

Outlines

- Faire des modules
- Interaction via /sys
- Memory alloc
- Interaction via /dev
- Threading
- Hardware?
- Suicide: bottom/top kernel, irq, workqueues etc

Repo du WS: <https://github.com/UrLab/kernelworkshop>

Pad pour échanger du code: <https://pad.lqdn.fr/p/kernelurlab>

WARNING



Level¹: Hello world

- Deux fonctions needed

- `int init_module(void) // return 0 if ok`
- `void cleanup_module(void)`

- Deux headers needed

- `#include <linux/module.h> // requis pour de la magie`
- `#include <linux/kernel.h> // KERN_INFO`

- `printk`: comme `printf`, mais pour le kernel

- `KERN_WARNING`, macro string pour gérer la gravité de la situation
`printk(KERN_INFO "..", args)`

- `panic()` ;yolooooo

- Compilation: Makefile

```
CFLAGS=""
```

```
obj-m += licorn.o
```

```
all:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD)  
modules
```

- `insmod licorn.ko`

- `dmesg ;swag.`

Level¹++

- Use `module_init`, `module_exit` macro (exemple: `module_init(start_licorn_factory)`);
- Use macro:
 - `MODULE_AUTHOR("URLAB LICORN CORP");`
 - `MODULE_DESCRIPTION("The end of your life as you know it");`
 - `MODULE_LICENSE("GPL") // in rms we trust.`
- Use `static __init`, `static __exit` before your init code and exit code
 - **`static int __init`** `start_licorn_factory(void)`

Userspace interaction

- Kernel space vs userspace
- Moyen de transmission:
 - syscall
 - basé sur syscall: mécanisme de “fichier” (open/read/write/ioctl/close).
Simplification, voir socket etc
- Explorez /sys (le kernel emploie plein d'objets, hiérarchie etc)
- <http://lxr.free-electrons.com/> !!!! votre nouveau meilleur pote

Level²: Hello licorn %name

- L'api principale c'est via des choses qui s'appellent des kobjects (on va y revenir après)
- C'est une hiérarchie. Elle est exportée dans /sysfs. Votre module peut mettre des dossiers (des kobjects) et des fichiers dedans
- Le module doit enregistrer un nouveau kobject, qui va posséder des fichiers (dans le init du module)

- `struct kobject custom_dir;`
- `kobject_init(&custom_dir, &module_type);`
- `kobject_add(&custom_dir, &THIS_MODULE->mkobj.kobj, "custom");`

- `module_type`? Préciser la manière de read/write les fichiers qui sont dedans

```
struct sysfs_ops module_file_ops = {
    .show = licorn_magic_show,
    .store = licorn_store
};
struct kobj_type module_type = {
    .sysfs_ops = &module_file_ops
};
```

- Fonction de read/write:

```
static ssize_t licorn_magic_show(struct kobject *obj, struct
attribute *attr, char *buf) {}
static ssize_t licorn_store(struct kobject *obj, struct attribute
*attr, const char *buf, size_t count) {}
```

Level²: Hello licorn %name

- Résoudre sur base du attr->name, retourner quelque chose en fonction de ça (rien, un string,...)
- pour déclarer un fichier:
- `sysfs_create_file(&custom_dir, &name_file_attr);`
- ```
struct attribute name_file_attr = {
 .name = "name",
 .mode = 0666
};
```

→ algo:

```
module_init() {
 kobject_init(...);
 kobject_add(...);
 sysfs_create_file(&custom_dir, &name_file_attr);
}
```

```
module_cleanup() {
 sysfs_remove_file(&custom_dir, &name_file_attr);
 kobject_put(&custom_dir);
}
```

Idea: par exemple, déclarer deux fichiers, un contenant le nom (qui serait copier dans un string dans le module), un uniquement writable, qui quand il est appelé fait un `printk("Hello licorn %s", name)`.

# Memory management

- `#include <linux/slab.h>`
- `Kmalloc (kzalloc), kfree (easy)`
  - `void * kmalloc(size_t size, int flags); //flags = GFP_KERNEL`
  - `void kfree(void * ptr); // NULL ok.`
- More efficient way, but lot more complex.
- Ldd3 is now your best friend

<https://www.kernel.org/doc/Documentation/CodingStyle>



# Linked liste

- `#include <linux/list.h>`
- Surprise! C'est un truc à mettre à l'intérieur d'une struct, pas un truc qui englobe une struct. Par exemple:
  - ```
struct love {  
    char buffer[128];  
    struct list_head more_love;  
};
```
- `struct list_head rainbows; init: INIT_LIST_HEAD(&rainbows);`
- `list_add(love_instance->more_love, &rainbows)`
- `struct love * ptr; list_for_each_entry(ptr, &rainbows, more_love)`
{
- `ptr` est 1 elem de rainbos, `ptr->data = ...`
- `list_del(&love_instance->more_love)`
- gaffe au segfault, voir `list_empty`.
- LDD3 at page 313

Level³: A very fat licorn

Algo idea:

- dans init: faire un dossier custom, puis initialiser une liste global (une list_head)
- faire une fonction add_love(..) qui kmalloc une nouvelle struct love, copie de la data d'userspace dedans, puis fait un list_add (nouvelle instance de love, liste globale)
- faire une fonction list_all_love qui emploie l'itérateur vu plus haut pour en afficher le contenu
- faire un get_love qui récupère le premier elem de la liste, met son contenu dans buf puis le list_del && kfree

Warning:

- faire un list_empty dans le get_love sinon segfault
- pas multiprocess safe, voir avec des locks (pas sexy)
- pas oublié de kfree une non empty list dans le clean_up

module parameters

- Mega easy
- Static <type> parameter = default_value;
- module_param(parameter, type, S_IRUGO);

/dev

- Main driver access point
- dev est repéré par un major et un minor (dev_t) → (ls /dev)
- `dev_t dev; alloc_chrdev_region(&dev, 0, number, "name");`
- `major = MAJOR(dev);`
- `cdev_init(cdev * device, &device_fops);`
- `device.owner = THIS_MODULE`
- `int devno = MKDEV(major, minor)`
- `cdev_add(device, devno, 1);`

Structure nécessaire

```
Struct licorn {  
    struct list_head rainbows;  
    struct cdev device;  
    int minor, readed;  
};
```

```
static struct licorn * licorns;
```

```
allouer de la mem dessus: licorns = kmalloc(sizeof(struct licorn) * X, ...)
```

```
for (l = 0; l < X; ++l) {
```

```
    init code ^
```

```
    elem.minor = l;
```

```
}
```

container_of

```
Struct thing {  
    struct attribute attr;  
    ...  
}
```

```
struct thing thing_instance = {  
    .attr = {.name = "my_name", .mode ...}  
    ...  
}
```

```
par exemple, le read reçoit (kobject, attr, ...) {  
    struct thing * my_thing = container_of(attr, struct thing, attr);  
    ...  
}
```

/dev, read/write

- Plus complexe
- ```
struct file_operations licorn_fops = {
 .owner = THIS_MODULE,
 .read = licorn_read,
 .write = licorn_write,
 .open = licorn_open
};
```
- ```
static ssize_t licorn_write(struct file * filp, const char __user *buf, size_t len, loff_t * o);
```
- ```
static ssize_t licorn_read(struct file * filp, char __user * buf, size_t len, loff_t * o);
```
- ```
static int licorn_open(struct inode *inode, struct file *filp);
```
- filp a un espace private_data, que vous pouvez set à l'open, en chopant l = container_of(inode->i_cdev, struct licorn, device);
- il faut obligatoirement que le read retourne un moment 0 sinon il va boucler (easy fix: mettre une var dans struct licorn, qui est reset à 0 à chaque open et set à 1 à chaque read)