Formation à Linux Embarqué

Jérôme Pouiller <j.pouiller@sysmic.org>

Troisième partie III

Modules noyau

Configuration globale

General setup :

- *Prompt for development and/or incomplete code/drivers* : Débloque les options de compilation pour les drivers/options instables (staging, etc...)
- *Cross-compiler tool prefix* : Affecte la variable `CROSS_COMPILE`
- *Local version* : Ajoute un identifiant à la version. Indispensable dans les phases d'intégration. La version peut être lue dans `/proc/version`. Il est aussi possible de faire `make kernelrelease` dans un répertoire de compilation du noyau.
- *Automatically append version information* : Ajoute l'identifiant git à la version. Indispensable dans les phases de développement
- *Kernel compression mode* : Permet de choisir le type de compression. Chaque algorithme a ses inconvénients et ses intérêts.
- `SWAP` : Permet de gérer un espace d'échange dur un disque

Configuration globale

- `SYSVIPC` et `MQUEUE` : Communication inter-processus définis par Posix
- `IKCONFIG` : Embarque le `.config` dans le noyau
- `EXPERT` et `EMBEDDED` Débloque les options permettant principalement de réduire la taille du noyau en supprimant des modules importants
- `CC_OPTIMIZE_FOR_SIZE` : Compile avec `-Os`
- `KPROBES`, `PERF_EVENTS`, `PROFILING`, `GCOV_KERNEL` : Active les différentes instrumentations du noyau

Les périphériques de block

`MODULES` : Active la gestion des modules

`BLOCK` : Il est possible de désactiver la gestion des périphériques de block si votre système n'utilise que de la mémoire flash.

- *IO Schedulers* : Permet de choisir un ordonnanceur d'E/S différent de celui proposé en standard

System type :

- Permet de choisir le type d'architecture et de chipset
- Il est possible de désactiver certains cache lors des phases de développement
- Vous trouverez aussi dans ce menu les options relatives au jeu d'instructions accepté

Options de l'horloges

Kernel features

- `HZ` (pas sur ARM) : Définit l'intervalle de réordonnancement de l'ordonnanceur. Plus cette valeur est forte, plus l'overhead introduit par le changement de contexte est important et plus les temps de réponses des tâches sont courts
- `NO_HZ` : Permet de rendre la période de réordonnancement des tâches dynamique. Devrait permettre un léger gain de CPU (finalement négligeable avec l'ordonnanceur en $o(1)$). Permet surtout de gagner en consommation électrique.
- `HIGH_RES_TIMER` : Gère les timers avec une horloge différente de l'ordonnanceur (l'horloge est alors gérée comme un périphérique à part). Permet d'obtenir une bien meilleure précision sur les mesure de temps, à condition que votre matériel possède une horloge *HighRes*.

Options de l'ordonnanceur

- *Preemption Model* : Permet d'activer la préemption du noyau. Le pire temps réponse sont améliorés, mais le temps moyen est généralement moins bon. Un noyau préemptif stresse beaucoup plus de code. Ne pas activer si vous utilisez des drivers extérieur non garanti pour cette option.
- `RT_PREEMPT` (sur certaines architectures seulement) : Permet de threader les IRQ et ainsi de remplacer les spinlock par des mutex. Ajoute un protocole d'héritage de priorité aux mutex. Le kernel devient alors totalement préemptif. A n'utilisez que lors d'application temps réelle. Etudiez des solutions à base d'hyperviseurs.
- Ne confondez pas la préemption du noyau avec la préemption des tâches utilisateur.

Option de gestion de la mémoire

- `EABI`, `OABI`, etc... : Différentes format d'appel des fonctions. Spécifique à ARM (mais très important)
- *Memory Model* : Permet de gérer les futurs systèmes à mémoire asymétriques entre les CPU
- `COMPACTION` : Permet de compresser les page de mémoire plutôt que les mettre en swap. Particulièrement utile dans les systèmes sans swap !
- `KSM` : Permet de fusionner les page mémoire identiques. Uniquement utile avec des machines virtuelles ou des chroot. Sinon, les noyau sait que le fichier est déjà en mémoire et ne duplique pas la page

Configuration du boot et du FPE

Boot options :

- *Flattened Device Tree* : Utilise *OpenFirmware*, le nouveau format de description matériel appelé aussi *Flatten Device Tree*
- *Default kernel command string* : Permet de passer des paramètres par défaut au noyau (nous verrons cela un peu plus loin)
- *boot loader address* : Permettent de démarrer le noyau à partir d'une ROM, d'une MMC, etc...
- *Kernel Execute-In-Place from ROM* : Permet d'exécuter un noyau non compressé à partir d'une ROM

Floating point emulation : Si une instruction sur des nombres à virgule flottante est rencontrée et ne peut pas être exécutée, le noyau peut alors émuler l'instruction (voir aussi `-msoft-float`)

Configuration réseau

Networking :

- Possibilité d'activer les innombrables protocoles réseaux de niveaux 1, 2 et 3
- *Network options* : Beaucoup de fonctionnalité réseau : client dhcp, bootp, rarp, ipv6, ipsec, les protocole de routage, gestion de QoS, support des VLAN, du multicast,
- *Unix domain sockets* : Les sockets *UNIX* (cf. sortie de `netstat`)
- *TCP/IP networking* : Les sockets bien connue *TCP/IP*
- *Netfilter* : Le firewall de Linux. D'innombrable options. Permet l'utilisation d'iptables si l'option `IPTABLES` est active.

Configuration des systèmes de fichiers

File systems :

- *Second extended*, *Ext3 journalling file*, *The Extended 4 filesystem* : Le file system standard de Linux
- *FUSE* : Permet de développer des systèmes de fichiers en espace utilisateur
- *Pseudo filesystems* Systèmes de fichiers sans supports physiques
  - `TMPFS` : File system volatile en RAM. Très utilisé avec des système en flash vu que l'accès à la Flash est coûteux en temps et destructeur pour la flash
  - `SYSFS` et `PROC_FS` : Permettent au noyau d'exporter un certain nombre de donnée interne vers le userland. Beaucoup d'outils système tirent lors informations de ces systèmes de fichiers. Ils doivent être montés dans `/sys` et `/proc`. `/proc` est plutôt orienté processus alors que `/sys` est orienté modules et paramétrage du noyau.

Configuration des systèmes de fichiers

- *Miscellaneous filesystems* Contient des systèmes de fichiers spécifiques
  - *eCrypt filesystem layer* : Gestion transparent d'un file system codé
  - *Journalling Flash File System v2* : Spécialisé pour les Flash avec gestion de l'écriture uniforme, des *bad blocks* et des *erase blocks*.
  - *Compressed ROM file system* : Spécialisé pour ROM sans accès en écriture.
  - *Squashed file system* : Idem *cramfs* mais fortement compressé
- *Network File Systems*
  - *NFS client support* : File system sur ethernet. Très utilisé dans l'embarqué durant les phases de développement
  - *Root file system on NFS* : Permet de démarrer le noyau sur une partition NFS

Configuration des Drivers

Device Drivers Des centaines de drivers. Notons :

- *path to uevent helper* : Le programme apellé lorsqu'un nouveau périphérique est détecté (cf. `/proc/sys/kernel/hotplug` et `/sys/kernel/uevent_helper`)
- *Maintain a devtmpfs filesystem to mount at /dev* : Un tmpfs spécifique pour les devices automatiquement monté sur `/dev`. Les fichiers devices sont alors automatiquement créés sans l'aide d'un programme extérieur.
- *Memory Technology Device* : Les flashs
- *Staging drivers* : Des drivers en cours de bêta

Configuration du noyau

Mais aussi :

- *Kernel Hacking* : Options concernant le débugging du noyau.
- *Security Options* : Plusieurs framework permettant de gérer des droits plus fin sur les programmes exécutés et/ou de garantir l'intégrité des donnée à l'aide de TPM.
- *Cryptographic API* : Fonctions de cryptographies sélectionnées automatiquement par d'autres modules (particulièrement les protocoles réseaux)
- *Library routines* : Idem *Cryptographic API* mais avec principalement des calculs de checksum.

Les modules noyau

my_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 apeller lors du chargement du module

```
module_init
```

- Déclare la fonction à appeller lors du déchargement du modules

```
module_exit
```

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

```
MODULE_AUTHOR
```

- Description du modules

```
MODULE_DESCRIPTION
```

- Version du module

```
MODULE_VERSION
```

Les modules noyau

my_module

```
#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

Makefile :

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

Puis, on appelle :

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

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
install: modules_install
modules modules_install help clean:
	$(MAKE) -C $(KDIR) M=$(shell pwd) $@
```

et on appelle

```
host$ make ARCH=arm CROSS_COMPILE=arm-linux- KDIR
   =../linux-3.5.7/usb-a9260
```

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

```
target$ dmesg
```

- Charger/décharger un module correctement installé/indexé

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

- Mettre à jour le fichier de dépendance

```
target% depmod
```

Paramètres

Il est possible de passer des paramètres aux modules :

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

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

```
#include <linux/moduleparam.h>
module_param(param, int, 0644);
```

La paramètre doit évidement être alloué :

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

Il est fortement recommandé de documenter le paramètre

```
MODULE_PARM_DESC(param, "Display this value at
    loading and unloading");
```

/sys

Etudions `/sys`

- `/sys/module/my_module/parameters` : paramètres. Modifiable si déclaré modifiables
- `/sys/module/my_module/sections` : des info sur la zone de chargement

Quelques macros de base

- License. Indispensable

```
MODULE_LICENSE
```

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

```
EXPORT_SYMBOL
```

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

```
EXPORT_SYMBOL_GPL
```

Parlons des licenses

Le noyau est sous license GPL. Néanmoins, le débat est ouvert sur la possibilité qu'un module propriétaire puisse se linker avec. Le débat n'est pas tranché. Le noyau laisse la posisbilité 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.

Compilation avec Kbuild

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

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

`$(CONFIG_MY_OPTION)` sera remplacé par :

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

Fichier `Kconfig` :

```
config MY_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`

Les modules noyau

my_module

Quelques conseils :

- Pour mettre un code à la norme :

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

- Voir `Documentation/CodingStyle`
- Sauvez l'espace de nom du noyau : utilisez des static !
- Votre code doit être réentrant

Obtenir de la documentation

```
host$ make mandocs
host% make installmandocs
```

- Paquet linux-doc : `/usr/share/doc/linux-doc/html/kernel-api`
- Pas de paquet pour ces pages de man

Communiquer avec le noyau

char_dev

- Voie classique pour un driver pour communiquer avec le noyau
- Major, Minor

```
target% mknod my_chardev c <major> <minor>
```

- Communication en écrivant/lisant des données

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

- Possibilité de faire un peu de configuration par les ioctl

Communiquer avec le noyau

char_dev

Tentons de faire un driver qui renvois la dernière chaine recue :

```
target% insmod my_chardev.ko
target% mknod my_chardev c 249 0
target% echo toto > my_chardev
target% cat my_chardev
toto
```

Communiquer avec le noyau

char_dev

Enregistrement du *file device*

```
#include <linux/fs.h>
static int major = 0;

static int __init m_init(void) {
  major = register_chrdev(0, "m_chrdev", &m_fops);
}

static void __exit m_exit(void) {
  unregister_chrdev(major, "m_chrdev");
}
```

Communiquer avec le noyau

char_dev

`&m_fops` est une structure contenant les références des callbacks à appeller par le noyau (cf. `include/linux/fs.h`) :

```
static struct file_operations m_fops = {
  .owner        = THIS_MODULE,
  .read         = m_read,
  .write        = m_write,
};
```

Communiquer avec le noyau

char_dev

Nous devons maintenant implémenter `m_read` et `m_write`

```
static ssize_t m_read(struct file *file, char *
    user_buf, size_t count, loff_t *ppos) {
    return 0;
}

static ssize_t m_write(struct file *file, const
    char *user_buf, size_t count, loff_t *ppos) {
    return count;
}
```

Effectuons un premier test :

```
target% echo toto > my_chardev
target% cat my_chardev
```

Communiquer avec le noyau

char_dev

Ajoutons la gestion du buffer. Il est necessaire de prendre des précaution pour accéder à la mémoire du programme utilisateur.

```
static char global_buf[255];

static ssize_t m_read(struct file *file, char *
    user_buf, size_t count, loff_t *ppos) {
    if (count > idx)
        count = idx;
    copy_to_user(user_buf, global_buf, count);
    idx = 0;
    return count;
}

static ssize_t m_write(struct file *file, const
    char *user_buf, size_t count, loff_t *ppos) {
    if (count > 255)
        count = 255;
```

Communiquer avec le noyau

char_dev

Protégeons notre code contre les accès concurent :

```
#include <linux/mutex.h>
static struct mutex mutex;

static ssize_t m_read(struct file *file, char *
    user_buf, size_t count, loff_t *ppos) {
  mutex_lock(&mutex);
[...]
  mutex_unlock(&mutex);
  return count;
}

static ssize_t m_write(struct file *file, const
    char *user_buf, size_t count, loff_t *ppos) {
  mutex_lock(&mutex);
[...]
  mutex_unlock(&mutex);
```

Communiquer avec le noyau

char_dev

Ajoutons un *ioctl* :

```
#include "mod2_chr.h"
static long m_ioctl(struct file *file, unsigned int
    nr, unsigned long arg) {
  if (nr == FIFO_GET_LEN) {
    int ret = copy_to_user((void *) arg, &idx,
        sizeof(idx));
    return ret;
  }
  return -EINVAL;
}
static struct file_operations m_fops = {
[...]
  .unlocked_ioctl = m_ioctl,
};
```

Dans `mod2_chr.h` :

Communiquer avec le noyau

char_dev

Testons :

```
#include "mod2_chr.h"

int main(int argc, char **argv) {
        int ret, arg = 42;
        int fd = open(argv[1], O_RDONLY);
        ret = ioctl(fd, FIFO_GET_LEN, &arg);
        printf("Returned %m with argument %d\n", -ret
            , arg);
        return 0;
}
```