PNL - MU4IN402 : Communication entre le noyau et l'espace utilisateur

Version 18.01

Julien Sopena[1]

[1]julien.sopena@lip6.fr
Équipe REGAL - INRIA Rocquencourt
LIP6 - Université Pierre et Marie Curie

Master SAR 1ère année - 2020/2021

Grandes lignes du cours

Outline

Une multitude de mécanismes

Le noyau possède un grand nombre de mécanismes permettant la communication entre le noyau et l'espace utilisateur. Il s'en ajoute régulièrement et il est très difficile d'en supprimer.

Le choix du bon mécanisme est souvent difficile :

c'est un "troll" récurrent de la LKML

On peut les classer en plusieurs familles :

- les appels systèmes (syscall)
- les paramètres des modules
- les systèmes de fichiers en RAM (ramfs)
- les sockets
- les ioctl

Outline

Les file systems en RAM

Definition

Un ramfs est un système de fichier qui ne correspond pas un device. Ces données sont uniquement en mémoire et souvent recalculées au moment d'une lecture. Classiquement un fichier représente une valeur, mais dans certains cas ils peuvent correspondre à un ensemble de valeur.

Avantage : L'espace utilisateur y accède avec les fonctions standards `read()` et `write()`. En shell, on peut par exemple utiliser les commandes cat et echo.

Inconvénient : Ces mécanismes sont synchrones dans le sens user vers noyau, mais asynchrone dans l'autre sens.

Il existe de nombreux ramfs répondant à des besoins différents :

procfs, sysfs, configfs, debugfs, ...

Utilisation d'un ramfs

Chaque ramfs est un composant à part entière du noyau. Il faut donc les activer, avant de les monter et de les utiliser.

1. s'assurer que le noyau ai été compilé avec le ramfs choisi :

```
cat .config | grep CONFIG_DEBUG_FS CONFIG_DEBUG_FS=y
```

Utilisation d'un ramfs

Chaque ramfs est un composant à part entière du noyau. Il faut donc les activer, avant de les monter et de les utiliser.

1. s'assurer que le noyau ai été compilé avec le ramfs choisi :

```
cat .config | grep CONFIG_DEBUG_FS CONFIG_DEBUG_FS=y
```

2. monter le ramfs dans un répertoire du système

```
mount -t debugfs none /sys/kernel/debug
```

Utilisation d'un ramfs

Chaque ramfs est un composant à part entière du noyau. Il faut donc les activer, avant de les monter et de les utiliser.

1. s'assurer que le noyau ai été compilé avec le ramfs choisi :

```
cat .config | grep CONFIG_DEBUG_FS CONFIG_DEBUG_FS=y
```

2. monter le ramfs dans un répertoire du système

```
mount -t debugfs none /sys/kernel/debug
```

3. lire une valeur il suffit de lire un fichier

```
cat /sys/kernel/debug/my_module/log_limit
```

Utilisation d'un ramfs

Chaque ramfs est un composant à part entière du noyau. Il faut donc les activer, avant de les monter et de les utiliser.

1. s'assurer que le noyau ai été compilé avec le ramfs choisi :

```
cat .config | grep CONFIG_DEBUG_FS CONFIG_DEBUG_FS=y
```

2. monter le ramfs dans un répertoire du système

```
mount -t debugfs none /sys/kernel/debug
```

3. lire une valeur il suffit de lire un fichier

```
cat /sys/kernel/debug/my_module/log_limit
```

4. modifier une valeur il suffit d'écrire dans le fichier

```
echo ``1024'' > /sys/kernel/debug/my_module/log_limit
```

Outline

Le procfs

Le procfs est le plus vieux des ramfs. Monté dans le répertoire `/proc`, il était destiné à externaliser des informations sur les processus. Depuis, il a été utilisé pour accéder à de nombreuses informations du kernel. Son utilisation est aujourd'hui déconseillée pour tout ce qui ne touche pas directement aux processus.

Avantage :

- certainement le plus connu, il est largement documenté.

Inconvénient :

- peu hiérarchique, il est difficile de structurer les données.

Il existe 2 API différentes :

1. Legacy procfs API : elle est simple mais limite la taille d'un fichier à une page (`PAGE_SIZE`).
2. `seq_file` API : plus complexe elle permet de lever la limite de taille grace à une liste de buffers.

```c
static ssize_t system_state_read(struct file *file, char __user *buf,
                                 size_t count, loff_t *ppos)
{
        const char *tmp = SYSTEM_RUNNING ? "The system is : runing\n"
                        : "The system is : not runing\n";;

        return simple_read_from_buffer(buf, count, ppos, tmp, strlen(tmp));
}

static const struct file_operations system_state_fops = {
        .open = simple_open,
        .read = system_state_read,
        .llseek = noop_llseek,
};

static struct proc_dir_entry *system_state_proc_dir;

static int system_state_init(void)
{
        proc_create("my_state", 0, NULL, &system_state_fops);

        return 0;
}
module_init(system_state_init);

static void system_state_exit(void)
{
        remove_proc_entry("my_state", system_state_proc_dir);
```

Outline

sysfs

Le sysfs est le successeur du procfs. Monté dans `/sys`, il est destiné aux informations des sous-systèmes, des éléments matériels et de leurs drivers.

Aujourd'hui, il doit être le choix par défaut.

Avantage :

- il permet de gérer hiérarchiquement l'information grace à un ensemble de sous-répertoires sur plusieurs niveaux ;
- offre un ensemble de mécanismes pour libérer la mémoire et détruire récursivement les répertoires.

Inconvénients :

- plus complexe à mettre en place ;
- chaque fichier doit correspondre à une donnée unique ;
- la taille d'une donnée ne peut dépasser une page (`PAGE_SIZE`).

Les répertoires : `kobject`

La `struct kobject` est au cœur du *sysfs* :

- chaque répertoire correspond à un `kobject`
- un `module` est une abstraction d'un device ou des sub-subsytem
- un fichier du *sysfs* n'est pas un `kobject`

Les champs les plus importants sont :

- `char *name` : le répertoire sera créé sous ce nom
- `struct kobject *parent` : le `kobject` du répertoire père
- `struct kref* kref` : compteur de références pour gérer la destruction du repertoire (voir cours sur la mémoire)
- `struct kset *kset` : regroupe tous les `kobject` d'un type donné

Les fichiers : `kobj_attribute`

Chaque fichier du *sysfs* correspond à une et une seule valeur exportée et est associé à une instance de la `struct kobj_attribute`.

Les informations sur le fichier (nom et droits) sont regroupées au sein d'une sous-structure `struct attribute`.

L'association à une valeur se fait donnant les adresses de deux fonctions :

- `show` : comment remplir un buffer fourni en paramètre avec une chaîne de caractère correspondant à la valeur exportée
- `store` : comment transformer la chaîne de caractère d'un buffer fourni en paramètre en valeur à assigner à une variable

Exemple 1 : un fichier en lecture seul

```
static ssize_t system_state_show(struct kobject *kobj,
                                 struct kobj_attribute *attr,
                                 char *buf)
{
        return snprintf(buf, PAGE_SIZE, "The system is : %sruning\n",
                        system_state == SYSTEM_RUNNING ? "" : "not ");
}

static struct kobj_attribute system_state_attribute =
        __ATTR(system_state, 0400, system_state_show, NULL);

static struct kobject *my_state_kobj;
```

```c
static int __init my_state_init(void)
{
        int retval;

        my_state_kobj = kobject_create_and_add("my_state", kernel_kobj);
        if (!my_state_kobj)
                goto error_init_1;

        retval = sysfs_create_file(my_state_kobj,
                                   &system_state_attribute.attr);
        if (retval)
                goto error_init_2;
        return 0;

error_init_2:
        kobject_put(my_state_kobj);
error_init_1:
        return -ENOMEM;
}
module_init(my_state_init);

static void __exit my_state_exit(void)
{
        kobject_put(my_state_kobj);
}
module_exit(my_state_exit);
```

Exemple 2 : deux fichiers avec des écritures

```
static u32 timer = 0;

static ssize_t timer_state_show(struct kobject *kobj,
                                struct kobj_attribute *attr,
                                char *buf)
{
        return snprintf(buf, PAGE_SIZE, "The system is : %sruning\n",
                        system_state == SYSTEM_RUNNING ? "" : "not ");
}

static ssize_t timer_state_store(struct kobject *kobj,
                                struct kobj_attribute *attr,
                                char *buf, size_t count)
{
        u32 val;

        int rc = sscanf(buf, "%u",&val);
        if ((rc != 1) || (val <= 0))
                return -EINVAL;

        my_timer = val;
        return count;
}
```

```
static struct kobj_attribute system_state_attribute =
        __ATTR(system_state, 0400, system_state_show, NULL);

static struct kobj_attribute system_state_attribute =
        __ATTR(timer_state, 0600, timer_state_show, timer_state_store);

static struct attribute *attrs[] = {
        &system_state_attribute.attr,
        &timer_state_attribute.attr,
        NULL,
};

static struct attribute_group attr_group = {
        .attrs = attrs,
};

static struct kobject *my_state_kobj;
```

```c
static int __init my_state_init(void)
{
        int retval;
        my_state_kobj = kobject_create_and_add("my_state", kernel_kobj);

        if (!my_state_kobj)
                goto error_init_1;

        retval = sysfs_create_group(my_state_kobj, &attr_group);
        if (retval)
                goto error_init_2;
        return 0;

error_init_2:
        kobject_put(my_state_kobj);
error_init_1:
        return -ENOMEM;
}
module_init(my_state_init);

static void __exit my_state_exit(void)
{
        kobject_put(my_state_kobj);
}
module_exit(my_state_exit);
```

Outline

configfs

Le configfs est le pendant de sysfs : monté dans `/sys/kernel/config`, il permet de créer et détruire des objets noyau depuis l'espace utilisateur, alors que sysfs permet de manipuler des objets créés et détruits par le noyau lui même.

Avantage :

- il permet de créer des objets depuis l'espace utilisateur.

Inconvénient :

- il est plus complexe à mettre en place ;
- limite strictement chaque fichier à une valeur.

Outline

debugfs

Le debugfs a été introduit par *Greg Kroah-Hartman* pour accélérer les développements noyaux. Monté dans `/sys/kernel/debug`, il offre une API très souple et des fonctions de haut niveau extrêmement simples.

Avantage :

- très souple d'utilisation, la taille des données n'est pas limité ;
- l'API fournit des fonctions de haut niveau permettant d'exporter une valeur sous forme d'un fichier en une seule ligne de code.

Inconvénient :

- doit être réservé au débogage.

```c
static struct dentry *my_state_dir;

static int my_state_init(void)
{
        struct dentry *new_file;

        my_state_dir = debugfs_create_dir("my_state", NULL);
        if (my_state_dir == 0)
                return -ENOTDIR;

        new_file = debugfs_create_u8("system_state", 0444, my_state_dir,
                                     (u8 *) &system_state);
        if (new_file == 0){
                debugfs_remove_recursive(my_state_dir);
                return -EINVAL;
        }
        return 0;
}
module_init(my_state_init);

static void my_state_exit(void)
{
        debugfs_remove_recursive(my_state_dir);
}
module_exit(my_state_exit);
```

Outline

Communication par sockets

Avec les ramfs l'espace utilisateur n'est pas prévenu lorsqu'une valeur est modifiée par le noyau. La communication par socket permet de notifier l'espace utilisateur aussi bien que de réagir lorsque l'utilisateur envoie un message.

Avantage : c'est une communication symétrique

Inconvénient : l'implémentation applicative est plus complexe On peut utiliser plusieurs familles de socket :

- `AF_INET` : repose sur l'utilisation de socket UDP

Communication par sockets

Avec les ramfs l'espace utilisateur n'est pas prévenu lorsqu'une valeur est modifiée par le noyau. La communication par socket permet de notifier l'espace utilisateur aussi bien que de réagir lorsque l'utilisateur envoie un message.

Avantage : c'est une communication symétrique

Inconvénient : l'implémentation applicative est plus complexe On peut utiliser plusieurs familles de socket :

- `AF_INET` : repose sur l'utilisation de socket UDP
- `AF_PACKET` : permet de définir les entêtes des paquets.

Communication par sockets

Avec les ramfs l'espace utilisateur n'est pas prévenu lorsqu'une valeur est modifiée par le noyau. La communication par socket permet de notifier l'espace utilisateur aussi bien que de réagir lorsque l'utilisateur envoie un message.

Avantage : c'est une communication symétrique

Inconvénient : l'implémentation applicative est plus complexe On peut utiliser plusieurs familles de socket :

- `AF_INET` : repose sur l'utilisation de socket UDP
- `AF_PACKET` : permet de définir les entêtes des paquets.
- `AF_NETLINK` : propre à la communication entre le noyau et l'espace utilisateur, il existe de multiples implémentations.

Outline