtual filesystem provided by the Linux kernel. It's providing information about system, drivers, modules etc. It's a logically and hierarchically organized tree of files. It's also possible to tell something to the kernel through the procfs, to set some module parameters, to get some device statistics and so on. The purpose of use is quite wide these days.

Many applications use the procfs (e.g. **route** to grab a routing table from the kernel). It's standard for all Linux distributions to use it and in most cases it's mounted to `/proc`. Originally, the "proc" word was used because the filesystem's purpose was mainly to provide information about processes (applications currently running on the system). But it was embraced by programmers to satisfy other communication needs. Nowadays it's used for many things from getting information about connected devices to setting up some kernel parameters. The procfs directory tree structure is not strict and somewhere you can find things that should be on the different place and so on.

Each file in the `/proc` directory structure is bound with specific kernel functions. A file in this context is called *entry*. The content of all files is dynamically generated when the `read()` system call is used. It directs an execution to the mentioned kernel function. The write process operates in the same flavour. Files are always relevant to some module

(or hard wired piece of kernel code). If a module is added into the kernel then the files automatically appear and if the module is removed, they disappear immediately. Therefore, as was described, the content of filesystem is generated "on the fly" and there is no problem to add some new files. The most of files in `/proc` directory are read-only.

3.2.1 Reading proc entry

When a read-only file needs to be created in the `/proc` directory, the first thing to do is to implement a function that will make an output. The output is passed to the read function whenever any application calls the `read()` syscall for the file. If an application reads from this file, the kernel allocates a page in the memory. The function will write output data to the memory page which will be passed to the user space as a result of the `read()` call. Prototype of a procfs read function should look like this.

```
int read_proc(char *page, char **start, off_t offset,
              int count, int *eof, void *data);
```

Here comes a description of the parameters. The *page* pointer refers to the kernel allocated memory page mentioned before. The generated output should be written here. The parameter *start* points to a memory inside the allocated page where data were written. It's used only if the output is bigger than one page or it's `NULL` otherwise. The parameters *offset* and *count* has the same meaning as for the *read* syscall. The *eof* parameter indicates that there are no more data to return. The *data* parameter is driver-specific pointer which can be used for whatever needed. The function returns a number of bytes written to the page as like as the `read()` syscall does. Note that if `read()` is called from the user space and there is more bytes than one page size requested, the read function will be called sequentially on data chunks. Then *start* and *offset* are important to process.

This interface is not really suitable for reading data of bigger size than the size of a page. It can be done by using the mentioned *start* parameter but its usage is a bit tricky and developers can do a lot of mistakes while they are doing this. For this purpose a *seq_file* interface was added into the kernel. This interface is providing a set of functions handling large virtual files in the kernel. That makes working with bigger data easier thus there is no transfer speed improvement.

3.2.2 Writing proc entry

Similar to read handlers, there are write functions used in the `/proc` filesystem too. Their purpose is to implement read-write files or write-only files.

```
int write_proc(struct file *filp, const char __user *buff,
               unsigned long len, void *data);
```

The *filp* parameter is a pointer to structure of opened file. This should be ignored when the function is not using a file to write and has some other effect. The *buff* is a pointer to data passed to the function. The pointer is a user space pointer so there is needed to use the `copy_from_user()` function instead of `memcpy()` here. The *len* parameter stands for the length of data passed by *buff*. The parameter *data* is a private pointer and has the same usage as in the read function.

3.2.3 Entry manipulation

To create and use a new file in procfs the following actions need to be proceeded. Firstly, a file needs to be created during the module init. This is done by `create_proc_entry()`. Then the handling functions must be set for the write and read requests as described earlier. Also it's important to remove the entry when the module unloads by calling the function `remove_proc_entry()`. It's even possible to create directories and symlinks by `proc_mkdir()` and `proc_symlink()`. See next example for easier understanding.

```
struct proc_dir_entry *entry;
entry = create_proc_entry(‘‘example’’, 0644, &proc_root);
// bind the handling read function
entry->read_proc = example_read;
// bind the write funtion
entry->write_proc = example_write;
...
// at last remove the entry
remove_proc_entry(‘‘example’’, &proc_root);
```

The parameter `proc_root` tells that the root of procfs, which is common to be `/proc`, should be us as a parent directory.

3.2.4 Conclusion

Although the implemention of an inter-space communication via proc filesystem is easy it's not recommended to do so because it is now becoming obsolete in some ways. For changing information, there is more suitable and suggested to use a *sysfs* which will be described later in this text. Of course, the proc filesystem is nowadays indivisible part of the Linux system and it's likely to be used in the future for a long time.

3.3 Sys Filesystem

The sys filesystem called *sysfs* is also a virtual filesytem like the procfs. It comes with 2.6 Linux kernel release hand in hand with the unified *Device Model* which is featured there. They provide a device tree which is hierarchically structured in many views. Devices connected to buses, modules in use, block devices and many other information. It is meaned to replace the procfs in this regard. Mainly due to its very chaotic and wildly evolved hierarchy. The structure of the sysfs directory tree is well defined and developers should respect it.

As I mentioned before, the sysfs also acts like a medium for Linux Device Model. It provides a general abstraction describing the structure of the system. At this time it's used for many tasks:

- *Power Management and system shutdown* - it ensures that devices will shutdown in a proper way even in case they are dependent one on another.
- *Communications with user space* - thanks to the sysfs it's possible to interfere with devices in this model easily by the user space processes.

- *Hotpluggable devices* - it provides an opportunity to connect and disconnect devices on the running system with notifying a user about it in the smooth way.
- *Device classes* - this can help user applications to be informed what kind of devices are available.
- *Object lifecycles* - this is dealing with changing objects during the time in case they are made for devices that are hotplugged, suspended and so on.

3.3.1 Architecture

The main power of the sysfs rests in its object model. The basic term is *kobject*. Originally it stands for a reference counting only. But its purpose was enhanced. Now tasks handled by a kobject structure include representation in the sysfs, gluing data structures of a device and handling hotplug events. Each object consists of other objects or attributes. It's a tree where the kobject on a higher level contains other kobjects as its subclasses. The object tree in the memory can be exported to the filesystem easily. Then kobjects are directories and attributes are files. Kobjects are also collected by *kset* in a set of objects. It acts like a container.

Another important term is *ktype*. It is represented by the kobj_type structure and it defines a kobject (or kobjects - can be used for many kobjects) behaviour. The structure contains a pointer to the zero-reference handling function, an attribute set and a pointer to *sysfs_ops*. Pointers to the read and write system calls handling functions are stored here.

Another good feature of the sysfs is a unified reference counting system. That means every code that needs to access kobject increments reference counter first, then it does some work and finally it decrements the counter. The functions `kobject_get()` and `kobject_put()` are handling this.

3.3.2 Objects manipulation

The creation of kobjects can be done by `kobject_init()` and `kobject_add()`. This can be either done via the single function `kobject_register()`. It's required to zero the kobject structure and set the parent kobject and the kset before calling these functions. Let's see example.

```
struct kobject *kobj;
// allocate memory for object
kobj = (struct kobject *) malloc(sizeof(struct kobject));
// zero allocated memory
memset(kobj, 0, sizeof (struct kobject));
// set kset, ktype and parent object
kobj->kset = kset;
kobj->kset = ktype;
kobj->parent = parent;
// init and add kobject in one function
kobject_register(kobj);
// must set the name of object
kobject_set_name(kobj, ‘‘my new object’’);
```

Existing kobject can be removed by `kobject_del()` or `kobject_unregister()` (which combines the first function with `kobject_put()`). This is the way how to work with the directory tree of sysfs. Alone it isn't meaningful. Files (attributes) are needed to appear in directories represented by kobjects.

If there is a demand to add some entries into the sysfs without any linkage to existing ones, a new directory in the root of sysfs has to be made. This procedure is called "subsystem creation" and it's done by the function `subsystem_register()` which prototype is in `linux/kobject.h`. It should be used like this:

```
decl_subsys(my, NULL, NULL);
subsystem_register(my_subsys);
```

After calling this, the pointer `&my_subsys.kset.kobj` can be used for other kobjects as a parent. Also a new directory that belongs to the new subsytem appears in the sysfs root directory. In the previous example it's named `my/`.

3.3.3 Attributes manipulation

There is a default set of attributes in each created kobject. It depends on specified ktype what they are. All kobjets of the same ktype have the same set of attributes. In the ktype structure, there is a member called *`default_attrs`* which is an array of attribute structures.

```
struct attribute{
  const char *name;
  struct module *owner;
  mode_t mode;
};
```

In the attribute structure the member *`name`* is a name of attribute (and also file). The member *`owner`* is a pointer to the current module if there is any. This usually has to be set to `THIS_MODULE` or `NULL`. The last member *`mode`* specifies permissions that the file will have in the sysfs.

In most cases, default attributes defined in the ktype structure satisfy all related kobjects needs. This uniform approach is also good for the code cleanness. But sometimes when an instance of kobject is some kind special, it is needed to specify another attribute as well. This can be done by creating an instance of attribute structure filling it with needed data. Then the new attribute is added to the kobject by calling the function `sysfs_create_file()`. The `sysfs_ops` structure registered to the ktype of this kobject will be used to handle reads and writes.

The function `sysfs_create_link()` allows to make symbolic links to attributes between two different kobjects. There are also opposite functions to these two mentioned above. They are `sysfs_remove_file()` which will delete a file entry from a directory and `sysfs_remove_link()` which does the same with a symbolic link. Prototypes of the functions follow. The first two functions return zero on success and negative error code otherwise.

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
int sysfs_create_link(struct kobject *kobj, struct kobject *target,
                      const char *name);
void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr);
void sysfs_remove_link(struct kobject *kobj, const char *name);
```

3.3.4 File operations

There is another important member of the ktype structure besides *default_attrs*. It's *sysfs_opt* and it points to the structure of the same name. It contains two members. A pointer to the *show* function and a pointer to the *store* function.

```
struct sysfs_ops{
  ssize_t (*show) (struct kobject *kobject, struct attribute *attribute,
                   char *buffer);
  ssize_t (*store) (struct kobject *kobject, struct attribute *attribute,
                    const char *buffer, size_t size);
};
```

When a file in a sysfs directory is read, the `show()` function is called with the parameters *kobject* and *attribute* that is going to be read. It fills *buffer* with generated data. The buffer is allocated previously to a size of page. The function returns a number of bytes written to the buffer or a negative value in case of error.

The `store` function is called in case of a write call. Function reads *size* bytes from *buffer* and does something with them. The buffer is again previously allocated page so *size* can be no longer then `PAGE_SIZE`. A return value is a number of bytes read or a negative number to indicate an occurrence of an error.

3.3.5 Conclusion

The sysfs is now very popular. It takes place of the obsolete and chaotic procfs and it also provides a replace for `ioctl()` calls of pseudo char devices. This makes driver's API safer from possible bugs that may occur when using older mechanisms, but developers must follow certain conventions. Each sysfs attribute should export only one value and this value should be text based and easy to use in C. This also gives an opportunity to look at and change attributes directly from the command line. The main goal of this is to eliminate big and weirdly structured files used in the procfs. Another convention is to keep the directory tree well arranged and to take care of a clean hirearchy. The sysfs is an effective tool and if we want it to be in the future it's necessary to take care of it during writing kernel parts that use it.

3.4 Netlink socket

The *Netlink protocol* is another inter-space communication method. It was developed for a specific sort of use. During an integration of network layer into the Linux kernel the need of another communicating mechanism rised. Therefore the Netlink protocol came. It is implemented by the *Netlink socket*.

The Netlink socket provides high-speed full-duplex link between the kernel and a user space application. For a user space application it acts like an ordinary socket and the work with it is the same. The Netlink socket is created by calling `socket()` with the `PF_NETLINK` domain.

```
netlink_socket = socket(PF_NETLINK, socket_type, netlink_family);
```

Netlink uses the `AF_NETLINK` address family. For this sort of use of `socket()` there are socket types `SOCK_RAW` and `SOCK_DGRAM` equivalent so it doesn't matter which one is used. There is also a couple of protocol types (*`Netlink_family`*) which take place within the Netlink socket. They are specifying the kernel module or the Netlink group to communicate with. Let's look at the most significant ones:

- *NETLINK_ROUTE* - used for both IPv4 and IPv6 for gathering information from routing tables, modifying them, setting interface addresses, traffic classes, queuing disciplines and so on
- *NETLINK_FIREWALL* - gets the packets passed through Netfilter to the user space, manipulates them and re-injects them back
- *NETLINK_IP6_FW* - the same for IPv6
- *NETLINK_NETFILTER* - used for communicating with Netfilter by **iptables**
- *NETLINK_INET_DIAG* - used for monitoring INET sockets
- *NETLINK_ARPD* - used for managing the network neighbours table

Each protocol has its own syntax and specifics. Generally, there are packets sent through the socket. They have an unified look according to the used protocol type. But they are are wrapped by a Netlink socket header which will be described later in the text. It's good that Netlink uses a BSD-like socket style thus there is easy for developers to use them because they are used to it.

Originally, the Netlink socket was developed and used only for network-related needs although nowadays there are several more sorts of use. Main advantage against the others communication mechanisms is that the Netlink socket is asynchronous. That means that in case that an application wants to receive some information from the kernel it must poll with a syscall or read a file from the procfs in the loop and so on. This may be expensive because polling should be quite frequent if we need near real-time manner. Comparing to this, the Netlink socket only blocks on the read and if the kernel sends data through the socket, the application gets data immediately. It can be also used in combination with `poll()` mechanism so the application handles events on the Netlink socket just like on a network socket, a serial line or any other file descriptor based communication. This makes the Netlink socket something special among all other mechanisms (as well as character devices which will be mentioned later in this text).

Another feature of the Netlink socket which makes it better than any other mechanism is an opportunity to handle a multicast communication. That's its uniqueness. One application can send data to a group address. Then any other application can listen to the group address. This is pretty usable for distributing kernel events system wide. For each protocol type, there can be defined up to 32 multicast groups. Each group is identified by one bit in a 32-bit integer value.

3.4.1 User space API

The Netlink socket uses the same API like other standard sockets, from a user space application point of view. So `socket()` for open, `close()` for close, `sendmsg()` for sending data and `recvmsg()` for receiving data can be used. Because socket handle is a file descriptor

too, generally the standard `read()` and `write()` functions can be also used as like as any other file descriptor manipulating functions.

After creating a new Netlink socket by calling `socket()` with appropriate parameters, a local address must be assigned. This is done by calling `bind()` as we know it from TCP/IP sockets. The socket address structure looks like this.

```
struct sockaddr_nl{
  sa_family_t nl_family;
  unsigned short nl_pad;
  __u32 nl_pid;
  __u32 nl_groups;
};
```

Before the `bind()` call is performed, the variable `netlinkaddr` of type struct sockaddr_nl should be defined. Then *nl_family* needs to be set to `AF_NETLINK` and *nl_pad* to zero. The *nl_groups* member is the 32-bit multicast integer mentioned above. If there is an interest to get some multicast messages from some group, then the bit related to desired group must be set. Just set *nl_groups* to zero to receive unicast messages only.

The last structure member *nl_pid* stands for the PID of the current process using socket. So generally, it can be simply set to the PID of the process which can be get by `getpid()`. But more complicated situation might appear when the process contains more threads and at least two of them want to open the same socket. Then they will have the same *nl_pid*. That's the situation when several conflicts may appear so it must be prevented. This can be solved using the PID with an ID of the thread. That should be get by `pthread_self()` when using the standard POSIX thread library.

```
netlinkaddr.nl_pid = (pthread_self() << 16) | getpid();
```

An example how `bind()` should be called after setting `netlinkaddr` structure follows. Note that the *sc* parameter is a file decriptor previously returned by `socket()`.

```
bind(sc, (struct sockaddr *) &netlinkaddr, sizeof(struct sockaddr_nl));
```

Now the socket is initialized, the address is assigned and also the multicast groups are registered if there are any. It is time to send some data through it. Sending messages is a tricky process in the opposite of sending data to the TCP/IP socket. Here, data are sent in the form of messages via the `semdmsg()` function like they do in the UDP communication.

Message headers must be assembled before sending a message. The Netlink socket requires to send its Netlink header before any useful data (in all protocols). Look how the Netlink header structure looks like.

```
struct nlmsghdr{
  __u32 nlmsg_len;
  __u16 nlmsg_type;
  __u16 nlmsg_flags;
  __u32 nlmsg_seq;
  __u32 nlmsg_pid;
};
```

The *nlmsg_len* member is a length of payload including a length of this header. The NETLINK_SPACE() macro mentioned later is usually used to set this item. The *nlmsg_type* member is used to indicate a message type. It is set to NLMSG_ERROR in a case of error message. If a message is multipart then NLMSG_DONE should be sent in the last header to tell a receiver that it is the last. Most usually it's set to zero. Each Netlink protocol specifies its own types of messages. The *nlmsg_flags* indicates the manner of a message receiver. It indicates if there is need to send an acknowledgement message back or echo the message and so on. And it's also usually set to zero. The structure member nlmsg_seq is used to track the message. At last nlmsg_pid should be set to the same value as nl_pid in the sockadd_nl structure passed to bind()

A Netlink message consists of a stream of bytes which can be multiple Netlink headers with associated payloads. Payloads are put behind each header. This one or more headers and payloads are included into another structure which is used by the sendmsg() and recvmsg() functions.

```
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

The mentioned structure is the msghdr structure and it's wrapping a whole Netlink message.

```
struct msghdr{
  void *msg_name;
  socklen_t msg_namelen;
  struct iovec *msg_iov;
  size_t msg_iovlen;
  void *msg_control;
  socklen_t msg_controllen;
  int msg_flags;
};
```

In case of using this structure for sending Netlink messages, only the first four structure members should be filled. The *msg_name* and *msg_namelen* members carry the destination address and its length. The member *msg_iov* points to a Netlink header and *msg_iovlen* is always set to 1. It has been decsribed how to make a message header. Now will be described how to make the destination address.

Every message sent by the Netlink socket has a destination address set in the msghdr structure. The address is represented again by the sockaddr_nl structure described above. If the message should go to the kernel then *nl_pid* and *nl_groups* should be set to zero. When the message is a unicast message then *nl_pid* is set in the same fashion as the local address of the process and *nl_groups* is set to zero. If there is demanded to send a multicast message then *nl_groups* should have the relevant bits set (they may be more than one if the message should be sent to more groups).

Finally the packet is ready to be sent by sendmsg(). You may wonder there are no payload data but only headers. The magic is in storing data to the Netlink nlmsghdr structure by macros defined in netlink.h. This macros are also useful for the computation of a length of the message. Of course, it's necessary to allocate a memory for the Netlink structure considering a length of the payload that will be added by the macro. The payload is placed right beyond the Netlink header. The three most significant macros prototypes

are showed below. The first one computes a length of the whole header with payload, the second gives a length of a payload and the last returns a pointer where data should be stored.

```
int NLMSG_SPACE(size_t len);
int NLMSG_LENGTH(size_t len);
void *NLMSG_DATA(struct nlmsghdr *nlh);
```

Consider a Netlink header `nlh` of the type struct nlmsghdr and a destination address `daddr` of the type struct nlmsghdr, both filled well. Also consider a message header `msg` of the type struct msghdr. Now put them together. Then `msg` can be passed to `sendmsg()`. An example of assembling and sending a message is showed below. Note that there is a temporary variable `iov` used to set the Netlink header (that contains payload).

```
struct iovec iov;
msg.msg_name = (void *) &daddr;
msg.msg_namelen = sizeof(daddr);
iov.iov_base = (void *) nlh;
iov.iov_len = nlh->nlmsg_len;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
sendmsg(fd, &msg, 0);
```

To receive a message from the Netlink socket by the `recvmsg()` function, the similar things to the message sending, as showed in the example above, should be done. The difference is that after the final `recvmsg()` call an address of a source is stored in `daddr` variable and a Netlink header with a payload in `nlh`. Of course, it's necessary to know the maximal length of data that might come from the socket to allocate an appropriate chunk of memory for the Netlink header and the payload. This maximal length should be set to `iov.iov_len`. The real received Nelink header length will be stored in `nlh->nlmsg_len`. A pointer to the payload can be get using the `NLMSG_DATA` macro.

3.4.2 Kernel space API

The user space side of the Netlink socket has been described. Now take a look at the kernel Netlink socket API. It's pretty different to the user space one. It is implemented in `net/netlink/af_netlink.c` and it provides the Netlink API for any part of the kernel. All protocols like `NETLINK_ROUTE` or `NETLINK_ARPD` are implemented using this API. If the Netlink socket is wanted to be used for a different purpose, the first thing to do is to create a protocol type. This is done by adding a line containing something like following to the file `linux/netlink.h` (considering the last existing record number is 16).

```
#define NETLINK_MYOWN 17
```

As the Netlink socket is initialized by the `socket()` call in the user space, there is also a similar function in the kernel space. But it's not so general as `socket()`. It's closer related to Netlink. Here is its prototype:

```
struct sock *netlink_kernel_create(int unit, unsigned int groups,
                                   void (*input)(struct sock *sk, int len),
                                   struct module *module);
```

The parameter *unit* stands for the protocol type (NETLINK_ROUTE for example). The *groups* parameter is a number of multicast group we want to use and the *module* parameter is a pointer to the module structure of the kernel module who called this function. The *input* parameter is a pointer to the function which handles incoming messages. It could be `NULL` in case the kernel doesn't need to receive messages. The last parameter is usually set to `THIS_MODULE`.

Once the Netlink socket is created, all incoming messages for this socket will be handled by calling the registered `input()` function. The function should use its *sk* to get messages from the queue by calling the function `skb_dequeue(&sk->received_queue)`. This call returns a pointer to the structure sk_buff. The structure has the members *data* and *len*. The Netlink header of the type struct nlmsghdr can be get from them. Then the Netlink header macros mentioned above can be used to get a payload and to perform other operations. It's recommended to do the work of `input()` handler in separate kernel threads to avoid blocking when the processing of messages takes more time. The pointer to sb_buff structure (named skb in following example) should be freed after all (when it's not used for the sending of an answer).

```
skb_pull(skb, skb->len);
kfree_skb(skb);
```

As like as in the user space, there is a possibility to send unicast and multicast messages in kernel via the Netlink API. There are two functions which send messages depending on it's unicast or multicast.

```
int netlink_unicast(struct sock *ssk, struct sk_buff *skb,
                    __u32 pid, int nonblock);
int netlink_broadcast(struct sock *ssk, struct sk_buff *skb,
                      __u32 pid, __u32 group, gfp_t allocation);
```

The parameter *ssk* is a pointer to the sock structure previously returned by the function `netlink_kernel_create()`. The *pid* parameter is an identifier of a receiver and *group* is the 32-bit integer used to specify multicast groups. The *nonblock* flag is used to signalize if the send should be non-blocking and it is so if it's set to `MSG_NONBLOCK`. If there is a need to block then set it to zero. The *allocation* parameter stands for the kernel memory allocation type and it's necessary because socket buffers for multicast messages may be cloned in the memory. Possible values are `GFP_ATOMIC`, `GFP_USER` or `GFP_KERNEL` and they are the same as for the `kmalloc()` function.

I didn't mentioned the *skb* parameter in the previous paragraph because it needs some extra care. It is a buffer of an output message and it needs to be previously prepared. The Netlink header with a payload is stored there in `skb->data`. An incoming sk_buff can be used for an answer, if the payload needed to be send is no longer then the incoming payload. Otherwise a new sk_buf must be made by the `alloc_skb()` and `skb_put()` functions like this:

```
skb = alloc\_skb(NLMSG\_SPACE(payloadSize), GFP\_ATOMIC);
nlh = (struct nlmsghdr *) skb\_put(skb, NLMSG\_SPACE(payloadSize));
```

The source and destination addresses should be set before calling one of the sending functions. This is done in the different fashion comparing to the user space API. There is a macro `NETLINK_CB` which has a pointer to a socket buffer as an argument and it gets an access to set up addresses. See the next example how it can be done.

```
// local addresses
NETLINK_CB(skb).groups = local_groups;
NETLINK_CB(skb).pid = 0; // zero means kernel
// destination addresses
NETLINK_CB(skb).dst_groups = dst_groups;
NETLINK_CB(skb).dst_pid = dst_pid;
```

3.4.3 Conclusion

If we want to make an inter-space communication, the Netlink socket seems to be quite good option. Unlike the system calls or the procfs there is no need to change so many things or to make the kernel much chaotic. To comunicate over the Netlink socket, there is only need to add a netlink protocol to `netlink.h` and communication in the standard socket way is ready. The main advantage is the possibility of sending multicast messages. An application using Netlink can be made as a kernel module. The only problem is similar to the syscalls' problem with syscall numbers. A new Netlink protocol number must be used and it makes our code unusable world-wide. Netlink protocol numbers must be controlled by core kernel developers too.

3.5 Character devices

Another way to do an inter-space communication is to make a new character device (also called chardev). Then it is a pseudo character device driver. I have mentioned a word "pseudo" for a reason. In fact, this mechanism uses the kernel device access interface but in our application it has nothing to do with any physical device at all. This way is very common and popular because it's easy to implement and it's quite straightforward.

Character devices in Linux were envolved to implement a communication with a simple types of devices. These are serial ports or a keyboard or any other devices operating with sequential data. These devices can be accessed by reading and writing to special files called *nodfiles*, usually located in the `/dev/` system directory (but can be anywhere else). For example for the first serial port there is usually a nodfile named `/dev/ttyS0`.

This nodfiles are not the ordinary files. They have *major* and *minor* numbers which are significant for them. By these numbers the file is connected to corresponding driver and device in the kernel. A major number stands for a driver, a minor number identifies a device within the driver. For example tty driver has the major number 4. Then the first serial port has the minor number 64 and the second has 65. A file in the `/dev/` directory can be made by calling the **mknod** command and its use is demonstrated by the following example. The "c" means a character device.

```
/bin/mknod /dev/ttyS0 c 4 64
```

Now look how to implement a custom communication using a character device. The first thing is to pick a free major number. There is a list of used major numbers in `linux/major.h`. So grab a free number and add a new line to the file. Another thing to prepare is to make a file in the `/dev/` directory as described before or eventually the nodfile can be made by the `mknod()` function in an application. It doesn't matter what minor number is chosen because it's not needed for this type of driver. Usually in this case it is set to zero.

When implementing the kernel side (the driver), the first thing is to register the char device. This should be done by calling `register_chrdev()`.

```
int register_chrdev(unsigned int major, const char *name,
                    const struct file_operations *opts);
```

On the opposite, the last thing needed to do is a chardev unregistration by calling `unregister_chrdev()`.

```
int unregister_chrdev(unsigned int major, const char *name);
```

The registration is usually done during a kernel module load or the other driver initialization and the unregistration is done when the module is removed. The *major* parameter is the device major number and *name* is a name of driver. These two values are shown in `/proc/devices` and besides that the name has no reason. If the *major* parameter is set to zero then the register function picks the last available free major number and returns it. The last parameter *opts* is in fact a list of system calls which are implemented for this device with a link to the handling function. Make an imagination while looking at the example. All variables starting with "my_" are handling functions for the desired system call (or the file operations in this context) of this driver.

```
static struct file_operations opts = {
  .owner = THIS_MODULE,
  .ioctl = my_ioctl,
  .read = my_read,
  .write = my_write,
  .open = my_open,
  .release = my_close
};
```

It's usual to implement only the `ioctl()` function for the communication purpose. The rest is not necessary to implement and in most cases there can be anything needed done only by `ioclt()`. This call was originally ment to tell tape drivers to roll back the tape or serial driver to set the speed or the handshaking type. In the pseudo chardev its used for performing any kernel driver actions. Take a look at its prototype.

```
int ioctl(struct inode* ip, struct file* filp,
          unsigned int cmd, unsigned long arg);
```

The first parameter can be avoided because it is useless in this application. The second parameter can be useful for storing some void pointer of a connection context in its *private_data* item. The *cmd* parameter is used to specify a command code that a driver should do and it is passed from the user space as as well as *arg* which is used to carry information. It can be a value directly or an memory address where valuable data are stored. The function processes the command with some effect and returns a value dependent on the command. It's good habit to return a negative value in case of failure. This value will be set as *errno* (this manner is shared among all operations).

Sometimes, it is more suitable to implement the `read()` and `write()` functions, not just only `ioclt()`. The main advantage is that the `poll()` mechanism can be used to wait in the user space until the kernel sends some data while using `read()` instead of `ioclt()`. Technically, the `poll()` mechanism can be used with `ioctl()` too but it's very uncommon.

```
ssize_t read(struct file *filp, char __user *buff,
             size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char __user *buff,
              size_t count, loff_t *offp);
```

The first parameter *filp* has the same sense as in `ioclt()`. The *buff* parameter is a pointer to data in the user space and *count* is a number of bytes that should be read from *buff* or the maximum count of bytes that may be written to *buff*. The *offp* parameter might be used by a module to store a position of read or written data.

Note that a memory address from the user space cannot be accessed directly. It's necessary to copy the specified memory area from the user space by a special function. The similar operation must be done to copy some chunk of data from a kernel memory back to the an address in the user space. These functions are `copy_from_user()` and `copy_to_user()`. They return a number of bytes that failed to copy. They behave like the ordinary `memcpy()` function and its use is similar. But they should be used instead in case we care of the security and the system stability. It's due to user space pages can be swapped and therefore not available at the time of access.

When the driver is made and loaded, the communication can be easily performed by using mentioned calls. Relevant nodfile must be previously created in the `/dev/` directory and opened by `open()`. Note that `ioclt()` command codes must be the same in a driver and in an application.

Although implementing an inter-space communication via char devices is easy, it is not that pure. Mainly due to relatively big mess among major numbers. Many developers are using major numbers on their own and that causes conflicts. Therefore it is recommended to use dynamic major numbers. It's recommended to choose the different mechanism when the needs suits better.

3.6 Relay interface

This mechanism, formerly known as *relayfs*, was developed for relaying a large amount of data from the kernel to the user space via user specified *relay channels*. This communication type can be very fast and effective but it's significant insufficiency is that it can only transfer data from the kernel space to the user space but not vice versa. Although sometimes the dupex communication is not required.

Mentioned relay channel is a relay mechanism implemented as a set of *kernel buffers*. Each belongs to one CPU and is exported to the user space as a *relay file*. The file is created in a host file system, for example the *debugfs*. A kernel code is using the relay API to write to this buffers. Then related files can be opened, read or mmaped in the standard way. Even there can be used the `poll()` mechanism to wait for data became ready. There is no protocol and it's up to a particular application what is relayed and in what form. The reason is to keep it as simple as it can be.

The kernel buffer consists of several *sub-buffers*. Messages are written to sub-buffers from the first to the last always with an attention to fill the most of sub-buffer's space. If there is no space for a whole message, it goes to another in the row. One message cannot be splitted into more sub-buffers. This is the cause of unused holes. Since the kernel knows about them, the user space can be notified about them to make a read more effecient. When the user space reads whole sub-buffer, it's freed. Meanwhile the kernel can write messages to another sub-buffer. Some function prototypes of the kernel relay API follow.

```
rchan *relay_open(const char *base_filename, struct dentry *parent,
                  size_t subbuf_size, size_t n_subbufs,
                  struct rchan_callbacks *cb);
void relay_close(struct rchan *chan);
void relay_write(struct rchan *chan, const void *data, size_t length);
```

It is easy to create a new relay channel. This is done using `relay_open()`. The *`base_filename`* parameter is a string containing a name of the file that will appear in the host filesystem. The file name will have a suffix containing the number of CPU because there the file represents the buffer of one CPU. The *`parent`* parameter is a pointer to the directory entry where files should appear. If it's `NULL` then files will be placed in a root of the host file system. The *`subbuf_size`* and *`n_subbufs`* parameters specify the size of each sub-buffer and their count.

The last parameter is a pointer to the structure which contain callback functions. If it's `NULL` then default callbacks which do void operation are used. There are two most significant functions, the rest is used for special reasons. The first function is `create_buf_file()` and it is called from `relay_open()` once for each CPU. It returns a pointer to the dentry structure and it's used for a file creation. The second function name is `remove_buf_file()`, it's called by `relay_close()` and it removes files previously created by `create_buf_file()`. These functions can be implemented by calling the debugfs API functions `debugfs_create_file()` and `debugfs_remove()`.

The `relay_write()` function should be used to write data to a channel. Its first parameter is a pointer to the chan structure previously returned by `relay_open()`. The *`data`* parameter points to data that has to be written and *`length`* stands for its size. A programmer must be careful to write the right count of bytes, because only one sub-buffer is used in one `relay_write()` call (data cannot overflow to the next sub-buffer).

The Relay interface is a fairly new kernel feature so it's not used for many things. So far there is only one use of it in the whole vanilla kernel. But its simplicity and robustness gives it a potential to be used widely in the future.

Chapter 4

Bandwidth and latency tests

There were described all possibilities how to do user space vs. kernel space communication. All were developed for some sort of use and all have their advantages as well as disadvantages. It can be interesting to compare all to each other aspecting a possible communication bandwidth and a latency of feedback. I've implemented all these mechanisms in testing applications. How it was done and what results it brought is a content of the text in this chapter.

4.1 Implementation

In the context of all inter-space mechanisms, I had to implement basically two kinds of tests. The first was a test of bandwidth in both ways, when transporting data from user space to kernel space and vice versa. It is desired to copy data between kernel buffer and user space buffer. Data size may vary up to 2MB. The purpose of the second kind of test was to take the response time between event happening in the user space and its kernel space reaction. This caught a latency of the method.

For time measuring the function `gettimeofday()` is used. It gets the system time in a precision of microseconds. Each test is repeated several times in a loop. The reason is to eliminate an overhead. Time measuring starts before the loop and ends after it. Then the total time divided by the number of loops gives the time of one iteration. This is approximately the real time of one iteration.

There is necessary to make both sides of communication for each mechanism. A user space application does the measurement and it is the active part of communication. Also the kernel side, which does the passive serving work for the application in the user space, needs to be implemented. The kernel part should be made as a kernel module and except one method it is.

Now look how the tests are implemented. There will be described hinges of each method. For a better understanding, look into sources.

4.1.1 Pure System Call

This is the only mentioned type of communication which cannot be done as a kernel module. It must be hardwired into the kernel. I placed the test source file `syscall_test.c` into the `kernel/` directory of the kernel source directory. The `Makefile` of this directory must be edited, adding `systall_test.o` in between other targets. Then the syscall function should be implemented. It's prototype follows.

```
asmlinkage long sys_test(int type, char __user *buf, size_t count);
```

This function should be registered to a syscall number. As was written in chapter 3, adding a syscall to the system is architecture dependent. In my case the architecture is `x86_64`. For this architecture the syscall table is located in `include/asm-x86_64/unistd.h`. Following two lines should be added to the end of syscall list (assuming the last syscall number is 279).

```
#define __NR_test 280
__SYSCALL(__NR_test, sys_test)
```

Then the maximum syscall count must be set to this new value (because the new call is now the last) like this:

```
#define __NR_syscall_max __NR_test
```

The new syscall is registered and now the syscall work should be implemented. I used three parameters with my syscall `sys_test()`. The first is a type of test. It specifies read, write or latency test. The second is a pointer to a user space buffer and the last is the count of bytes that should be processed. When writing from the user space, the function `copy_from_user()` is used to copy data to the local kernel buffer. When reading the `copy_to_user()` function is used to do the inverse operation. The latency test is implemented to do a void operation. Just need to know how long does it take to enter and leave the syscall.

The user space application uses the testing syscall by calling the function `syscall()` with appropriate parameters.

4.1.2 Proc Filesystem

I choose the standard procfs interface in the kernel module rather than using `seq_file`. The reason is that for a testing purpose there is no significant benefit of its use. Also the standard interface is widely used in the kernel so it was easier to take an inspiration.

There are the functions `test_read()` and `test_write()`. They work with a buffer. They are called by the procfs layer when an user space application reads or writes to a specified file.

In the module init function, there is need to make a file entry in the `/proc/` directory and also to set reading and writing functions to the made ones.

```
entry = create_proc_entry(``test'', 0644, &proc_root);
entry->read_proc = test_read;
entry->write_proc = test_write;
```

The user space application reads from the file and writes to the file like to a standard file. There is need to rewind the file using `lseek()` to begin between two calls. The latency test measures time between the start of write and the end of read.

4.1.3 Sys Filesystem

The implementation of the sysfs was more complicated then in case of previous two methods. It was caused by a file size in the sysfs, which is limited to `PAGE_SIZE`. On my architecture (amd64) it's 4KB. I resolved it by mapping the kernel buffer onto multiple sysfs files.

During the module init there is needed to register a new subsystem `sysfs_test` which will create a directory named `sysfs_test/` in the sysfs root. Then a group of attributes is created there. The first attribute is called `datasize` and it contains a size of data which can be read or written. By setting this, the handling function takes care of creating eventually destroying data files where the kernel buffer is mapped. The second attribute only exports `PAGE_SIZE`.

Another thing that's left to do when the module inits is to register a new kobject. Mentioned datafiles should be placed under it. This kobject registering causes a creation of the directory `datapages`. Functions that handles read and write requests on the datafiles are stored in a ktype of this new kobject.

The user space application needs to set the `datasize` attribute first. Then it gets the size of one page from the `pagesize` attribute to count a number of pages it needs to use. All files each related to one page are opened before the measurement. Then data transfers are done by writing and reading them. The latency test is made in the same way as for the procfs test.

4.1.4 Netlink socket

The implementation of this method was not quite simple either and it takes me the longest time of all to do it. It was caused mainly by the work with Netlink headers and packeting of data to messages. The care must be taken of a message length. A message is carried by the `sk_buff` structure and a total length of this structure including the message cannot be longer than the size of `slab`, which is 128KB. The communication is based on multi messages, so the last message is flagged by setting type `NLMSG_DONE`. This forces the opposite side to sent a reply.

The kernel side of the Netlink socket is created by `netlink_kernel_create()` during the module init. It uses the previously chosen Netlink protocol number defined by `NETLINK_TEST`. This also registers the handling function `test_rcv()` for incoming messages.

```
test_sk = netlink_kernel_create(NETLINK_TEST, 0, test_rcv, THIS_MODULE);
```

The handling function does a work of the module. The user space application is sending messages where the first byte specifies a command. It tells whether data may be written to the kernel buffer or if they may be read or if it's the latency test. It may be also the special command which sets a data length for an operations.

The user space application prepares output and input headers first. They will be used for sending and receiving messages. Then the Netlink socket is opened with the same protocol number as in the kernel module and tests are done. The latency test is done by sending only the command byte in a message and the kernel side replies with the accept byte.

4.1.5 Character devices

This mechanism implementation was much simplier than previous two. There are three file operations for read, write and ioctl calls created and registered via `register_chrdev()` in

the kernel module. The major number and the name of the character device are passed to this function too.

```
register_chrdev(CHARDEV_TEST_MAJOR_N, CHARDEV_TEST_NAME,
                &chardev_test_ops));
```

As for the syscalls, the pointer passed to the write and read operations is a user space pointer so it needs to be handled with the `copy_to_user()` and `copy_from_user()` functions.

The user space application for this method needs to create a nodfile first. It uses a temporary nodfile which is created before any operation by the `mknod()` function. Then the file is opened and actions are done on it. The nodfile is removed before the application terminates. This approach is non-standard and was chosen for testing purposes only. The latency is measured by calling `ioctl()` which the kernel module handles by doing a void operation.

4.1.6 Relay interface

This method provides a data transfer from the kernel space to the user space only so there is not reasonable to do the latency test either. A chardevice ioctl call is used to trigger a write of the requested amount of data. There is one sub-buffer used because there is no need to use more in this test implementation. When the module inits, the mentioned serving chardev is created as well as a relay is opened by `relay_open()`. The testing module creates relay file buffers via the debugfs. The debugfs must be mounted to `/debug/`

The user space application needs to open the chardevice. Also it needs to open the relay buffer file. It's placed in the debugfs root so it must be mounted previously. The chardevice nodfile is made the same way as for the chardevice test.

4.2 Results

The tests have been processed on a dedicated computer. It was AMD Athlon64 3000, 1GB RAM running the Debian Etch stable Linux distribution. When the tests were running, all services have been shut down so the processor was used only by the kernel test modules and the user space test applications. The write and read tests were done for data size from 64 bytes to 2 megabytes in multiples of two. The number of iterations was chosen for each data size to keep the test running for at least one minute (to avoid the influence of an overhead). Look at the charts where results are illustrated.

The read test results are shown in 4.1. The best performances, as was expected, have the chardev and syscall methods. Also the sysfs is not bad. The write test in 4.2 shows the similar results. Only a sysfs performance is much lower comparing with the read test. The latency test results in 4.3 shows the lowest latency for the syscall and the chardev.

On the first look there is a wonder why the Netlink results are so bad. But it has a rational explanation. Comparing to the syscalls or the character devices where every data transfer is done via a single function call, a Netlink data path is much complicated. For example when sending data to the kernel, the data must be copied from the application buffer to the Netlink header by `memcpy()` first. Then it's send via `sendmsg()` and the kernel makes the structure `sk_buff` and puts the data there. Then the kernel module copies the data to the local kernel buffer. So the data are copied three times. Also realize that data should be transferred by about something less then 128KB blocks. It's also disadvantage of the Netlink socket.

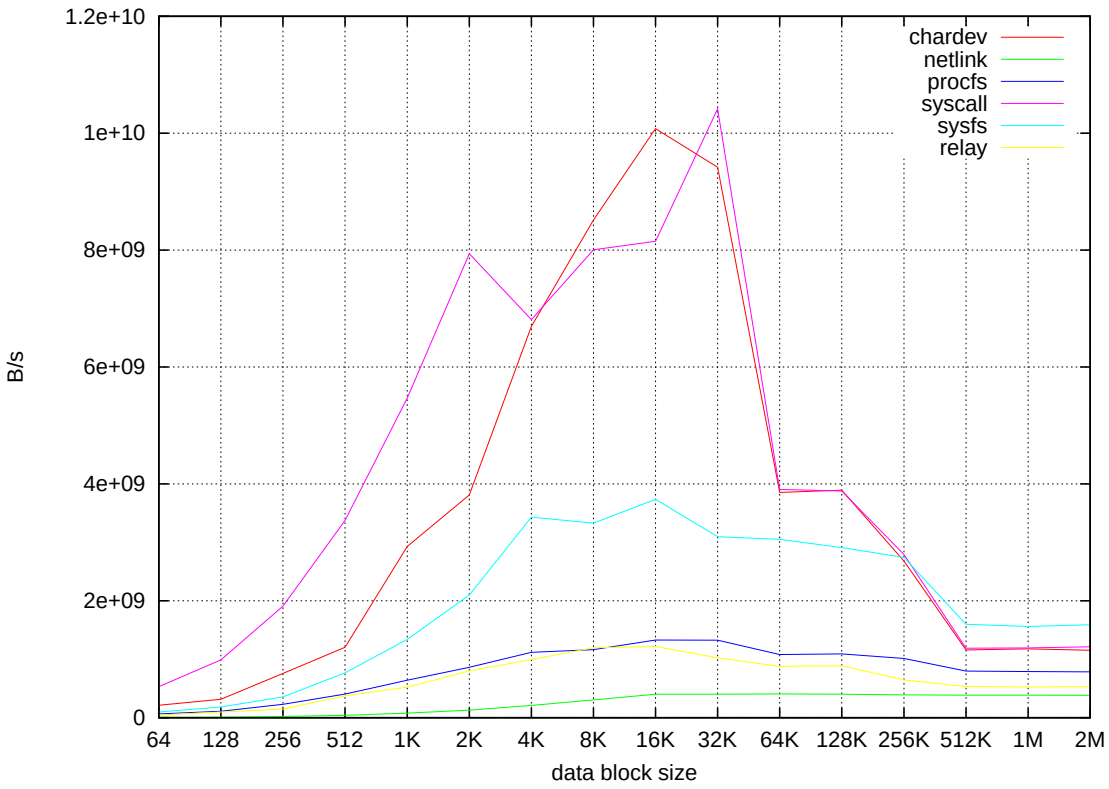


Figure 4.1: Read test

There are other mechanisms that also copy data at least two times. In a case of the sysfs and the procfs, their engine copies pages from and to the kernel space and the testing module copies this page to and from the local buffer. In a case of the relay interface, the kernel module writes data from the local buffer to the relay buffer. Then the application copies data to its buffer.

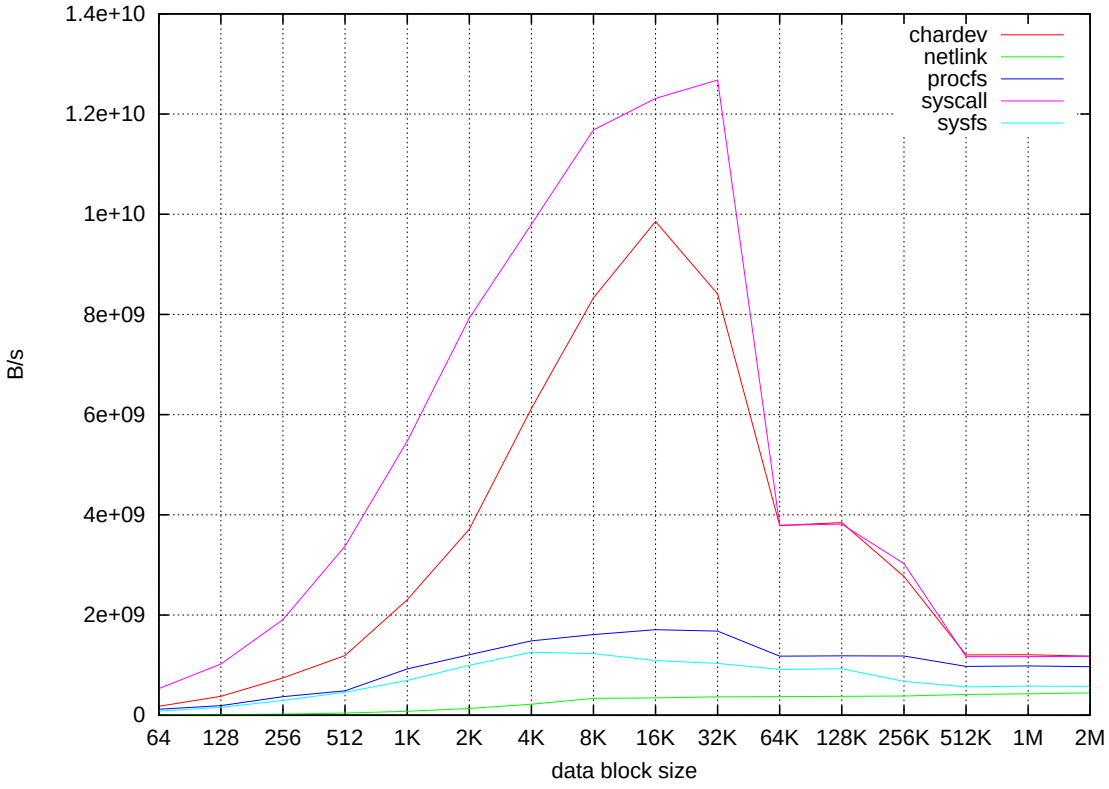


Figure 4.2: Write test

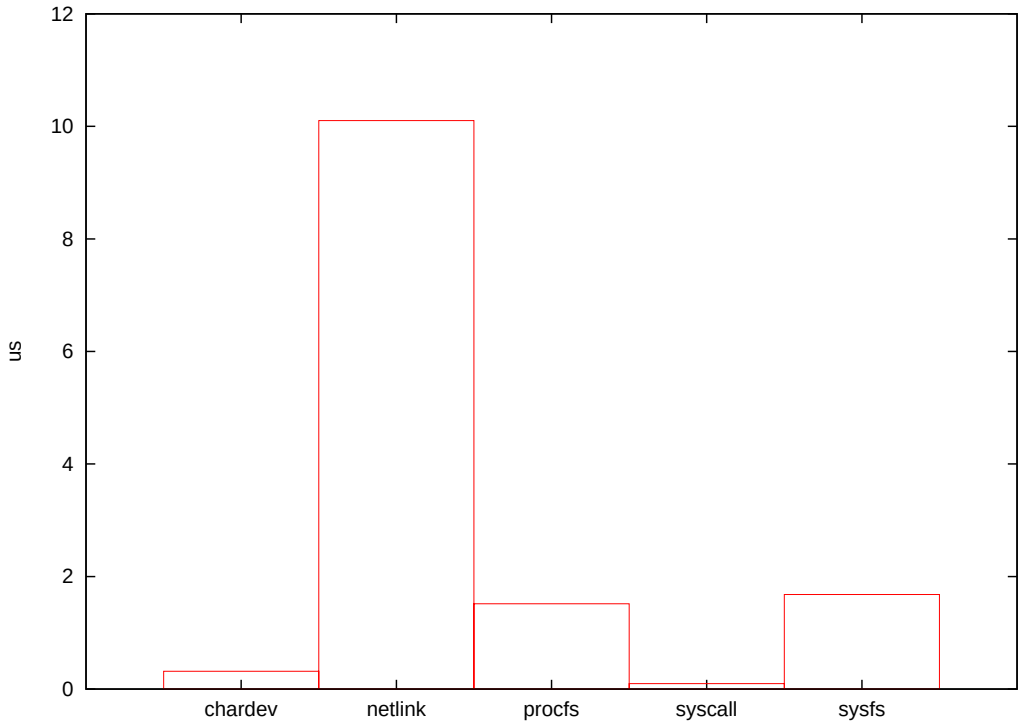


Figure 4.3: Latency test

Chapter 5

Choosing mechanism

This chapter will consult needs for a communication mechanism that will be used between a kernel module dealing with RedirFS and a user space application. First, cases of usage will be considered for this solution with having a respect to communication needs. The last thing that is also important is the portability of solution to another operating systems.

5.1 Case studies

There are generally two basic cases of use for a RedirFS user space solution. The first is an application providing the control of RedirFS filters. This will give an opportunity to activate and deactivate filters, to change a set of affected paths and to get a list of registered filters. In this case, each filter is made and registered to the RedirFS API by the kernel module, including operations handlers. This case requires a simple communication mechanism. No large data will be transferred through, so there is no need for a high performance communication. There is also no need for a low latency. This solution will be implemented as a user space application with the opposite implanted to RedirFS.

The second case consists in the opportunity to create *userspace filters*. This will provide the full RedirFS API with a possibility to register filters with operation handling functions implemented in the user space. In this case, large data can be transferred both directions so there is need for a high performance communication mechanism.

The second case can be pretty usable for writing and debugging filters. Also it can be usable for example for anti-viral scanning when a scanning application can run in the user space. It can get whole files to scan or only their paths. Other applications are file crypting and compressing. This solution will be implemented as a user space library with a kernel module as its opposite. The module will emulate user filters as standard RedirFS filters.

5.2 Portability

The user space library and the RedirFS control application will be created and the communication mechanism with the kernel part will be developed for use in Linux. Thus it will be good if it can work with none or minimal changes in other UNIX-like operating systems where RedirFS can be ported. Especially the accent is put on FreeBSD where RedirFS is being ported when working on this thesis.

As I realized, the only similar inter-space communication mechanism for Linux and FreeBSD are the character devices. So in the view of portability the winner is obvious.

5.3 Result

According to needs that have been mentioned for both main cases of use, the best option for implementation is a character device. The `ioctl()` call can be used for setting filters and the `read()` and `write()` calls can be used for data transfers. Also it is nice that the `poll()` mechanism can be used which could be useful for waiting for events. Character devices have also a good performance and a latency of calls. They are easy to add to the system and the portability point of view prefers them.

Chapter 6

RedirFS control

This solution was originally meaned to be the a part of the user space RedirFS library. This would need to extend the RedirFS API by some functions. In concrete, a function that gets a list of inserted filters, a function that gets a list of set paths and at a function to get a filter handler by the filter name. This allows all filters to access the others, to set their parameters etc. It is not suitable to provide this functionality because every filter should have its responsibility of setting itself.

Therefore was decided to separate the RedirFS control from the RedirFS user space library. The kernel part was implanted directly into the RedirFS code and the user space part was implemented as a standalone application called **redictl**. There was also need to make a mechanism that could give filters the possibility to decide if they allow to modify their settings. This is done by "the modificator callback" and it will be described later.

6.1 RedirFS control extension

A new file `redirctl.c` was added to the original RedirFS sources. Here, the chardev is registrated in `redirctl_init()` and unregistrated in `redirctl_destroy()`. These two functions are called from the RedirFS module init and exit functions. The redirctl chardevice implements only the `ioctl()` file operation. There is no need to get events from the kernel side, so an implemention of `poll()` has no reason.

There was also need to extend or slightly modificate the functionality of the original sources. Because `redirctl` needs to identify a filter by its name, the `rfs_register_filter()` function in `filter.c` was changed to make the name property unique. This file was also extended by following functions:

```
enum rfs_err rfs_set_mod_cb(rfs_filter filter,
                            enum rfs_err (*mod_cb)(union rfs_mod *));
```

This is the only function that was added to the RedirFS API for this solution. It's used to set a modificator callback for a filter. A modificator callback is a filter's function that is called everytime when `redictl` sets some parameter. Type of parameter and optionally some related data are passed to a modificator callback by the rfs_mod union. If a modificator is not set than `redirctl` is not able to set any of the filter's parameters.

```
int flt_get_by_name(rfs_filter *filter, char *name);
```

This function is used to get a filter handler by its name. It's needed by almost every `redictl` command. It walks through the list of registered filters and compares the name of each to the one passed by the parameter. Therefore the name property needs to be unique. The filter handler is set to `NULL` if there is no filter registered by that name.

```
int flt_get_all_infos(struct rfs_filter_info **filters_info, int *count);
```

It walks through the registered filters and fills the array of struct rfs_filter_info. A memory for this array and also for a name property of each filter is alocated here. It must be freed after its use. The *count* parameter is set to a number of registered filters. The function returns zero in case of success. Non-zero value is returned otherwise and in that case the allocated memory is freed.

```
enum rfs_err flt_execute_mod_cb(struct filter *flt, union rfs_mod *mod);
```

This function executes a filter's modificator callback if its set. The parameter *mod* is a pointer to the rfs_mod union in which a type of operation and relevant data are carried.

Another file that has been edited is `path.c`. There was added only one function to it:

```
int path_get_infos(rfs_filter filter, struct rfs_path_info **paths_info,
                   int *count);
```

This function gets a list of previously set paths of the filter. It behaves similar to `flt_get_all_infos()`. To walk through all paths, the `rfs_path_walk()` function is used with the defined callback `path_get_infos_cb()`. The walk is performed two times. First, a number of affected paths is get. Then a memory is allocated for an array of rfs_path_info structures. Then in the second walk the array is filled up. As a parameter to the callback function, there is defined a special structure where processing data are stored.

Now I'm getting back to file `redirctl.c`. Here, besides mentioned init and destroy function, are this functions:

```
void free_filters_info(void);
void free_paths_info(void);
```

These two functions are used in fact as opposites to the functions `flt_get_all_infos()` and `path_get_infos()`. They free previously allocated memory and they work with global variables carrying pointers to arrays and their lengths. They are called for a memory cleanup from `redirctl_ioctl()`.

```
int copy_chararr_from_user(char **chararr, void __user *ptr);
int copy_chararr_to_user(void __user *ptr, char *chararr, int len);
```

These functions are used to copy a char array from and to the user space. They are wrapping `copy_from_user()` and `copy_to_user()`. Before each char array in the memory, there is an int value that indicate its length. These functions are working with it. Both return a number of total bytes read or written.

```
int redirctl_ioctl(struct inode *ino, struct file *filp,
                   unsigned int command, unsigned long arg);
```

This is the `ioclt()` file operation handling function. It's the heard of `redirctl`. It processes several commands. A code of command is passed by the *`command`* parameter. The *`arg`* parameter is used as a user space pointer where data should be written to or read from. It's converted to the `ptr` void pointer at the very beginning. If the command needs the rfs_filter handler for its work (and almost every does), it is get by `flt_get_by_name()`. Supported commands are specified in `redirfs.h` by the redirctl_cmd enumeration and they are:

- REDIRCTL_CMD_GET_FILTERS_INFO_PREPARE performs `flt_get_all_infos()` which saves an info structure array and its length to global variables. It returns the length to tell a user space caller how much memory it needs to allocate.
- REDIRCTL_CMD_GET_FILTERS_INFO_DATA transfers info structures from the array to the user space `ptr` pointer and performs a global variables cleanup by `free_filters_info()`.
- REDIRCTL_CMD_GET_FILTER_PATHS_INFO_PREPARE is similar to the first one, this time it gets a path info structure array by `path_get_infos()`.
- REDIRCTL_CMD_GET_FILTER_PATHS_INFO_DATA transfers the path info structure array to the user space and does a cleanup calling `free_paths_info()`.
- REDIRCTL_CMD_SET_FILTER_PATH reads path info data from the `ptr` pointer, creates the rfs_mod union, sets a modificator type, fills the rfs_path_info structure with read data and passes it to `flt_execute_mod_cb()`.
- REDIRCTL_CMD_ACTIVATE_FILTER activates an incident filter in the similar way as the path setting is done.
- REDIRCTL_CMD_DEACTIVATE_FILTER is an opposite to the previous one.

Read or written data are stored sequetially in the memory with no padding (as it is usual for structures). Every char array including strings is stored in the same fashion with its length prior to the data. The work with this is handled by the `copy_chararr_from_user()` and `copy_chararr_to_user()` functions.

6.2 Redirctl user space application

The RedirFS control user space part is **redirctl** command-line application. It provides basic filter operations. Chosen operation is specified by the first command-line argument. Operations may be following ones:

- *list* - prints a list of registered filters with a name, a priority and an activity flag.
- *paths* - prints paths set to a filter with path flags (include or exclude, single or subtree).
- *setpath* - sets a path to a filter with the path passed as the third parameter, an include or exclude keyword as the fourth and a single or subtree keyword as the last.
- *activate* - activates a filter
- *deactivate* - deactivates a filter

All operations except *list* needs to know what filter they should work with. The filter is identificated by its name, which should be the second parameter. Last three operations may fail with "operation not permitted". That means that the filter did not set its modificator callback or this operation is not possible on this filter.

The structure of the application source, `redirctl.c`, is following. The application first tries to open a nodfile for redictl. The nodfile should be created manually in a system without *udev*, or it's created automatically when the RedirFS kernel module inits otherwise. Then redictl parses command-line parameters to gather the operation type. According to this, it checks the rest of operation parameters. Then they are passed to appropriate function. Each operation is implemented in a separate function. There the communication is performed via one or more `ioctl()` calls.

Chapter 7

RedirFS user space filters

Another thing that has to be done is to make an opportunity to write filters in the user space. That's the job of *urfs*. Filter is called *ufilter* in this context. The urfs extension is implemented by a kernel module and a user space library. This two parts communicate via a character device again. The accent is put on the easy portability of filters to ufilters and vice versa, although running filters in the user space has its logical limitations. So there is generally not possible to port every filter to the user space.

7.1 urfs kernel module

This module can be told as a kernel urfs stub. It's dealing with the RedirFS API on the one side and with the user space library on the other. From the RedirFS point of view there is no difference between urfs and a filter because they both use the same interface. So urfs behaves like one or more filters each mapped to one ufilter in the user space.

7.1.1 Communication protocol

The protocol used between the urfs module and the library was chosen to be simplest as it can be. Also is was necessary to make it quite effective because a large amount of data could be passed through. Therefore the concept of short messages is used. There are basically two types of them. Input messages comming from the user space and output messages passing back. Both types has its own union where the format for each command is specified. These unions are defined in the `urfs_kernel.h` header file. There are command codes defined in the urfs_cmd enumeration.

The message consists of a command identificator at the first place. Then there are values of various types and also user space pointers to some valuable data. Strings, character arrays or structures are transferred this way. They are not the direct part of the message but there are only pointers to them. This approach provides the best performance. That's because there is no need to copy data to directly to a message and at least one memory copy is saved (sometimes more, when a message is assembled of more independent data chunks). This is the reason why the word "short" was used in hyphenization with a message.

There are also two types of commands used. The first comes from the user space and the module processes it and passes a reply back. Commands that belongs to this group are following. Each command returns an error in a reply message in the same way as the RedirFS API functions. That's why the rfs_err enumeration is used.

- URFS_CMD_FILTER_REGISTER registers a new filter to RedirFS. It also creates a new ufilter instance and saves the filter's handle in here. An identificator of the new ufilter is returned. Input parameters are the same as for the `rfs_register_filter()` function.
- URFS_CMD_FILTER_UNREGISTER does the opposite to the previous command. The input parameter specifies a ufilter id.
- URFS_CMD_FILTER_SET_PATH sets a path for a ufilter calling `rfs_set_path()`.
- URFS_CMD_FILTER_ACTIVATE activates a ufilter.
- URFS_CMD_FILTER_DEACTIVATE deactivates a ufilter.
- URFS_CMD_FILTER_SET_OPERATIONS sets user space callbacks to a ufilter.
- URFS_CMD_CONN_SWITCH_CALLBACKS enables or disables a deliver of callback events to user space handling callbacks.
- URFS_CMD_OP_CALLBACK_GET_ARGS copies arguments needed for a callback to work. It uses a request id to identificate the current callback.

The second type comes up in the kernel module and the user space library processes it and produces a reply. This type is used to pass callbacks to ufilters. There is only one command in this type group.

- URFS_CMD_OP_CALLBACK is performed whenever RedirFS calls any filter callback. Through this command a callback is delivered to a user space handling callback.

7.1.2 Hierarchy and implementation

There are two header files used in the urfs kernel module. The `urfs.h` header is used to share data types and funtion prototypes inside the module. The second header file named `urfs_kernel.h` is used by the module but it's also included from the user space library. It defines the urfs interface including messages formats, command types and urfs structures used in callback arguments.

Three most important structures are defined in `urfs.h`:

- `struct ufilter` - In this structure, there is a handle for a filter stored. It also contains a ufilter id, a pointer to the conn structure under which the ufilter belongs and a list of active requests (currently processed).
- `struct conn` - Instance of this structure is created for every user space process which is using the urfs module because it's done during device open. It contains an array of subordinated ufilters. It also contains a list of output messages that have to be sent to the user space.
- `struct request` - This structure is used for the work with every outgoing command. It stores data for pending commands. Each has its unique id.

User filters

The work with ufilters is covered by `filter.c`. There are several functions which do the wrapper of the original RedirFS API functions. Besides them there are four other functions which deserve a closer look.

```
void ufilter_request_add(struct ufilter *ufilter, struct request *request);
```

This and the following function provide the interface to a ufilter request list. This should be used to add a request into the list.

```
struct request *ufilter_request_get(struct ufilter *ufilter,
                                    unsigned long long request_id,
    int del);
```

To get a request from a ufilter list, this function should be called. Wanted request is identificated by its id. If there is demand to remove the request from the list then the *del* parameter should be set to non-zero value.

```
unsigned long long ufilter_get_unique_request_id(struct ufilter *ufilter);
```

This function gets an id of a request. The next id is stored in the ufilter structure and it's computed by incrementing the current value. Therefore the request id is unique within one ufilter. Of course, there is a limit value after which the id sequence will repeat. But this value is big enought to ensure that two requests would never have the same id in the real life because the long long type is used for it.

```
enum rfs_retv ufilter_generic_cb(rfs_context context,
                                 struct rfs_args *args);
```

Each operation could have its handling callback in the user space. In the kernel module, there is need to register a kernel callback too and invoke the user space callback whenever the kernel callback appears. For this purpose, the generic callback function `ufilter_generic_cb()` is registered for every user space callback. It creates a new request, fills it with a pointer to operation's arguments and then it calls the function `process_out_cmd_op_callback()` from `chrdev.c` to perform the command. There is also need to copy data from the user space process which invoked the operation to the kernel space memory. That's important because chardev operations called from the process where ufilter runs can't access this memory. After the operation callback command is performed and the request is done, there is need to copy data back.

RedirFS modification

It was necessary to modify RedirFS slightly. The reason is that a callback function needs to know what ufilter it is called for. Therefore there was added a void pointer *private_data* to the context structure in `redir.h`. This pointer is filled right before each callback call in `rfs_precall_flts()` and `rfs_post_call_flts()`. A pointer to appropriate ufilter is stored in the filter structure of each filter as a void pointer. Then two functions are needed to extend RedifFS API:

```
enum rfs_err rfs_set_private_data(rfs_filter filter, void *private_data);
```

This function is used to set a ufilter pointer of the filter specified by the *filter* parameter. The pointer to a ufilter is passed by *private_data*.

```
enum rfs_err rfs_get_context_flt_private_data(rfs_context context,
                                              void **private_data);
```

A pointer to appropriate ufilter can be get from the *context* parameter of a callback by this function.

Connections

When a user space process opens the urfs chardevice, a new connection is created and assigned. Every operation with this device runs in the context of the connection. A connection also handles a queue of outgoing messages. Work with connections is encapsulated by `conn.c` which contains following functions:

```
struct conn *conn_create(void);
```

To create a new connection, `conn_create()` is called. It allocates a memory for the conn structure and it initializes all needed structure members, including an output message list and an array of ufilters. The function returns a pointer to the allocated structure.

```
void conn_destroy(struct conn *c);
```

This function is called on release operation of the chardevice. It does a necessary cleanup which includes an unregistration of all ufilters and freeing all previously allocated memory including a list of messages to send and a list of requests.

```
int conn_alloc_ufilter(struct conn *c, int *ufilter_id);
```

This is used to create a new ufilter. It allocates a memory for the ufilter structure and it also puts a pointer to it into the array of ufilters that belongs to a parent connection. That's the one under which the new ufilter will belong and it's referred by the `c` parameter. If there is no free place in the array, a negative value is returned. An id of created ufilter is returned otherwise. The id is in fact the index to the ufilter array of the connection.

```
void conn_free_ufilter(struct conn *c, int ufilter_id);
```

Frees the ufilter structure and marks its place in ufilter array of connection as unused.

```
struct ufilter *conn_get_ufilter(struct conn *c, int ufilter_id);
```

This function simply returns a pointer to the ufilter structure by the given *ufilter_id* parameter. If there is no such ufilter, the `NULL` value is returned.

```
void conn_msg_append(struct conn *c, struct omsg_list *omsg_list);
void conn_msg_insert(struct conn *c, struct omsg_list *omsg_list);
```

New output messages that have to be send to the user space should be added by these two functions. They do almost the same thing with one difference. The first appends a message to the end of the list and the second inserts it in the list to the beginning. After the message is added, the wake queue *waitq* is woked up. This causes the return of the user space `poll()` call if there is any. Two types of a message addiction are necessary for the right processing of commands. Commands comming from the user space have to insert replies in the list. Commands that come from the kernel side appends messages to the list's end. This approach ensures that a command called from the user space gets back its reply and not a command from other side instead.

```
int conn_msg_pending(struct conn *c);
```

Returns non-zero value if there is at least one message to send waiting, zero value otherwise.

```
struct omsg_list *conn_msg_get_next(struct conn *c);
```

Takes out the first message from the list. Returns `NULL` if the list if empty.

```
void conn_switch_callbacks(struct conn *c, int enable);
```

If *enable* is non-zero then callbacks are enabled. They are disabled in other cases. This function is called from `conn_create()` to disable callbacks. Callbacks should be enabled right before the main loop of a user space process by appropriate command.

```
int conn_enabled_callbacks(struct conn *c);
```

Checks the status of callbacks. Returns non-zero value if they are enabled.

Requests

Requests are very important part of the urfs module. They are used to perform kernel originated commands. They also remember data between output and input messages. The list of active requests is a part of the ufilter structure. The request structure also contains the list of allocated chunks of the user space memory. Members of this list are structures of the type useralloc_chunk. This structure stores a pointer to allocated memory and its size. The functions that work with requests follows.

```
struct request *request_create(unsigned long long request_id);
void request_destroy(struct request *request);
```

Every request should be created by `request_create()` before its use. The function allocates a memory for the request structure and returns a pointer to it. It also initializes a list of user space memory chunks, the request id passed by the parameter and a completion structure. To destroy the request, `request_destroy()` should be called.

```
void __user *request_useralloc(struct request *request, unsigned long size);
```

This function is called whenever it's needed to allocate some memory for a user space process. Allocation is done by the `do_mmap()` fuction. Allocated memory pointer and its size are stored in the useralloc_chunks list.

```
void request_userfree(struct request *request, void __user *ptr);
```

This is the opposite to the previous one. It seeks the *ptr* parameter in the list of allocated memory chunks and removes it. The memory of the chunk is freed by `do_munmap()`.

```
void request_userfreeall(struct request *request);
```

It's similar to the previous function, but it frees the whole list.

Character device and communication

All supporting souce codes and their functions were described. Now it's time to pay attention to the heard of the whole urfs module. It's the source file `chrdev.c` where the communication with the user space is proceeding. Used character device registrates following five file operation handlers:

```
int urfs_open(struct inode *ino, struct file *filp);
```

This function is called when the device is opened. It creates a new connection by `conn_create()`. Returned pointer is stored into the *private_data* member of the *filp* parameter. All other operation handling functions will be called with the same *filp* parameter, so this is used to share the pointer to the current connection. This provides the support for more simultaneous user space use of urfs module.

```
int urfs_release(struct inode *ino, struct file *filp);
```

Destroys the current connection.

```
unsigned int urfs_poll(struct file *filp, struct poll_table_struct *wait);
```

This provides the user space to wait for messages. The wait lasts until there is any message put into the list of messages to send for the current connection by `conn_msg_append()` or `conn_msg_insert()`.

```
ssize_t urfs_write(struct file *filp, const char __user *buff,
                   size_t count, loff_t *offp);
```

The write handling function takes care about incoming messages from the user space buffer *buff*. In one write, always one and only message comes. The function `urfs_write()` analyses the command code of the message and then it calls appropriate function which will process the message.

```
ssize_t urfs_read(struct file *filp, char __user *buff,
                  size_t count, loff_t *offp);
```

The read handling function puts right one message to the user space buffer *buff*. This message is taken out from the list of messages to send of the current connection.

Incoming command messages are processed by separate functions. The prototype of all is similar to this.

```
int process_in_cmd_x(struct conn *c, union imsg *imsg);
```

The letter "x" is substituted by a command name. The parameter *imsg* is the input message union. Data needed to process the command are stored there. If this command is user space originated (now there is only one case this is not true) then output reply message is created and inserted into the list of messages to send.

To process a kernel originated output command, there are functions of this prototype:

```
int process_out_cmd_x(struct ufilter *ufilter, struct request *request);
```

There is only one function of this type so far and it's `process_out_cmd_op_callback()`. It creates an output message and appends it to the messages to send list.

User space structures

Structures that are used in the kernel, such as *dentry* or *inode*, cannot be passed to the user space directly. The reason is that they contain many kernel related things which make no sense in a user space context. Therefore every kernel structure that appears in operation's arguments is converted to its urfs equivalent structure. It contains only members that are demanded to use. This list can be extended in the way a user space filter writer needs.

Operation callback event life cycle

The life cycle of one event will de decsribed for the better understanding of its processing. Consider a user space application with an opened connection to urfs, a registered ufilter, at least one operation callback set and the path to some testing file set.

1. Some user space application calls the operation on the testing file.
2. RedirFS catches this call and runs the registered `ufilter_generic_cb()`.
3. A new request with a unique id is created.
4. The function `process_out_cmd_op_callback()` is called with the pointer to the ufilter and the pointer to the request. After this call a waiting for the request competion is started.
5. A new output command message is created and appended to the list in the function `process_out_cmd_op_callback()`.
6. The append of the message breaks the user space `poll()` function and the user space application calls `read()` to get the message.
7. The user space application processes the message and calls a command to get operation arguments.
8. The user space callback is called, a reply message is assembled and `write()` is called to answer.
9. Run goes back to the kernel where an input message is processed by the function `process_in_cmd_op_callback()`. Then it completes the request which unblocks the original `ufilter_generic_cb()` which will finish the callback.

Extensions and improvements

So far there are implemented only operations and their arguments that are used by demo applications. If there would be need to use the different operation or to use some other arguments, it is easy to extend current sources acconding to that.

The thing that could be made the better way is the obtaining of request id.

7.2 urfs userspace library

The user space library called *liburfs* was designed in respect for an easy filters portability. Therefore the original RedirFS API prototypes remained. Two structure types are used. They are defined in `urfs.h`.

- `struct urfs_conn` - Is used to store a file descriptor of an opened connection. It also contains an array of pointers to subordinated user filters.
- `struct urfs_filter` - Instance of this structure is made for every single ufilter. It stores a pointer to the parent connection and arrays of pointers to callback functions.

The liburfs API defines several functions. These are working with the urfs_conn and urfs_filter structures and they are implemented in `urfs.c`. They are following ones:

```
int urfs_open(struct urfs_conn *n);
void urfs_close(struct urfs_conn *n);
```

The first of these two functions opens the urfs chardevice nodfile and saves its file descriptor to the structure referenced by the parameter *n*. It also zeroes an array of user filters in the structure. The second closes the file descriptor of given connection only.

```
int urfs_filter_alloc(rfs_filter *filter, struct urfs_conn *c);
void urfs_filter_free(rfs_filter filter);
```

The first function should be called after `urfs_open()` and before any other filter manipulating function which most probably should be `rfs_register_filter()`. It allocates a memory for the urfs_filter structure and it sets its connection pointer. The second function only frees the previously allocated memory.

```
int urfs_main(struct urfs_conn *c, rfs_filter filter);
```

The urfs main loop is implemented by this function. It should be called when all filters are already registered and set up. When it is called, first it does the command which enables callbacks. It also sets a signal handler for common termination signals which will cause the loop break. Then it goes into an infinite loop where it waits for callbacks and handles them by calling registered user callbacks.

The other functions in the `urfs.c` file implement the original RedirFS API prototypes. Each assembles a command message from its parameters and sends the message to the kernel module. Then it waits for a response and reads the message. A return value is usually the one that came in the reply message.

Only the implementation of `rfs_set_operations()` is more complicated and it deserves more care. Here comes its prototype:

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

This function processes the *op_info* parameter. It contains pointers to user space callbacks. The pointers will be stored to the urfs_filter structure referenced by the `filter` parameter. There is also an array of flags created. The array contains flags for each operation code. These flags specify whether it's defined pre-callback and post-callback. This array is passed to the kernel module and it registers generic callbacks according to flags.

There are also few static functions in `urfs.c` which deserve to look at. They are used for every communication need of any previously mentioned functions.

```
int wait_for_msg(struct urfs_conn *c);
```

This is used to wait for a message from the kernel and it returns only if the message come. It's implemented using the `select()` function instead of `poll()`, but it internally calls the `poll()` syscall as well. The reason is that I'm used to do it this way.

```
int imsg_send(struct urfs_conn *c, union imsg *msg);
```

To send a message to the kernel part, this function should be used. It uses `write()`.

```
int omsg_recv(struct urfs_conn *c, union omsg *msg);
```

This function receives a message from the kernel part of urfs. It uses `read()`.

```
send_and_receive(struct urfs_conn *c, union imsg *imsg, union omsg *omsg);
```

This combinates the previous two.

7.3 Demo applications

Demo applications' sources can be found in the `demos/` directory. They need liburfs to be compiled and installed before because they compile against the dynamic library. All demo applications perform operations this way:

1. Opens a connection to the urfs kernel module.
2. Allocates a filter.
3. Registers the filter.
4. Set filter's operations.
5. Set filter's paths.
6. Activate the filter.
7. Goes to the main loop to process callbacks.
8. Deactivated the filter

9. Unregisters the filter
10. Closes the connection

There were so far two user space filters created.

- *udummyflt* - It's a dummy filter remake for the user space. It does the same thing as its kernel opposite.
- *scrambleflt* - This was done for a demonstration of processing reads and writes. It simply crypts files by swapping nibbles of each char.

Chapter 8

Epilogue

This thesis brought a description of all communication methods that can be found in the current Linux kernel and in this consists its uniqueness. First, the eye is put on the system calls which are the base part of any other inter-space communication. Then the other mechanisms that are based on the system calls were presented. The procfs, the sysfs, the Netlink socket, the character devices and the relay interface were described with their implementation issues. This part of text could be useful for developers to familiarize with each method. It can help to decide what to choose for use in a particular solution. In can also help with the implementation because many methods are poorly documented and a look for inspiration in kernel sources can be a long distance run sometimes.

Another thing that a developer may appreciate are bandwidth and latency tests. The tests show a performance of each communication mechanism. They also can be useful as practical examples. The bandwidth tests were supposed to transfer a chunk of data from a buffer in the user space memory to another buffer in the kernel space memory and vice versa. The data chunk size was variable and the same sizes were used for each method. The winner of these tests were the system calls followed by the character devices. The latency tests were meant to show the time of feedback. The winner was the same as for bandwidth tests.

The choose of an inter-space communication method was affected by the portability of solution. In this point of view, there wasn't any choice because the only one mechanism, which can be found on the other systems too, are character devices. The test results for this method were also good and the implementation isn't so severe.

The RedirFS control tool called *redirctl* was developed. It consists of an extension of the original RedirFS kernel module and a new user space application. This application is command-line based and it can be called directly by a user or anywhere in a script. The communication between both parts is done by a character device.

The main goal was to develop an implementation of user space filters. This is a job of *urfs*. This solution consists of the kernel module and the user space library called *liburfs*. The communication is also handled by a character device. The library provides an opportunity to create and run the RedirFS filter in the user space. Of couse, there are some limitations of this solution which are caused by the running in the user space, where the filter cannot do so much things like it does in the kernel space. But it can be still very useful for many things, i.a. debugging of filters.

There are some things which can be done in the future. So far, there are a few operations supported by liburfs. They are the necessary ones that are used by the demo applications. The set of operations should be extended as well as an amount of transferred operation

arguments. This extension should be made according to needs of possible user filters. Therefore the case studies of the filter types which could be implemented in the user space should be done.

Also interesting will be the comparing of the performance of a kernel filter and the same filter implemented by liburfs. Of course, different types of filters may have different results. This comparsion should show an amount of overhead that rises by the communication and the user space run of a filter.

Bibliography

[1] Robert Love. *Linux Kernel Development, 2nd Edition*. Novell Press, 2005. ISBN 0-672-32720-1.

[2] J. Corbet, A. Rubini, and G.Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly, 2005. ISBN 0-596-00590-3.

[3] Pavel Herout. *Učebnice jazyka C, 4. přepracované vydání*. Kopp, 2005. ISBN 80-7232-220-6.

[4] František Hrbata. Callback Framework for VFS layer. Master's thesis, FIT VUT v Brně, 2005.

[5] M. T. Jones. Access the Linux kernel using the /proc filesystem. `http://www-128.ibm.com/developerworks/linux/library/l-proc.html` (March 2007).

[6] Kevin He. Why and How to Use Netlink Socket. `http://www.linuxjournal.com/article/7356` (April 2007).

[7] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. RFC3549 - Linux Netlink as an IP Services Protocol. `http://rfc.net/rfc3549.html` (April 2007).

[8] Kernel Objects and Sysfs. `http://docs.blackfin.uclinux.org/doku.php?id=kernel_objects` (March 2007).