PNL - 4I402

TP 06 – Mécanismes de communication du noyau Linux

Redha Gouicem, Maxime Lorrillere et Julien Sopena

mars 2019

Environnement : ce TP reprend l'environnement de programmation mis en place lors du troisième TP. Il suppose entre autre l'utilisation dans qemu de l'image *pnl-tp.img*, ainsi que l'existence d'un fichier personnel *myHome.img* correspondant au `/root`. Il suppose aussi que vous ayez un fichier de configuration pour le noyau Linux 4.19.3 et que vous ayez dans `/tmp` un exemplaire des sources.

Exercice 1 : Système de fichiers virtuels sysfs

Le sysfs est un système de fichiers virtuels du noyau qui permet d'exporter des objets du noyau (structures de données, variables, etc.) dans l'espace utilisateur. Ce système de fichiers est généralement monté dans `/sys`.

De nombreux sous-systèmes du noyau ont leurs propres API pour manipuler le sysfs (block, fs, power, etc.). Vous pouvez observer ces différents sous-systèmes en parcourant l'arborescence de `/sys`.

```
ls /sys
```

En fait, vous avez déjà manipulé l'une de ces API en TP : les fonctions `module_param` ont notamment pour objectif d'exporter des variables de vos modules au sein du sysfs, dans le répertoire `/sys/module/<module name>/parameters`.

L'architecture du sysfs est étroitement liée aux `struct kobject` : à chaque kobject du noyau correspond un répertoire du sysfs, les fichiers sont eux des *attributs* (`struct attribute`). Certains kobjects particuliers sont définis globalement, comme par exemple `kernel_kobj`, qui correspond au répertoire `/sys/kernel`.

Les attributs de base fournis par le sysfs (`struct attribute`) ne fournissent aucun moyen de manipuler les fichiers, cette structure est surtout utilisée comme base pour les différents sous-systèmes et permet de caractériser un attribut par son nom (le nom du fichier), par un mode d'accès.

Dans ce TP, nous utiliserons la structure `struct kobj_attribute` qui permet de définir un attribut simple avec des opérations de lecture/écriture. Cette structure embarque une `struct attribute` et défini les opérations exécutées lors de la lecture du fichier (`show`) et lors de l'écriture dans le fichier (`store`).

Le sysfs fourni plusieurs macros pour simplifier la création d'attributs, notamment :

— `__ATTR_RO` permet de créer un attribut qui ne sera accessible qu'en lecture seule ;
— `__ATTR_WO` permet de créer un attribut qui ne sera accessible qu'en écriture ;
— `__ATTR_RW` permet de créer un attribut qui sera accessible en lecture et en écriture.

En utilisant ces macros, la définition d'un attribut en lecture/écriture nommé `foo` nécessitera la définition des fonctions `foo_show` et `foo_store`.

Une fois l'attribut *défini*, sa création passe par l'utilisation de la fonction `sysfs_create_file` :

```
sysfs_create_file(struct kobject *parent, struct attribute *attr)
```

Dans ce TP, le `kobject` parent que nous utiliserons est `kernel_kobj`, ce qui placera nos fichiers dans le répertoire `/sys/kernel`.

De même, la suppression du fichier passe par la fonction `sysfs_remove_file`. Cette opération est importante et doit **impérativement** être effectuée lors du déchargement de votre module.

Question 1

Réaliser un module `hellosysfs` qui génère un fichier `/sys/kernel/hello` dont la lecture donnera la chaîne `"Hello sysfs!"`. Ce fichier contient-il vraiment cette chaîne ?

Pour tester, une fois votre module chargé, il suffit de lire ce fichier avec votre terminal.

```
# cat /sys/kernel/hello
    Hello sysfs!
```

Question 2

On souhaite maintenant pouvoir écrire une valeur dans le fichier `/sys/kernel/hello` de façon à modifier l'affichage lors de la lecture. Modifier votre module `hellosysfs` en conséquence. Par exemple :

```
# cat /sys/kernel/hello
    Hello sysfs!
# echo -n beer > /sys/kernel/hello
# cat /sys/kernel/hello
    Hello beer!
```

Exercice 2 : Initiation aux ioctl

Un *ioctl* est un appel système qui permet de communiquer directement avec le noyau. Plus particulièrement, les *ioctl* sont utilisés pour communiquer avec des pilotes de périphériques.

Du point de vue d'un utilisateur, l'appel système *ioctl* s'exécute sur un fichier ouvert, avec un numéro de requête et un paramètre de type variable. Nous vous encourageons à regarder la page de manuel correspondante (`man 2 ioctl`).

Dans le noyau, un *ioctl* n'est rien d'autre qu'une opération de la `struct file_operations` : `unlocked_ioctl`. Pour l'implémenter, une approche classique consiste à créer un pilote de périphérique qui implémente l'opération *ioctl*. En effet, tout périphérique sous Linux peut être représenté par un fichier spécial (vous pouvez en trouver dans le répertoire `/dev`). Lorsque le pilote de périphérique est créé, ajouter un nouveau périphérique au système consiste à créer un fichier spécial de type caractère ou bloc : c'est le rôle de la commande `mknod`.

Un périphérique est identifié par un numéro majeur et un nom. Vous pouvez observer la liste des pilotes de périphérique présents dans votre noyau dans le fichier `/proc/devices`. Pour créer un pilote de périphérique de type caractère, on utilise la fonction `register_chrdev` :

```
int register_chrdev(int major, const char *name, const struct file_operations *fops)
```

Lorsqu'on ne connaît pas le numéro majeur, on peut utiliser la valeur `0`, dans ce cas le noyau choisira un numéro majeur libre aléatoirement et le renverra. Pour supprimer un pilote de périphérique caractères, on utilise la fonction `unregister_chrdev`.

L'opération `unlocked_ioctl` est exécutée lorsqu'un appel système *ioctl* est effectué sur un périphérique dont le numéro majeur est celui du pilote de périphérique. Elle reçoit en paramètre le numéro de requête et un paramètre de type `unsigned long` et renvoie 0 lorsque le traitement de l'*ioctl* s'est déroulé correctement, ou un code d'erreur négatif.

Attention ! Pour communiquer plusieurs données au noyau, il est possible d'utiliser le paramètre comme un pointeur de structure. Mais dans ce cas il ne faut pas oublier que la structure se trouve dans l'espace utilisateur : il convient donc d'utiliser les fonctions `copy_from_user` ou `copy_to_user` pour manipuler les données à cette adresse.

Les numéros de requête que vous définissez sont partagés entre le noyau (votre module) et l'espace utilisateur (qui utilise l'appel système *ioctl*). Afin d'éviter des erreurs, comme envoyer un *ioctl* à un mauvais périphérique, le noyau, par convention, demande à ce que chaque numéro de requête soit unique. Pour cela, il fourni des macros spécifiques :

- `_IOR` : crée un numéro de requête en lecture
- `_IOW` : crée un numéro de requête en écriture
- `_IOWR` : crée un numéro de requête en lecture/écriture

Ces macros prennent en paramètre un *nombre magique*, un numéro de séquence, et le type de la donnée qui sera échangée entre l'espace utilisateur et le noyau. Le but ici est de générer un numéro de requête unique en composant des informations. Dans le cadre de ce TP, **nous utiliserons le nombre magique** `'N'` (oui, c'est un nombre) et des numéros de séquence commençant à 0.

La définition des numéros de requête, et le cas échéant celles des structures échangées avec le noyau, ont tout intérêt à être isolée dans un `.h` qui pourra être partagé entre le code noyau (module) et le code utilisateur (appel système *ioctl*).

Question 1

Réalisez un module `helloioctl` qui crée un nouveau pilote de périphérique de type caractère nommé `"hello"`. Pour le moment, ce périphérique n'implémente aucune opération, vous utiliserez donc une `struct file_operations` vide. Pour que le noyau Linux lui attribue un nombre majeur aléatoire, vous utiliserez comme nombre majeur la valeur 0. Vous pouvez afficher le nombre majeur choisi par le noyau dans votre fonction `init`. N'oubliez pas de supprimer votre périphérique caractère lors du déchargement de votre module.

Chargez votre module, vérifiez que votre pilote de périphérique caractères a bien été enregistré en analysant le contenu du fichier `/proc/devices`.

Question 2

On souhaite maintenant implémenter un *ioctl* qui renvoie la chaîne `"Hello ioctl!"` à l'utilisateur. Pour cela, vous allez définir un nouveau numéro de requête, en lecture seule, que vous appellerez `HELLO`. Vous pouvez utiliser le nombre magique `'N'`. Vous avez tout intérêt à définir ce numéro de requête dans un fichier `.h` séparé, que vous pourrez réutiliser par la suite.

Implémenter l'opération `unlocked_ioctl` pour votre périphérique caractère. Dans votre implémentation, vous renverez la chaîne `"Hello ioctl!"` à l'utilisateur lorsque la requête `HELLO` est utilisée, sinon votre fonction renverra `-ENOTTY`.

Pour tester votre *ioctl*, vous allez d'abord devoir créer le périphérique de type caractère à l'aide de la commande `mknod`. Par exemple, si le numéro majeur de votre pilote est 42 :

```
# mknod /dev/hello c 42 0
```

Vous devez ensuite réaliser un programme en C, qui utilisera l'appel système *ioctl* pour récupérer le message de votre pilote.

Question 3

On souhaite maintenant pouvoir modifier le contenu du message retourné par votre *ioctl*. Pour cela, ajouter un numéro de requête à votre module, en écriture, que vous appellerez `WHO`. Cette requête permettra de fournir une chaîne de caractère au module, qui la retournera lors de l'utilisation de la requête `HELLO`.

Modifiez votre implémentation de l'opération `unlocked_ioctl` en conséquence, et adapter votre programme C pour tester ce nouvel *ioctl*. Par exemple, si l'utilisateur fourni la chaîne `"beer"` à la requête `WHO`, la requête `HELLO` renverra le message `"Hello beer!"`.

Exercice 3 : Manipulation des processus dans le noyau Linux

On souhaite réaliser un module capable de surveiller l'activité CPU et mémoire d'un processus, y compris s'il a été masqué à l'aide d'un rootkit. Pour cela, nous avons besoin d'étudier les `struct pid` et `struct task_struct`.

Question 1

Analyser la `struct pid` définie dans le fichier `include/linux/pid.h`. Quel est le rôle de cette structure ?

Question 2

Les champs `->utime` et `->stime` de la `struct task_struct` enregistrent les temps CPU *user* et *system*. Quelle est l'unité de cette mesure ?

Question 3

Représentez les relations entre les structures `struct pid` et `struct task_struct`.

Exercice 4 : Monitoring de l'activité d'un processus

Dans un premier temps, nous allons créer pas à pas un simple module chargé d'afficher périodiquement les statistiques CPU d'un processus dans le *syslog*.

Conseil : tester votre module après chaque question !

Question 1

Créer un module **taskmonitor** prenant un paramètre `target` correspondant au PID du processus à surveiller. Dans ce module, ajouter une fonction `int monitor_pid(pid_t pid)` dont le rôle sera de récupérer la `struct pid` correspondant au PID passé en paramètre. Vous pourrez vous servir de la fonction `find_get_pid` pour obtenir cette structure.

Vérifier que vous arrivez à obtenir cette structure lorsque le PID recherché existe. Lorsqu'il n'existe pas, traiter correctement l'erreur.

Attention !
La fonction `find_get_pid` incrémente le compteur de références de la `struct pid`. Il est indispensable de rendre cette référence à l'aide de la fonction `put_pid` lorsque vous n'avez plus besoin de cette structure.

Question 2

Définissez une `struct task_monitor` dans votre module. Cette structure ne contient qu'un champ de type `struct pid *` et servira de descripteur pour le processus que vous allez surveiller.

Modifiez votre fonction `monitor_pid` pour qu'elle crée une `struct task_monitor` qui sera conservée jusqu'au déchargement de votre module. À quel moment devez vous rendre la référence de la `struct pid` ?

Question 3

On souhaite réaliser un `kthread` qui affichera périodiquement les temps CPU consommé par le processus.

Créez une fonction `int monitor_fn(void *unused)` qui sera exécutée par le `kthread` et qui affichera toutes les secondes si le processus monitoré est encore en vie. Pour cela, vous aurez besoin de récupérer la `struct task_struct` du processus, que vous pouvez obtenir à l'aide de la fonction `get_pid_task`. Vous pourrez également vous servir de la fonction `pid_alive` pour savoir si le processus monitoré est en vie.

S'il est en vie, affichez les temps CPU consommés par le processus. Par exemple :

```
pid 267 usr 0 sys
```

Dans la fonction d'initialisation de votre module, utilisez la fonction `kthread_run` pour créer le *kthread* qui exécute cette fonction. Vous pourrez vous inspirer des modules fournis lors du TP sur le débogage.

Attention !
La fonction `get_pid_task` incrémente le compteur de références de la `struct task_struct`. Il est indispensable de rendre cette référence à l'aide de la fonction `put_task_struct` lorsque vous n'avez plus besoin de cette structure.

Exercice 5 : Monitoring avec le sysfs

Dans cet exercice, nous allons modifier notre module de monitoring pour nous permettre de communiquer avec lui *via* le sysfs, en créant l'attribut `taskmonitor` (`/sys/kernel/taskmonitor`).

Question 1

En plus de l'affichage effectué par le `kthread` dans le syslog, on souhaite pouvoir récupérer les statistiques du processus monitoré en lisant le fichier `/sys/kernel/taskmonitor`.

Créer l'attribut `taskmonitor` en lecture seule qui donnera ces statistiques. Pour éviter de dupliquer du code, nous vous recommandons de définir une structure `struct task_sample` pour stocker ces statistiques :

```
struct task_sample {
  u64 utime;
  u64 stime;
};
```

Cette structure pourra être remplie à l'aide d'une fonction `get_sample`, qui renvoie si oui ou non le processus est toujours en vie. Cette fonction pourra être utilisée par votre thread et par votre attribut :

```
bool get_sample(struct task_monitor *tm, struct task_sample *sample)
```

Pour tester, vous pourrez exécuter les commandes suivantes :

```
# insmod ./taskmonitor.ko target=267
# cat /sys/kernel/taskmonitor
    pid 267 usr 0 sys
```

Question 2

On souhaite pouvoir suspendre l'exécution du thread de votre module sans avoir besoin de décharger le module. Pour cela, nous allons autoriser l'écriture de « commandes » dans l'attribut `taskmonitor` : lorsque l'utilisateur écrit `"stop"` dans ce fichier, le thread sera arrêté (et détruit). Lorsque l'utilisateur écrit `"start"`, un nouveau `kthread` est démarré. Lorsque le thread est arrêté, l'utilisateur peut toujours consulter les statistiques CPU en lisant le fichier `/sys/kernel/taskmonitor`.

Faites les modifications nécessaires à votre module. Vous veillerez à éviter que plusieurs `kthreads` ne s'exécutent en parallèle.

Exercice 6 : Module de monitoring avec ioctl

Cet exercice est similaire à l'exercice 5, mais en utilisant cette fois les *ioctl* pour communiquer avec votre module de monitoring. Pour cela, reprenez la version du module de monitoring que vous aviez à la fin de l'exercice 2.

Question 1

Ajoutez à votre module de monitoring un pilote de périphérique caractères qui sera utilisé pour contrôler le module.

Question 2

En plus de l'affichage effectué par le `kthread` dans le syslog, on souhaite pouvoir récupérer les statistiques du processus monitoré en utilisant un *ioctl*. Pour cela, ajoutez un numéro de requête `GET_SAMPLE` qui permet de récupérer des statistiques. Deux approches sont possibles, vous êtes encouragés à essayer les deux :

- Passer un buffer en paramètre qui sera rempli par votre module avec la chaîne de caractères à afficher.
- Passer un pointeur vers une `struct task_sample`, qui sera remplie par votre module. Dans ce cas, il est judicieux d'isoler la définition de cette structure dans un fichier `.h` dédié pour qu'elle puisse être partagée par votre module et les autres programmes.

Pour tester, modifiez un programme C qui affiche périodiquement les statistiques du processus monitoré.

Question 3

On souhaite pouvoir suspendre l'exécution du thread de votre module sans avoir besoin de décharger le module. Pour cela, nous allons ajouter deux requêtes en lecture seule à votre *ioctl* :

- `TASKMON_STOP`, qui ne prend pas de paramètre, qui arrête le thread ;
- `TASKMON_START`, qui ne prend pas de paramètre, qui lance un nouveau `kthread`.

Faites les modifications nécessaires à votre module. Vous veillerez à éviter que plusieurs `kthreads` ne s'exécutent en parallèle. Pour tester, modifiez votre programme C (ou créez en un nouveau) pour qu'il arrête/redémarre le thread en utilisant votre *ioctl*.

Question 4

On souhaite pouvoir modifier le pid monitoré par votre module sans avoir à le décharger/recharger. Pour cela, rajoutez une requête `TASKMON_SET_PID` à votre *ioctl* qui permet de donner un nouveau pid à surveiller.