
PROSO-Project 2

FIB 2011-2012

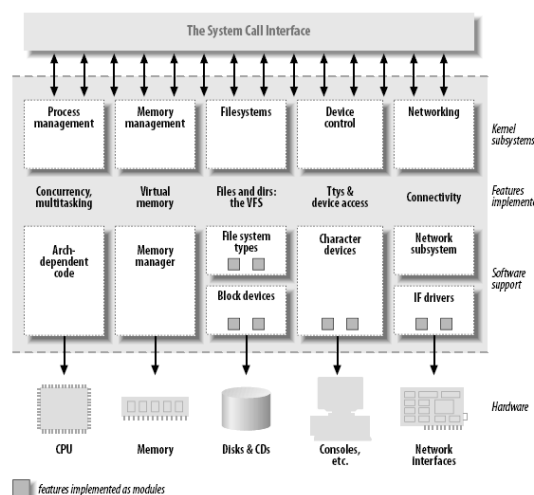
Objectives

- Put into practice in a real operating system, the concepts learn so far with ZeOS
 - System calls
 - Kernel data structures
 - Device drivers
- Get familiar with the development of Linux Kernel
 - Programming tools
 - Restrictions

Basic Concepts

- Modules
 - Means to add new functionalities to the Linux Kernel
 - Dynamically added/removed
- Device Drivers
 - Uniform APIs
 - Kernel <--> driver
 - User programs <--> drivers
 - Generic mechanism to access “devices”
 - Real devices (disk, keyboards, etc.)
 - Virtual devices (e.g. ram disk)
 - [Information from kernel components](#)

Kernel Modules and Device Drivers



Linux modules

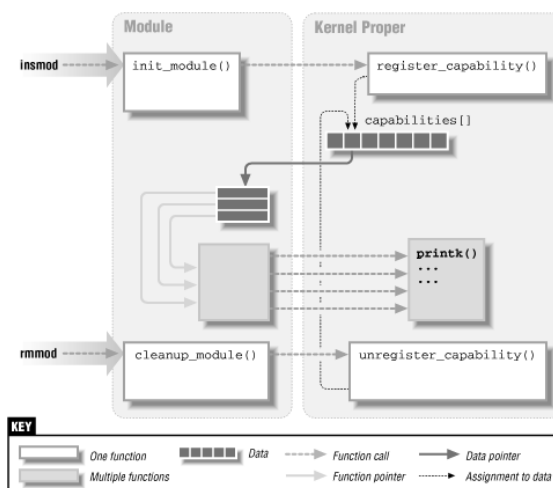
Modules

- Mechanism for adding dynamically functions to the kernel
 - Alternative is adding new `sys_calls`, but this requires rebuilding the kernel
- Same development limitations than other kernel components
 - Only kernel exported symbols can be accessed/modified
 - No access to `libc!`
 - Limited debugging tools (e.g. `printk`)

Kernel module development

- Program files that implement the module
 - Provide initialization and termination functions
 - Register functions to the kernel
 - Export functions to other modules
- Compile them
 - Produce object file (.ko = kernel object)
 - Requires kernel sources
- Insert in the kernel
 - Load module & dependencies
 - Pass initialization parameters
- Use it
 - maintain reference count

Modules and Linux Kernel



Module definition: example

```
#include <linux/module.h>
#include <linux/kernel.h>

/*
 * Module initialization.
 */
static int __init Mymodule_init(void)
{
    ...
}

/*
 * Finalization module.
 */
static void __exit Mymodule_exit(void)
{
    ...
}
module_init(Mymodule_init);
module_exit(Mymodule_exit);
```

Module definition : Macros

- **module_param** (parameter name and type)
 - int pid=1;
 - module_param (pid, int, 0);
- **MODULE_PARM_DESC** (parameter description)
 - MODULE_PARM_DESC (pid, "Process ID to monitor (default 1)");
- **MODULE_AUTHOR** (author list)
- **MODULE_DESCRIPTION**
- **MODULE_LICENSE** (GPL, BSD, ...)

Module management

- Install a module
#insmod mymodule.ko ^{initialization parameters} param=value, param=value
- Remove a module
#rmmod mymodule.ko
- Install a module and resolve dependencies
/lib/modules/version/modules.dep

```
/path_complet/modulA.ko:path_complet/modulB.ko  
/path_complet/modulB.ko:
```


#modprobe modulA.ko
- List information about a module
 - #modinfo mymodule.ko
 - #cat /proc/modules

Managing references to modules

- A module should be removed only when nobody is accessing the functions it provides
- Maintain internal counter of references
 - try_module_get (THIS_MODULE): Inc counter
 - module_put (THIS_MODULE): Dec counter
- For device driver related modules, the kernel can manage this automatically

Using the Linux kernel

- Lots of functions available for data structure management
 - find_task_by_pid
 - for_each_process
 - ...
 - Don't repeat existing functionality!
- Access symbols
 - only exported symbols are available
 - Look at /proc/ksyms or execute “ksyms -a” command
 - If not currently exported
 - modify kernel/ksyms.c
 - EXPORT_SYMBOL (variable)
 - Kernel recompilation is needed

Using the Linux kernel

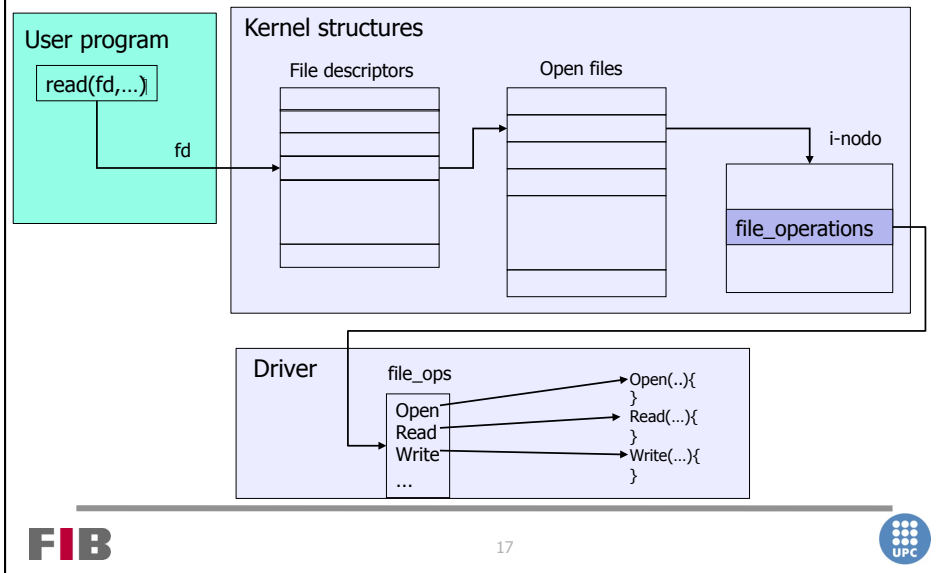
- Accessing to/from user address space
 - unsigned long copy_from_user(void *to, const void *from, unsigned long count);
 - unsigned long copy_to_user(void *to, const void *from, unsigned long count);
 - Validate return values
 - Different than ZeOS!!!

Printing messages

- Print message in the kernel using *printk*
 - `printk(KERN_<level> "message", param, param, . . .);`
 - *Different levels of messages*
 - `KERN_EMERG`
 - `KERN_ALERT`
 - `KERN_CRIT`
 - `KERN_ERR`
 - `KERN_WARNING`
 - `KERN_NOTICE`
 - `KERN_INFO`
 - `KERN_DEBUG`
- *Output goes to /var/log/message*

Device Drivers

Device independence



Device drivers

- Set of variables and functions that manages a device (logical or physical)
- Device driver definition: API standard
 - Internal API (not user-level)
 - based on the struct `file_operations`
- We have to provide only the functions required by the device (e.g. open, read)
- How to include a device driver in the kernel?
 - Statically: recompile the kernel
 - Dynamically: implement as a module

Device's operations

- Device driver definition: API standard

```
struct file_operations my_operations = {  
    owner: THIS_MODULE,  
    read: my_read,  
    ioctl: my_ioctl,  
    open: my_open,  
    release: my_release,  
};
```

maintain reference
counter automatically

- Look into `<linux/fs.h>` for types, etc.

Device drivers API

- Executed at open/close
 - int `my_open` (struct inode * i, struct file * f);
 - int `my_release` (struct inode * i, struct file * f);
- ssize_t `my_read` (struct file * f, char * buffer, size_t_size, loff_t * offset);
 - Use `copy_to_user` for accessing the buffer
 - Offset is input/output parameter. Current position in "file"
- int `my_ioctl`(struct inode * i, struct file * f, unsigned int request, unsigned long argp);
 - Used for control operations

Device identification

- Identified by a major and a minor
 - major: identifies a class of device (e.g. a printer)
 - minor: identifies different devices of the same class (i.e. two different printers)
- Allows the kernel to know which driver handles a device
- Match device's file major and minor

```
crw-rw-rw-  1 root  root    1,  3 Apr 11  2002 null
crw-----  1 root  root   10,  1 Apr 11  2002 psaux
crw-----  1 root  root    4,  1 Oct 28 03:04 tty1
crw-rw-rw-  1 root  tty    4,  64 Apr 11  2002 ttys0
crw-rw----  1 root  uucp   4,  65 Apr 11  2002 ttyS1
crw--w----  1 vcsa  tty    7,  1 Apr 11  2002 vcs1
crw--w----  1 vcsa  tty   7, 129 Apr 11  2002 vcsa1
crw-rw-rw-  1 root  root    1,  5 Apr 11  2002 zero
```

Device registration

- Device identifier must be registered inside the kernel:
`int register_chrdev_region (dev_t first, unsigned int count, const char *name);`
- To unregister:
`void unregister_chrdev_region (dev_t first, unsigned int count);`

Assign operations to devices

- First, create a new cdev structure:

```
struct cdev * cdev_alloc()
```

- Second, initialize the structure fields

- owner: with *THIS_MODULE*
- ops: with the *file_operations*

- Finally, assign this structure to the devices:

```
int cdev_add (struct cdev *dev, dev_t num, unsigned int count);
```

- To delete it:

```
void cdev_del (struct cdev *dev);
```

Inserting a new device driver dynamically

- Create a module with:

- Device driver functions
- New struct file_operations variable
- New device dev_t
- New structure cdev
- At init_module module
 - Register the device into the kernel:
 - Allocate the device identifier and associate the file_operations
- At cleanup
 - Unregister the device + Delete the cdev

How to use a new device?

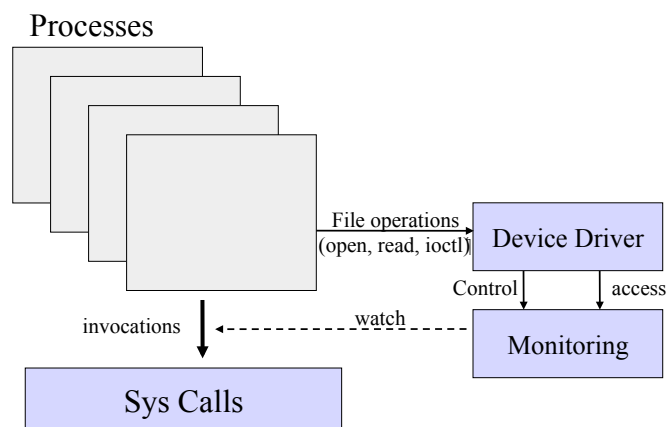
- Create a file with the `mknod` command using the new device's identification
 - `mknod <type> <major> <minor>`
 - e.g. `mknod mydriver c 255 1`
- Access the new file with standard I/O API
 - Open, read, write, close, etc

Description of work

Overview

- Develop a monitoring mechanism to measure the invocation of selected system calls
 - number of invocations
 - execution time
- Activate/deactivate dynamically this monitoring
- All processes are monitored (including those created after monitoring has started)
- Read monitoring information for a given running process

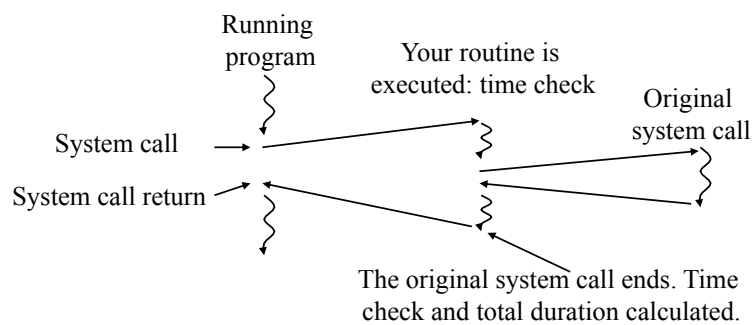
Monitoring processes



Module 1

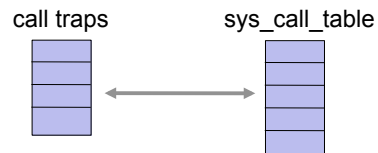
- Get per process information about *open*, *write*, *clone*, *close* and *lseek* system calls
 - How many times each call is executed
 - How many times they success
 - How many times they fail
 - Total time spent in each system call

Intercepting system calls



How to get the information?

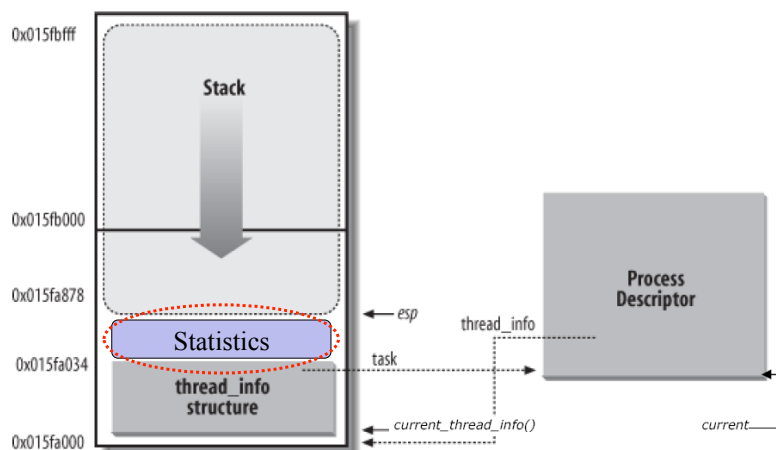
- Instrument the kernel by substituting original entries in `sys_call_table` by new ones



- On each call, the trap must:

- Get initial time → see the documentation
- Execute original system call
- Calculate execution time
- check call's return code
- Save information

Where to save information



Module 2: Access to information

- Open → Open the device
 - Only root and only 1 open
 - Defines `selected_process=current`, `selected_call=open`
- Read → Return statistics for the `selected_process` and `selected_call`
- Ioctl → Set the behaviour of the device
 - `CHANGE_PROCESS ==` Change selected process
 - `CHANGE_SYSCALL ==` Change selected syscall
 - `RESET_VALUES ==` Reset statistics of `selected_process`
 - `RESET_VALUES_ALL_PROCESSES ==` Reset statistics of all processes
- Release → Close the device

Improvements

- Module 1
 - Two new functions to enable/disable the instrumentation of one of the system_calls (*open*, *write*, *clone*, *close* and *lseek*)
 - It is mandatory to use a table to store original_syscalls addresses
- Module 2
 - Two new options in *ioctl* to enable/disable the instrumentation of one of the system_calls (*open*, *write*, *clone*, *close* and *lseek*)
 - » `ACTIVAR_SYS_CALL ==` enable
 - » `DEACTIVAR_SYS_CALL ==` disable
 - » Use functions implemented in Module 1

What to do?

- Module 1 and Module 2 → 80%
- Improvements → 20%

- You have to include exhaustive user tests to validate your modules:
 - Errors
 - Returns values
 - Expected functionality
 - ...
- It is mandatory to provide some .h where data structures and constants required by user codes will be declared