Formation au Noyau Linux

Jérôme Pouiller <j.pouiller@sysmic.org>

Troisième partie III

Les modules

12 Macros de base

13 Licenses des modules

14 Développement des modules

15 Ecriture des Makefile
- A l'extérieur du noyau
- A l'intérieur du noyau

16 Charger des modules

17 Utiliser des paramètres
- Le Coding Style
- La documentation
- Règles de documentation
- Où trouver la documentation
- Trouver de l'aide

18 L'API

19 Gestion des erreurs

20 Communiquer avec le noyau
- Les appels systèmes
- Les fichiers devices

Les modules noyau

my_module

Un template de module :

```
// Declare special functions
module_init(m_init);
module_exit(m_exit);

// These uppercase macro will be added to
    informative section of module (.modinfo)
MODULE_AUTHOR("Jérôme Pouiller");
MODULE_DESCRIPTION("A pedagogic Hello World");
MODULE_LICENSE("Proprietary");
MODULE_VERSION("1.1");
```

Quelques macros de base

Ces macros permettent de placer des informations sur des symboles particulier dans module ;

- Déclare la fonction à appeler lors du chargement du module (traditionnellement module_init)

```
module_init
```

- Déclare la fonction à appeler lors du déchargement du module (traditionnellement module_exit)

```
module_exit
```

- Déclare un auteur du fichier. Peut apparaître plusieurs fois.

```
MODULE_AUTHOR
```

Quelques macros de base

- Description du modules

```
MODULE_DESCRIPTION
```

- Version du module

```
MODULE_VERSION
```

- Rendre un symbole visible par les autres modules. Il sera alors pris en compte dans le calcul des dépendances de symboles.

```
EXPORT_SYMBOL(symbol)
```

- Idem `EXPORT_SYMBOL` mais ne permet sont utilisation que pour les modules GPL

```
EXPORT_SYMBOL_GPL(symbol)
```

- License. Indispensable

```
MODULE_LICENSE
```

Parlons des licenses

- Le noyau est développé sous licence GPLv2
- La GPL indique que tout travail dérivé d'un projet sous GPL doit être GPL.
- Ainsi, un module propriétaire ne doit pas compiler en statique avec le noyau (le résultat est alors considéré comme un travail dérivé du noyau). Il doit rester sous la forme d'un module.
- Pour le l'utilisation dynamique des symboles, le débat n'est pas tranché. Le noyau laisse la possibilité à l'auteur d'exporter ses modules avec `EXPORT_SYMBOL` ou avec `EXPORT_SYMBOL_GPL`.
- Si vous développez un module Propriétaire, vous n'aurez pas accès à toute l'API du noyau (environ 90% seulement).
- Il est néanmoins possible de contourner le problème en utilisant un module intermédiaire comme proxy logiciel.

Les licences

En remplissant correctement la macro `MODULE_LICENCE`, le système de compilation du noyau empêchera les compilations illégales. `MODULE_LICENCE` peut prendre plusieurs valeurs : GPL, GPL v2, GPL and additionnal rights, Dual MIT/GPL, Dual BSD/GPL, Dual MPL/GPL. Toutes les autres valeurs sont considérées comme propriétaire.

`/proc/sys/kernel/tainted` (ou `uname -a`) indique si des modules propriétaires sont chargé.

Quels conséquences ?

- Pas de support de la part de la communauté
- Pas de garantie de la qualité du modules
- Mauvaise image auprès de la communauté et des clients

Par expérience, les modules propriétaires que l'on croise dans l'embarqué peuvent cacher tout et n'importe quoi.

Headers

Il n'est pas possible pour le noyau de compiler avec des bibliothèques extérieur. En particulier avec la `libc`. Tous les headers habituels sont donc inaccessible dans le noyau.

Les headers "publiques" sont dans `include/`. On retiendra en particulier : `linux/module.h`, `linux/init.h`, `linux/kernel.h` indispensables pour compiler un module.

Les modules noyau

my_module

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

static int __init m_init(void)
{
	pr_info("my_module init\n");
	return 0;
}

static void __exit m_exit(void)
{
	pr_info("my_module exit\n");
}
```

Les modules noyau

my_module

Fichier `Makefile` à l'extérieur du noyau :

```
obj-m := my_module.o
```

Puis, on appelle :

```
host$ KDIR=/lib/modules/$(uname -r)/build
host$ make -C $KDIR ARCH=arm M=$(pwd) modules
```

Il aussi est possible d'appeler les cibles `help` et `modules_install`

Les modules noyau

my_module

Pour améliorer le processus, on ajoute ces lignes dans le `Makefile` :

```
KDIR ?= /lib/modules/$(shell uname -r)/build

default: modules

modules:
	$(MAKE) -C $(KDIR) M=$(shell pwd) modules

modules_install:
	$(MAKE) -C $(KDIR) M=$(shell pwd)
	    modules_install
```

et on appelle

```
1 host$ make ARCH=arm KDIR=../linux-2.6/usb-a9260
```

Référence : `Documentation/kbuild/modules.txt`

Compilation avec Kbuild

Fichier `Makefile` à l'intérieur de l'arborescence noyau :

```
obj-$(MY_COMPILE_OPTION) := my_module.o
```

`$(MY_COMPILE_OPTION)` sera remplacé par :

- ø : Non compilé
- m : compilé en module
- y : compilé en statique

Fichier `Kconfig` :

```
config MY_CONFIG_OPTION
      tristate "my_module: A small Hello World
          sample"
      help
        Pedagogic purpose only.

        To compile it as a module, choose M here.
            If unsure, say N.
```

Utilisation de Kconfig

Chaque entrée `config` prend comme attribut :

- Son type et sa description en une ligne :
  - `tristate`, le plus classique pouvant prendre les valeurs ø, m et y
  - `bool` pouvant prendre seulement les valeurs n et y
  - `int` prennant un nombre
  - `string` prennant une chaîne
- `default` Sa valeur par défaut
- `depends on` L'option n'apparait si l'expression suivante est vraie. Il est possible de spécifier des conditions complexes avec les opérateurs `&&`, `||`, `=` et `!=`
- `select` Active automatiquement les options en argument si l'option est activée
- `help` Description détaillée de l'option. Si votre description ne tient pas en moins de 20 lignes, pensez à écrire une documentation dans `Documentation` et à y faire référence

Utilisation de Kconfig

Il est aussi possible :

- D'inclure d'autres fichiers avec `source`
- De déclarer un sous menu avec `menu`
- De demander un choix parmis un nombre fini d'options avec `choice`

Référence : `Documentation/kbuild/makefiles.txt`, `Documentation/kbuild/kconfig-language.txt`

Gérer les modules

- Avoir des informations sur le module

```
host$ modinfo my_module.ko
```

- Charger un module

```
target% insmod my_module.ko
```

- Décharger un module

```
target% rmmod my_module
```

- Afficher le buffer de log du kernel (Diagnostic MESsaGes)

```
target$ dmesg
```

Gérer les modules

- Charger/décharger un module correctement installé/indexé dans `modules.dep`

```
target% modprobe my_module
target% modprobe -r my_module
```

- Mettre à jour le fichier de dépendances

```
target% depmod
host$ depmod -b ~/rootfs
```

- Notons que les modules sont des binaires *elf* standards qui peuvent être analysés avec les outils standard :

```
host$ arm-linux-objdump -t my_module.ko
```

Paramètres

Il est possible de passer des paramètres aux modules :

```
target$ modinfo my_module.ko
target% insmod my_module.ko param=2
```

Il est possible de passer le paramètre au boot avec
`my_module.param=2`

Paramètres

Nous devons déclarer le paramètre à l'aide de la macro

```
module_param(name, type, perm);
```

- `name` : Nom de la variable utilisée comme paramètre
- `type` : Type du paramètre (`bool`, `charp`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`) (remarquez que ca ne correspond pas tout à fait aux types C. Particulièrement pour `bool` et `charp`)
- `perm` : Permissions du fichier `/sys/module/<module>/parameters/<param>`. (utiliser les macros de `linux/stat.h`). Si 0, le fichier n'apparaît pas.

La paramètre doit évidement être alloué :

```
static int param = 0;
```

Il est fortement recommandé de documenter le paramètre

```
MODULE_PARM_DESC(param, "Useless parameter");
```

/sys

Etudions `/sys`

- `/sys/module/my_module/parameters` : Les paramètres. Modifiables si déclarés modifiables
- `/sys/module/my_module/sections` : Des infos sur la zone de chargement

Le Coding Style

Quelques remarques sur la programmation à l'intérieur du noyau :

- Il n'est pas possible d'accèder aux fonctions de la libc (ni des autres bibliothèques) à l'intérieur du noyau
- Pas de C++ dans le noyau. C'est techniquement fesable mais déconseillé et sera refusé par l'upstream
- Il n'y a pas de garde fou pour la mémoire du noyau. Un accès mémoire invalide pourra potentiellement est rattrapé par un Oops, mais ca n'est pas garanti.
- Attention à la portabilité. Votre code ne doit pas être dépendant de l'endianess ou de taille des integer.

Le Coding Style

- On ne doit pas avoir besoin d'utiliser l'ascenseur (vertical ou horizontal) pour lire une fonction (bref : 80 colonnes et pas trop de lignes par fonction)
- Indentation à la tabulation
- ... et une tabulation fait 8 espaces
- Si l'indentation vous empêche d'écrire votre fonction, c'est que celle-ci possède trop de niveaux d'imbrication
- Mettre au propre les tabulations et vérifier la taille des lignes :

```
host$ scripts/cleanfile my_module.c
```

- Incolade sur la même ligne que les blocs
- ... sauf pour les fonctions
- Pour indenter correctement votre code

```
host$ apt-get indent
host$ scripts/Lindent my_module.c
```

CodingStyle

- Les noms des variables locales doivent être courts
- Si vous avez besoin de nom plus explicites, c'est que vous avez trop de variables locales
- Pas de CamelCase
- Il faut préserver la propreté de l'espace de nommage en préfixant par statique tout ce qui ne doit pas être exporté (fonctions, variables)
- Référence : `Documentation/CodingStyle`

La documentation

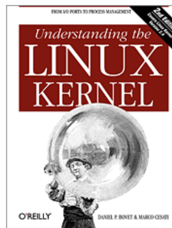
- Le noyau n'est pas un modèle de documentation
- En revanche, le système de développement facilite énormément la recherche d'informations
- Le noyau utilise un script perl appelé `kernel-doc`. Ce script parcours le code à la recherche de syntaxe du même type que doxygen. (Référence : `Documentation/kernel-docs.txt`)
- Un commentaire ne doit pas ce substituer à un code pas illisible. Préférez réécrire le code plutôt qu'ajouter un commentaire.
- Une fonction exportée doit être préfixée par un commentaire la décrivant. Ce commentaire ne doit surtout explique *comment* la fonction marche. Le commentaire indique le *quoi*, mais surtout le *pourquoi* cette fonction existe.
- Ces commentaires sont placée avant la *définition* des fonctions. Les fonction du noyau sont souvent définie dans les headers. Du coup, les commentaires sont parfois dans le fichier `.c` et parfois dans le fichier `.h`
- Les commentaires expliquant le fonctionnement général d'un framework doivent être écrit sous la forme d'une documentation

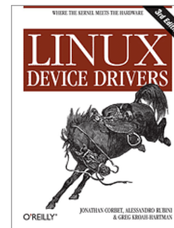
Trouver de la documentation

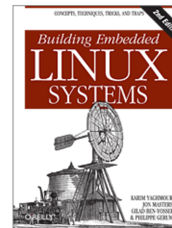
- Dans `Documentation` (en utilisant `grep`)
- La cible `htmldocs` permet de compiler les documentation au format Docbook (`Documentation/DocBook`)
- ... en particulier *Linux Device Drivers* et *Linux Device Drivers*. Toutefois, ces compilations sont loin d'être exhaustives
- La cible `mandocs` permet d'extraire les différente API invoquée dans les DocBook sous forme de pages de man.
- `http://kernel.org/doc` regroupe les diverses documentations en ligne
- Dans les commentaires du noyau (fichiers `.c` ou `.h`)
- `http://lxr.linux.no` offre une interface permettant de rapidement chercher des symboles et naviguer dans le code du noyau.

Les livres de référence

Les références :







- LWN : http://lvn.net
- Embedded Linux Wiki : http://elinux.org

Les références de nos amis de la communauté francaise :

- Gilles Blanc : Linux embarqué - Comprendre, développer, réussir
- Jérôme Pouiller : http://sysmic.org
- Thomas Petazzoni et Michael Opdenacker : http://free-electrons.com/docs/

sysmic

Trouver de l'aide

Demander de l'aide à la communauté

- Le fichier `MAINTAINERS` contient la liste des mainteneur de chaque sous-système
- Les entrées commencant par `L:` indique des *mailing lists*
- Notons `<linux-newbie@vger.kernel.org>`
- `MAINTAINERS` liste aussi les sites officiel, les fichiers de documentation, etc... pour les différents sous-projets du noyau
- `git blame` permet de savoir qui a modifier la fonctions ou les lignes qui nous intéresse. Cela permet de s'orienter vers le bon groupe
- Référence : `http://kernel.org/doc`

L'API non-stable

- L'API non-stable ne facilite pas la maintenance des documentations
- De même, la maintenance des modules hors du noyau demande beaucoup d'efforts
- C'est encore pire de maintenir un module pour plusieurs versions (driver nvidia par exemple)
- En revanche, l'API non stable permet des développement plus agiles. En fait, le noyau serait intenable si il essayait de maintenir une API stable.
- Lors qu'un développeur modifie une API, il doit patcher l'ensemble des driver utilisant l'ancienne API.
- On préfère souvent laisser l'ancienne et la nouvelle API cohabiter en demandant au maintainers de migrer vers la nouvelle

sysmic

L'API non-stable

- Il n'y a pas de raisons de maintenir un driver à l'extérieur du noyau, sauf les problèmes de licences
- Il peut être coûteux en temps d'intégrer un module dans le mainstream. Les mainteneurs vous demanderons sûrement de corriger beaucoup de chose.
- Il faut compter 2 à 4 fois le temps de développement initial pour intégrer un module dans le mainstream
- Une fois que votre module est intégré dans le mainstream et a passé la phase de staging, il sera théoriquement maintenu et supporté pour toujours.
- Paradoxalement, l'API non-stable permet que la durée de maintenance d'un driver dans le noyau sera longue (en théorie infinie)
- Référence `Documentation/stable_api_nonsense.txt`

Gestion des erreurs

- La plupart des fonctions retournant des `int` retournent une valeur négative en cas d'erreur
- Les valeurs retournée correspondent aux inverses des erreurs Posix définis dans `errno.h` (ie : `-EAGAIN`).
- Référence : *errno(3)*, `asm/errno-base.h` et `asm/errno.h`
- Les fonctions renvoyant des pointeurs peuvent retourner NULL ou des valeurs spéciale en cas d'erreur
- Utiliser les macro `PTR_ERR`, `ERR_PTR` et `IS_ERR`
- Rappel : le segfault n'existe pas dans le noyau.
- Référence `linux/err.h`

Gestion des erreurs

- Vous devez systématiquement gérer les erreur retourner par les sous-fonctions même si il vous semble impossible qu'elle se produise.
- Certaines pannes matérielles peuvent amener des comportements qui paraissent impossibles. Facilité le travaille des debugueurs en détectant ce type d'erreur le plus tôt possible
- Lorsque vous gérez une erreur, vous devez *défaire* tout ce qui à déjà été fait
- Une manière connue de gérer les erreur d'initialisation dans le noyau est d'utiliser `goto`

Gérer les erreurs

```
ptr = kmalloc(sizeof(device_t));
if (!ptr) {
    ret = -ENOMEM
    goto err_alloc;
}
dev = init(&ptr);
if (dev) {
    ret = -EIO
    goto err_init;
}
return 0;

err_init:
    free(ptr);
err_alloc:
    return ret;
```

Communiquer avec le noyau

char_dev

- Le seul moya de communiquer depuis le userspace vers le noyau passe par les appel systèmes.
- Il s'agit de placer les paramètres sur la piler et de déclencher une interruption logicielle particulière (sur x86, il s'agit de l'interruption 0x80, ou `sysenter/sysexit`). Ainsi, le noyau reprend la main et effectue une action en fonction des paramètres stockés que la pile.
- Le premier paramètre correspond au numéro de l'appel système (cf. *syscalls(2)*, `asm/unistd.h`).
- Il y a environ 300 appels systèmes dans le noyau. Il est rare d'ajouter de nouveau appels système et exceptionnel d'en modifier ou d'en supprimer.
- Le modèle Unix consiste à limiter le nombre des appels systèmes et de tout gérer par l'intermédiaire de fichiers.
- Parmi les appels systèmes associés aux fichiers, nous trouvons `open`, `read`, `write`, `mmap`, `ioctl` (pour modifier la configuration du périphérique), etc...

Communiquer avec le noyau

char_dev

- Il existe dans la norme Unix des fichier spéciaux, appelés *fichiers devices*, dont les appels systèmes ne sont pas mappés sur des fichiers réels mais sur des fonctions du noyau.
- Un fichier device est associé à deux nombre, le *major* et le *minor*.
- On spécifie ces deux nombres lors de lors de la création du fichier avec la commande `mknod` :

```
target% mknod my_device c 253 0
target% ls -l my_device
```

- Il est possible de communiquer avec les fichiers device en utilisant les outils standard

```
target% cat my_device
target% echo foo > my_device
```

Communiquer avec le noyau

char_dev

- Coté noyau, un driver peut allouer une(des) entrée(s) dans la table des devices et associer des fonctions au appels système associé à cette entrée (cf. `register_chrdev`, `struct file_operations` dans `linux/fs.h`)
- En standard, tous les fichiers devices sont regroupés dans `/dev`.
- Autrefois, les numéro de devices étaient normalisés (cf `Documentation/devices.txt`) et les fichiers de `/dev` était créés statiquement
- Avec l'augmentation du nombre de périphériques existants et l'arrivée des bus hotplugs tels que l'USB, ce modèle n'était plus tenable. Il existe donc des système d'allocation dynamique de numéro de devices (cf. `/proc/device` et `/sys/dev/{char,block}`) et des systèmes créant automatiquement les fichiers nécessaires (hotplug, udev, mdev, devtmpfs, etc...)

Sysmic

Communiquer avec le noyau

char_dev

- Il existe historiquement deux types de fichiers devices :
  - block : Pour les périphériques de tailles fixes à accès aléatoire. Principalement des périphériques de stockage.
  - character : Pour les périphériques avec des données sous forme de flux comme des ports séries.
- Cette différence est aujourd'hui de moins en moins évidente
- Les cartes réseaux sont une exception notable de se modèle. Il n'existe pas de fichiers devices pour les cartes réseau. Il est nécessaire d'utiliser des appels systèmes spécifiques.
- Notons aussi que de nos jours, de nombreuses fonctionnalités sont accessibles par les file systems virtuels (`procfs`, `sysfs`, `debugfs`, etc...). Ils permettent les même fonctionnalité que les fichier devices, mais sans leur complexité.

Mémoire userspace

Les fonction de la **struct** `file_operations` recoivent en paramètre des adresses provenant de l'espace d'adressage du processus appelant. En raison de la séparation des espaces mémoire (nous y reviendrons lorsque nous parlerons plus en détail du fonctionnement de la MMU), ces adresses ne peuvent pas être utilisée directement dans le noyau. Il est est nécessaire d'utiliser les fonctions `copy_from_user` et `copy_to_user` pour y accéder.

Communiquer avec le noyau

char_dev

Nous allons faire un driver permettant de faire un pipe avec un buffer :

```
target% insmod my_chardev.ko
target% echo toto > my_dhardev
target% cat my_chardev
toto
```

Nous utiliserons :

- copy_to_user, copy_from_user
- kcalloc, kfree
- register_chrdev, unregister_chrdev_region
- memmove

Communiquer avec le noyau

char_dev

Vérifions le comportement :

```
target% insmod my_chrdev.ko
target% mknod my_chrdev c 251 0
target$ for i in {1..64}; echo "$i " > my_chrdev
target$ cat my_chrdev
target% rmmod my_chrdev
```

Les ioctl

Les appels systèmes read et write permettent de communiquer correctement avec les périphériques. Néanmoins, les périphériques possèdent très souvent des options de paramétrages. On utilise alors l'appel système `ioctl`.

- Celui-ci prend en paramètre un numéro d'identifiant et un nombre variable d'arguments :

```
ret = ioctl(fd, FIFO_GET_LEN, &arg);
```

- Il est existe une normalisation pour le format du numéro d'ioctl (`FIFO_GET_LEN`). Les macro `_IO`, `_IOR`, `_IOW` et `_IOWR` permettent de générer des numéro d'ioctl corrects :

```
#define FIFO_GET_LEN _IOR('f', 0x01, int)
```

- On Référence : `Documentation/ioctl/ioctl-decoding.txt`
- Les ioctl existants sont listés dans `Documentation/ioctl/ioctl-number.txt`