

Linux Kernel Module

Intro au dev

Flavien ASTRAUD

{EPITECH.}

Oct. 2024

Plan

Introduction

- Principe & Objectif
- Déroulement
- Pré-requis

Kernel Linux

- Définition
- Grands principes / Caractéristiques
- Historique bref
- Ressources

Kernel module

- Package
- Modules de mon kernel

Un module minimal : "Hello world !

- Pré-requis (headers, package)
- Sources minimales
- Compilation
- À vous de jouer
- Solution ex 0
- Solution ex 1

Les paramètres d'un module

- Définition / initialisation
- Démo
- À vous de jouer
- Aller plus loin

Le debugfs

- Définition
- Code
- À vous de jouer

Surcharge d'un appel système

- Notions
- Implémentation

Timers

- Timers : définition
- Delay
- Timer API / High Resolution Timer (hrtimer)

procfs & sysfs

- procfs
- sysfs

api atomic_t

Conclusion

la pratique : votre projet

- Contexte
- Objectif
- Livrables
- Fonctionnalités
- Exemple de validation
- Critères de validation

Plan

Introduction

Principe & Objectif

Déroulement

Pré-requis

Principe & Objectif

- Aborder simplement par l'exemple le dev LKM
- Installer les outils et pré-requis
- Coder & compiler un module simple
- Connaître les notions de base

Déroulement

- 2 jours de cours :
 - Introduction de notions
 - Exemples → “livecoding”
 - Travaux pratiques
- 1 projet :
 - 1 follow-up
 - delivery

Technique

- Maîtrise du C
- Connaissance du Shell et de l'environnement Linux
- Notion d'architecture (Mémoire, cpu, ...)

Infra

- Disposer d'une VM Linux
- version du kernel : 6.8 (environ)
- Les exemples du cours seront effectués sur une "Ubuntu 24.04.1 LTS"

Plan

Kernel Linux

- Définition

- Grands principes / Caractéristiques

- Historique bref

- Ressources

Définition

- Qu'est-ce que c'est ?
 - Un programme (principalement du C)
 - Le coeur du système
- À quoi ça sert ?
 - fournir une API pour utiliser le matériel
 - Organiser les processus : partage des ressources

Grands principes / Caractéristiques

- Langages : C, assembleur, RUST (nouvellement)
- Architectures : monolithique avec des modules
- Multi-tâche & multi-utilisateurs
- POSIX
- Famille des UNIX

Historique bref

- Minix → Linux
- 0.01 : 17 septembre 1991
- 1.0.0 : 14 mars 1994

Ressources

- Le code source du noyau Linux est disponible sur le site :
<http://www.kernel.org/>
- https://fr.wikipedia.org/wiki/Noyau_Linux
- <https://sysprog21.github.io/lkmpg/>
- <https://kernelnewbies.org/>
- Répertoire "Documentation" dans les sources du kernel
- Sur irc : #kernelnewbies sur irc.oftc.net

Plan

Kernel module

- Package

- Modules de mon kernel

Package

- Installation des commandes pour gérer les modules :

```
apt install build-essential kmod
```

- modprobe : charger / décharger un module
- insmod : charger un module
- rmmod : décharger un module
- modinfo : affiche les informations

Modules de mon kernel

- lsmod & modinfo
- cat /proc/modules
- lsmod | grep MON_MODULE

Plan

Un module minimal : "Hello world !

- Pré-requis (headers, package)

- Sources minimales

- Compilation

- À vous de jouer

- Solution ex 0

- Solution ex 1

Un module minimal : "Hello world !

“Lorsqu’on débute, on commence par *“Hello world !”*.

Je ne veux pas savoir ce qu’il se passerait si on enfreint cette coutume.”

[<https://sysprog21.github.io/lkmpg/>]

└─ Un module minimal : "Hello world !

└─ Pré-requis (headers, package)

Pré-requis (headers, package)

- Il faut installer les headers du kernel.
- un compilateur et outils si nécessaire.
- Facultatif : installer les sources du kernel.
- debug

```
apt install gcc make
↳ linux-headers-`uname -r`
apt install linux-source
cp
↳ /usr/src/linux-source-6.8.0/linux-so
↳ /tmp
cd /tmp
tar xvf linux-source-6.8.0.tar.bz2
```

Sources minimales

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

int init_module(void)
{
    pr_info("Hello world 0.\n");

    return 0;
}

void cleanup_module(void)
{
    pr_info("Goodbye world 0.\n");
}

MODULE_LICENSE("GPL");
```

makefile

■ Le makefile :

```
obj-m += hello-0.o

PWD := $(CURDIR)

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

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

- └ Un module minimal : "Hello world !"

- └ Compilation

Compilation

- `make`

Vérification

- `modinfo ./hello-0.ko`

```
filename:      /home/k/kern_init/./hello-0.ko
license:       GPL
srcversion:    93D1874613E2633FD08C40F
depends:
retpoline:     Y
name:          hello_0
vermagic:      5.15.0-67-generic SMP mod_unload modversions
```

└ Un module minimal : "Hello world !

└ À vous de jouer

À vous de jouer

- ex 0 :
 - Ajouter l'auteur,
 - Ajouter la description dans les sources du module
- ex 1 :
 - Personnaliser le nom des fonctions `init` & `cleanup`

└─ Un module minimal : "Hello world !"

└─ Solution ex 0

Solution ex 0

- ex 0 :

```
MODULE_AUTHOR("YOUR NAME <email>");  
MODULE_DESCRIPTION("hello 0 à epitech");
```


Solution ex 1

■ ex 1 :

```
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

static int __init hello_1_init(void)
{
    pr_info("Hello, world 1\n");
    return 0;
}

static void __exit hello_1_exit(void)
{
    pr_info("Goodbye, world 1\n");
}

module_init(hello_1_init);
module_exit(hello_1_exit);

MODULE_LICENSE("GPL");
```

Plan

Les paramètres d'un module

- Définition / initialisation

- Démo

- À vous de jouer

- Aller plus loin

Définition / initialisation

- `module_param`
- `module_param_array`
- `MODULE_PARM_DESC`

Démo

```
sudo insmod ./kdev_param.ko str="TST 0"  
ls -l /sys/module/kdev_param/parameters/  
cat /var/log/syslog  
sudo bash -c "echo 1,2,3 > /sys/module/kdev_param/parameters/a"  
sudo bash -c "echo 21 > /sys/module/kdev_param/parameters/n"  
sudo rmmod kdev_param  
cat /var/log/syslog
```

À vous de jouer

- ex 2 : ajouter des paramètres à votre exemple hello-0
- sol. 2 : cf. `kdev_param.c`

Aller plus loin (en fct du temps) : CallBack

Surveiller la modification d'un paramètre :

- `module_param_cb`
- cf. `kdev_param_cb.c`

Plan

Le debugfs

- Définition

- Code

- À vous de jouer

Définition

- Simple moyen de rendre dispo. des informations au niveau utilisateurs
- Permet de modifier les variables du module
- Pas de règles (contrairement /proc & sysfs)
- `mount -t debugfs none /sys/kernel/debug`
- Documentation/filesystems/debugfs.rst

Code

- Création / suppression :

```
struct dentry *debugfs_create_dir(const char *name,  
                                  struct dentry *parent);  
void debugfs_remove(struct dentry *dentry);
```

- **Attention : : Nécessité de la suppression**
- Associer une variable "simple" à un fichier :

```
void debugfs_create_u64(const char *name, umode_t mode,  
                        struct dentry *parent, u64 *value);
```

À vous de jouer : ex 3

- écrire un module qui prendre un int en paramètre et l'expose dans le debugfs.
- Modifier la valeur avec "cat/echo"
- Afficher la valeur au déchargement

Exemple de code : ex 3

cf. `debugfs_lkmpg.c`

Plan

Surcharge d'un appel système

Notions

Implémentation

Notions

- La table des syscall
- Hooks (l'exception vdso)
- Les dangers

Implémentation

cf. `kstatx.c`

Plan

Timers

- Timers : définition

- Delay

- Timer API / High Resolution Timer (hrtimer)

Timers : définition

- Delay
- horloge à intervalle spécifique
- déclencher des traitements
- timers api / High Resolution Timer

Delay

- `timers-howto.rst`

timer / hrtimer

- `timers_00.c`
- `hrtimers_0[01].c`

Plan

procfs & sysfs

procfs

sysfs

procfs : kdev_procfs.c

- Création / suppression de l'entrée dans /proc/

```
static struct proc_dir_entry *kdev_tst_proc_file;  
kdev_tst_proc_file = proc_create(PROC_NAME, 0644, NULL, &proc_file_fops);  
proc_remove(kdev_tst_proc_file);
```

- Création de la structure proc_ops

```
static const struct proc_ops proc_file_fops = {  
    .proc_read = procfile_read,  
    .proc_write = procfile_write,  
};
```

- Implémentation des fonctions read & write

```
static ssize_t procfile_read(struct file *fp, char __user *buf,  
                             size_t len, loff_t *offset)  
static ssize_t procfile_write(struct file *fp, const char __user *buf,  
                              size_t buf_len, loff_t *offset)
```

sysfs : kdev_sysfs.c

- Création / suppression des entrées dans /sys/kernel
- Les structures de données kobject, struct attribute_group et struct attribute
- Fonctions read & write
- Documentation/filesystems/sysfs.txt
- Documentation/kobject.txt

Plan

api atomic_t

api atomic_t

- Opérations arithmétiques : **en une instruction sans interruption**
- les types : `atomic_t` `atomic64_t` `atomic_long_t`
- `Documentation/atomic_t.txt`

Plan

Conclusion

Conclusion

Conclusion

Plan

la pratique : votre projet

- Contexte

- Objectif

- Livrables

- Fonctionnalités

- Exemple de validation

- Critères de validation

Contexte

- *otp* : one time password
- Mise en pratique de votre apprentissage de dev kernel
- Réflexion, imagination pour trouver des solutions

Objectif

- Mettre en place un système (lkm + outillage) visant à offrir une solution *otp* via un ou plusieurs "device" et un ou plusieurs utilitaires.
- La solution doit mettre en oeuvre 2 méthodes d'*otp* :
 1. une liste de mot de passe
 2. un algo basé sur une clé et le temps.
- On doit envisager de créer plusieurs device pour gérer plusieurs *otp*.

Livrables

1. un module kernel (sources, makefile)
2. un ou plusieurs utilitaires (c ou script)
3. la documentation associée (peut être contenu dans les sources)

lkm

- Créer un ou plusieurs devices
- Configurer le device
 1. clé / Durée de validité
 2. Listes des mots de passe
- Permettre d'afficher / supprimer / modifier la clé, la durée, la liste

Utilitaires

- gestion du module :
 - Afficher / supprimer / modifier la clé, la durée, la liste (donc config du module)
- client otp : :
 - calculer un mdp (coder l'algo ou la liste)
 - vérifier un mdp

- Dans une fenêtre, on exécute `cat /dev/otpN` qui renvoie un otp
- Dans une autre fenêtre, on exécute un programme qui demande un otp et qui le valide

- avoir un lkm et les utilitaires associés
- 2 méthodes
- Qualité du code