Mécanisme de communication du noyau Linux

Exercice 1 : Manipulation des processus dans le noyau Linux

On souhaite réaliser un module capable de monitorer 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 structures struct pid et struct task\_struct.

***Question 1 :***

*Analysez la structure struct pid définie dans le fichier include/linux/pid.h. Quel est le rôle de cette structure.*

*Réponse :*

|  |
| --- |
| **[50](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L50) struct** [**upid**](http://lxr.free-electrons.com/ident?i=upid) **{**  **[51](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L51)**  **[52](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L52) int** [**nr**](http://lxr.free-electrons.com/ident?i=nr)**;**  **[53](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L53) struct** [**pid\_namespace**](http://lxr.free-electrons.com/ident?i=pid_namespace) **\***[**ns**](http://lxr.free-electrons.com/ident?i=ns)**;**  **[54](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L54) struct** [**hlist\_node**](http://lxr.free-electrons.com/ident?i=hlist_node) **pid\_chain;**  **[55](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L55) };**  **[56](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L56)**  **[57](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L57) struct** [**pid**](http://lxr.free-electrons.com/ident?i=pid)  **[58](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L58) {**  **[59](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L59)** [**atomic\_t**](http://lxr.free-electrons.com/ident?i=atomic_t)[**count**](http://lxr.free-electrons.com/ident?i=count)**;**  **[60](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L60) unsigned int** [**level**](http://lxr.free-electrons.com/ident?i=level)**;**  **[61](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L61) */\* lists of tasks that use this pid \*/***  **[62](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L62) struct** [**hlist\_head**](http://lxr.free-electrons.com/ident?i=hlist_head)[**tasks**](http://lxr.free-electrons.com/ident?i=tasks)**[PIDTYPE\_MAX];**  **[63](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L63) struct** [**rcu\_head**](http://lxr.free-electrons.com/ident?i=rcu_head) **rcu;**  **[64](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L64) struct** [**upid**](http://lxr.free-electrons.com/ident?i=upid) **numbers[1];**  **[65](http://lxr.free-electrons.com/source/include/linux/pid.h" \l "L65) };** |

***/\**** *What is struct pid?*

*A struct pid is the kernel's internal notion of a process identifier. It refers to individual tasks, process groups, and sessions. Whilethere are processes attached to it the struct pid lives in a hash* *table, so it and then the processes that it refers to can be found* *quickly from the numeric pid value. The attached processes may be* *quickly accessed by following pointers from struct pid.*

*Storing pid\_t values in the kernel and referring to them later has a* *problem. The process originally with that pid may have exited and the* *pid allocator wrapped, and another process could have come along* *and been assigned that pid.*

*Referring to user space processes by holding a reference to struct* *task\_struct has a problem. When the user space process exits* *the now useless task\_struct is still kept. A task\_struct plus a* *stack consumes around 10K of low kernel memory. More precisely* *this is THREAD\_SIZE + sizeof(struct task\_struct). By comparison* *a struct pid is about 64 bytes.*

*Holding a reference to struct pid solves both of these problems.* *It is small so holding a reference does not consume a lot of* *resources, and since a new struct pid is allocated when the numeric pid* *value is reused (when pids wrap around) we don't mistakenly refer to new* *processes.* ***\*/***

***Question 2 :***

*Les champs utime et stime et la structure struct task\_struct enregistrent les temps CPU user et system. Quelle est l’unité de cette mesure ?*

*Réponse :*

L’unité de cette mesure est ticks.

***Question 3 :***

*Représentez les relations entre les structures struct pid et struct task\_struct.*

*Réponse :*

* module which find the task\_struct run by a process from given pid
  + *struct task\_struct \*pid\_task(struct pid \*pid, enum pid\_type type)*
  + pid : Pointer to the struct pid of the process.
  + pid\_type: PIDTYPE\_PID,
  + PIDTYPE\_PGID,
  + PIDTYPE\_SID,
  + PIDTYPE\_MAX

Exercice 2 : 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.*

***Question 1 :***

*Créer un module taskmonitor prenant un paramètre target correspondant au PID du processus à monitorer. 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, traitez correctement l’erreur.*

***Attention !***

*La fonction find\_get\_pid incrémente le compteur de référence 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.*

*Réponse :*

|  |
| --- |
| #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/kthread.h>  MODULE\_DESCRIPTION("A exo02 kernel module");  MODULE\_AUTHOR("Jonathan - Minh-Hieu");  MODULE\_LICENSE("GPL");  int target = 0;  module\_param(target, int, S\_IRUSR | S\_IWUSR);  MODULE\_PARM\_DESC(target, "Process PID");  struct pid \* monitor\_pid(pid\_t nr)  {  struct pid \* s = find\_get\_pid(nr);    if (s == NULL)  pr\_warn("Processus introuvable !\n");  else  pr\_warn("Processus trouve !\n");  return s;  }  void free\_pid(struct pid \* s)  {  if (s != NULL)  {  put\_pid(s);  pr\_warn("Compteur libere !\n");  }  }  static int exo02\_init(void)  {  struct pid \* s;  pr\_warn("Exo02 module loaded\n");  s = monitor\_pid(target);  free\_pid(s);  return 0;}  static void exo02\_exit(void)  {  free\_pid(pid\_for\_monitor.p);  pr\_warn("Exo02 module unloaded\n");  }  module\_init(exo02\_init);  module\_exit(exo02\_exit); |

***Question 2 :***

*Définissez une structure 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 aller monitorer.*

*Modifier votre fonction monitor\_pid pour qu’elle crée une struct task\_monitor qui sera conservée jusqu’au déchargement de votre module. A quel moment devez-vous rendre la référence de la struct pid ?*

*Réponse :*

|  |
| --- |
| #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/kthread.h>  MODULE\_DESCRIPTION("A exo02 kernel module");  MODULE\_AUTHOR("Jonathan - Minh-Hieu");  MODULE\_LICENSE("GPL");  int target = 0;  struct task\_monitor {  struct pid \*p;  };  struct task\_monitor pid\_for\_monitor;  static struct task\_struct \*monitor\_thread;  module\_param(target, int, S\_IRUSR | S\_IWUSR);  MODULE\_PARM\_DESC(target, "Process PID");  struct pid \* monitor\_pid(pid\_t nr)  {  struct pid \* s = find\_get\_pid(nr);    if (s == NULL)  pr\_warn("Processus introuvable !\n");  else  pr\_warn("Processus trouve !\n");  return s;  }  void free\_pid(struct pid \* s)  {  if (s != NULL)  {  put\_pid(s);  pr\_warn("Compteur libere !\n");  }  }  static int exo02\_init(void)  {  pr\_warn("Exo02 module loaded\n");  pid\_for\_monitor.p = monitor\_pid(target);  return 0;  }  static void exo02\_exit(void)  {  if (monitor\_thread)  kthread\_stop(monitor\_thread);  free\_pid(pid\_for\_monitor.p);  pr\_warn("Exo02 module unloaded\n");  }  module\_init(exo02\_init);  module\_exit(exo02\_exit); |

***Question 3***

*On souhaite réaliser un kthread qui affichera périodiquement les temps CPU consommé par le processus. Créer 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. (utime et stime).*

*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 structure 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.*

*Réponse :*

|  |
| --- |
| #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/kthread.h>  MODULE\_DESCRIPTION("A exo02 kernel module");  MODULE\_AUTHOR("Jonathan - Minh-Hieu");  MODULE\_LICENSE("GPL");  int target = 0;  struct task\_monitor {  struct pid \*p;  };  struct task\_monitor pid\_for\_monitor;  static struct task\_struct \*monitor\_thread;  module\_param(target, int, S\_IRUSR | S\_IWUSR);  MODULE\_PARM\_DESC(target, "Process PID");  int monitor\_fn(void \*unused)  {  struct task\_struct \* task = get\_pid\_task(pid\_for\_monitor.p, PIDTYPE\_PID);    while (!kthread\_should\_stop())  {  if (pid\_alive(task) == 1)  {  pr\_info("Processus en vie !\n");  }  else  {  pr\_warn("Processus mort !\n");  break;  }    pr\_info("PID %d usr %llu sys %llu\n", pid\_nr(pid\_for\_monitor.p), task->utime, task->stime);  set\_current\_state(TASK\_INTERRUPTIBLE);  schedule\_timeout(5\*HZ);  }  return 0;  }  struct pid \* monitor\_pid(pid\_t nr)  {  struct pid \* s = find\_get\_pid(nr);    if (s == NULL)  pr\_warn("Processus introuvable !\n");  else  pr\_warn("Processus trouve !\n");  return s;  }  void free\_pid(struct pid \* s)  {  if (s != NULL)  {  put\_pid(s);  pr\_warn("Compteur libere !\n");  }  }  static int exo02\_init(void)  {  pr\_warn("Exo02 module loaded\n");  pid\_for\_monitor.p = monitor\_pid(target);  if (pid\_for\_monitor.p != NULL)  monitor\_thread = kthread\_run(monitor\_fn, NULL, "monitor\_fn");  return 0;  }  static void exo02\_exit(void)  {  if (monitor\_thread)  kthread\_stop(monitor\_thread);  free\_pid(pid\_for\_monitor.p);  pr\_warn("Exo02 module unloaded\n");  }  module\_init(exo02\_init);  module\_exit(exo02\_exit); |

*Notes :*

- kthread est un thread qui déroule une tâche en mode Kernel.

- Si une tâche est en mode kernel, normalement, par défaut, elle est en mode TASK\_UNINTERRUPTIBLE

- Si la machine n'a qu'un seul coeur et elle démarre une kthread, dans ce cas forcément, on doit attendre la terminaison de cette tâche pour retourner en mode utilisateur.

- Si la machine a deux coeurs (ou plus de deux coeurs), un coeur lance kthread, si on désire retourner en mode utilisateur au moment qu'on veut, on peut demander au deuxième coeur pour appeler la fonction kthread\_stop. Dans kthread que l'on a lancé, on peut tester si kthread devrait s'arrêter ou non par la fonction kthread\_should\_stop(), la thread s'arrête.

- Si la machine n'a qu'un seul coeur, et qu’on lance kthread par ce coeur, une solution pour retourner en mode utilisateur au moment qu'on veut est d'ajouter deux lignes dans la fonction de kthread

(1) set\_current\_state(TASK\_INTERRUPTIBLE);

(2) schedule\_timeout(5\*HZ);

(1) pour dire que, au moment de cette ligne, je peux être interruptible

(2) je m'endors dans 5 secondes, je rends la main au mode utilisateur (par exemple: je peux taper dans le terminal,... ); dans ces 5 secondes, l'utilisateur peux appeler la fonction kthread\_stop pour arrêter le kthread; si le kthread est dans 5 secondes de s'endormir, il s'arrête tout de suite, sinon, il devrait fait le dernier test kthread\_should\_stop() et s'arrêter.

Exercice 3 : Système de fichiers virtuels sysfs

*Le sysfs est un système de fichiers virtuels du noyau qui permet d’exporter des objets du noyau (structure de données, variable, 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,…). Vous pouvez observer ces différents sous-systèmes en parcourant l’arborescence de /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 kobjects : 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 utilisé 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 et écriture. Cette structure embarque une struct attribute est définit les opérations 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.*

*Ces macros attachent une structure struct attribute à un fichier virtuel. Le paramètre du macro est le nom du fichier.*

*Une fois l’attribut défini, sa création (enregistrer dans le système) passe par l’utilisation de la fonction sysfs\_create\_file :*

*int 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.*

|  |
| --- |
| **[139](http://lxr.free-electrons.com/source/include/linux/kobject.h" \l "L139) struct** [**kobj\_attribute**](http://lxr.free-electrons.com/ident?i=kobj_attribute) **{**  **[140](http://lxr.free-electrons.com/source/include/linux/kobject.h" \l "L140) struct** [**attribute**](http://lxr.free-electrons.com/ident?i=attribute)[**attr**](http://lxr.free-electrons.com/ident?i=attr)**;**  **[141](http://lxr.free-electrons.com/source/include/linux/kobject.h" \l "L141)** [**ssize\_t**](http://lxr.free-electrons.com/ident?i=ssize_t) **(\***[**show**](http://lxr.free-electrons.com/ident?i=show)**)(struct** [**kobject**](http://lxr.free-electrons.com/ident?i=kobject) **\***[**kobj**](http://lxr.free-electrons.com/ident?i=kobj)**, struct** [**kobj\_attribute**](http://lxr.free-electrons.com/ident?i=kobj_attribute) **\***[**attr**](http://lxr.free-electrons.com/ident?i=attr)**,**  **[142](http://lxr.free-electrons.com/source/include/linux/kobject.h" \l "L142) char \***[**buf**](http://lxr.free-electrons.com/ident?i=buf)**);**   * **The kernel copies automatically buffer from kernel space inside attribute show function via kobj\_attribute object given to user space and show to user when calling a user “show” function like cat.**   **[143](http://lxr.free-electrons.com/source/include/linux/kobject.h" \l "L143)** [**ssize\_t**](http://lxr.free-electrons.com/ident?i=ssize_t) **(\***[**store**](http://lxr.free-electrons.com/ident?i=store)**)(struct** [**kobject**](http://lxr.free-electrons.com/ident?i=kobject) **\***[**kobj**](http://lxr.free-electrons.com/ident?i=kobj)**, struct** [**kobj\_attribute**](http://lxr.free-electrons.com/ident?i=kobj_attribute) **\***[**attr**](http://lxr.free-electrons.com/ident?i=attr)**,**  **[144](http://lxr.free-electrons.com/source/include/linux/kobject.h" \l "L144) const char \***[**buf**](http://lxr.free-electrons.com/ident?i=buf)**,** [**size\_t**](http://lxr.free-electrons.com/ident?i=size_t)[**count**](http://lxr.free-electrons.com/ident?i=count)**);**   * **The kernel copies automatically buffer from user space to kernel space inside attribute store function via kobj\_attribute object given when calling a user “store” function like echo.**   **[145](http://lxr.free-electrons.com/source/include/linux/kobject.h" \l "L145) };** |

***Question 1 :***

*Réaliser un module hellosysfs qui génère un fichier /sys/kernel/hello dont la lecture de ce fichier donnera la chaîne « Hello sysfs ! ». Ce fichier contient-il vraiment cette chaîne ?*

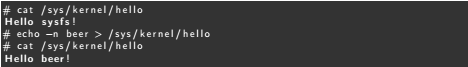
*Pour tester, une fois votre module chargé, il suffit de lire ce fichier avec votre terminal.*



*Réponse : (voir question 2)*

***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. Modifiez votre module hellosysfs en conséquence. Par exemple :*



*Réponse :*

|  |
| --- |
| #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/kthread.h>  #include <linux/kobject.h>  MODULE\_DESCRIPTION("A exo03 kernel module");  MODULE\_AUTHOR("Jonathan - Minh-Hieu");  MODULE\_LICENSE("GPL");  char exo3\_content[PAGE\_SIZE] = "Hello sysfs !";  ssize\_t my\_strncpy(char \*dest, const char \*src, ssize\_t n)  {  ssize\_t i;  for (i = 0; i < n && src[i] != '\0'; i++)  dest[i] = src[i];  for ( ; i < n; i++)  dest[i] = '\0';  return i-1;  }  static ssize\_t exo03attr\_show(struct kobject \*kobj,  struct kobj\_attribute \*attr, char \*buf)  {  return my\_strncpy(buf, exo3\_content, PAGE\_SIZE);  }  struct kobj\_attribute exo03\_attrb = \_\_ATTR\_RW(exo03attr);  static int exo03\_init(void)  {  pr\_info("Exo03 module loaded\n");  sysfs\_create\_file(kernel\_kobj, &(exo03\_attrb.attr));  return 0;  }  static void exo03\_exit(void)  {  sysfs\_remove\_file(kernel\_kobj, &(exo03\_attrb.attr));  pr\_info("Exo03 module unloaded\n");  }  module\_init(exo03\_init);  module\_exit(exo03\_exit); |

Exercice 4 : 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.*

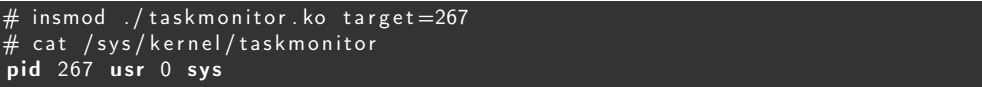
*Créer l’attribut taskmonitor en lecture seule qui donnera ces statistiques. Pour éviter de dupliquer du code, nous vous recommandons de définir une structure struct task\_sample pour stocker ces statistiques :*

|  |
| --- |
| *struct task\_sample {*  *cputime\_t utime ;*  *cputime\_t 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 :*



*Réponse :*

|  |
| --- |
| #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/kthread.h>  #include <linux/kobject.h>  MODULE\_DESCRIPTION("A exo04 kernel module");  MODULE\_AUTHOR("Jonathan - Minh-Hieu");  MODULE\_LICENSE("GPL");  int target = 0;  module\_param(target, int, S\_IRUSR | S\_IWUSR);  MODULE\_PARM\_DESC(target, "Process PID");  struct task\_sample {  cputime\_t utime;  cputime\_t stime;  };  struct task\_monitor {  struct pid \*p;  };  struct task\_monitor pid\_for\_monitor;  static struct task\_struct \*monitor\_thread;  struct task\_sample sample;  struct pid \* monitor\_pid(pid\_t nr)  {  struct pid \* s = find\_get\_pid(nr);    if (s == NULL)  pr\_warn("Processus introuvable !\n");  else  pr\_warn("Processus trouve !\n");  return s;  }  void free\_pid(struct pid \* s){  if (s != NULL){  put\_pid(s);  pr\_warn("Compteur libere !\n");  }  }  bool get\_sample (struct task\_monitor \*tm, struct task\_sample \*s){  struct task\_struct \* task = get\_pid\_task(tm->p, PIDTYPE\_PID);  s->utime = task->utime;  s->stime = task->stime;  return pid\_alive(task);  }  int monitor\_fn(void \*unused){  while (!kthread\_should\_stop()){  if (get\_sample (&pid\_for\_monitor, &sample)){  pr\_info("Processus en vie !\n");  }else{  pr\_warn("Processus mort !\n");  break;  }    pr\_info("PID %d usr %llu sys %llu\n", pid\_nr(pid\_for\_monitor.p), sample.utime, sample.stime);    set\_current\_state(TASK\_INTERRUPTIBLE);  schedule\_timeout(5\*HZ);  }  return 0;  }  ssize\_t my\_strncpy(char \*dest, const char \*src, ssize\_t n){  ssize\_t i;  for (i = 0; i < n && src[i] != '\0'; i++)  dest[i] = src[i];  for ( ; i < n; i++)  dest[i] = '\0';  return i;  }  static ssize\_t taskmonitor\_show(struct kobject \*kobj,  struct kobj\_attribute \*attr, char \*buf){  //get\_sample (&pid\_for\_monitor, &sample);  return sprintf(buf, "PID %d usr %llu sys %llu\n", pid\_nr(pid\_for\_monitor.p), sample.utime, sample.stime);  }  /\* Create a virtual file and assign it to a kob\_attribute variable. \*/  struct kobj\_attribute exo04\_attrb = \_\_ATTR\_RW(taskmonitor);  static int exo04\_init(void){  pr\_info("exo04 module loaded\n");  sysfs\_create\_file(kernel\_kobj, &(exo04\_attrb.attr));  pid\_for\_monitor.p = monitor\_pid(target);  if (pid\_for\_monitor.p != NULL)  monitor\_thread = kthread\_run(monitor\_fn, NULL, "monitor\_fn");  return 0;  }  static void exo04\_exit(void){  sysfs\_remove\_file(kernel\_kobj, &(exo04\_attrb.attr));  if (monitor\_thread)  kthread\_stop(monitor\_thread);  free\_pid(pid\_for\_monitor.p);  pr\_info("exo04 module unloaded\n");  }  module\_init(exo04\_init);  module\_exit(exo04\_exit); |

Exercice 5 : Initiation aux ioctl

Un ioctl est un appel système qui permet de communiquer directement avec le noyau. Les ioctl sont utilisés en particulier 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 structure struct file\_operation : unlocked\_ioctl. Pour l'implémenter, une approche classique consiste à créer un pilote de périphérique qui implémente l'opération unlocked\_ioctl. En effet, tout périphérique sous Linux peut être représenté par un fichier spécial, dont vous pouvez en observer des dizaines 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 , conststruct 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 qu'il est possible de convertir, généralement en pointeur. **Attention!**Lorque ce paramètre est un pointeur, il provient de l'espace utilisateur: il convient donc d'utiliser les fonctions **copy\_from\_user** ou **copy\_to\_user** pour manipuler les données à cette adresse. Enfin, **unlocked\_ioctl** renvoie 0 lorsque le traitement de l'ioctl s'est déroulé correctement, ou un code d'erreur négatif.

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 demande par convention à ce que chaque numéro de requête soit unique. Pour cela, il fournit des macros spécifique :

— \_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 utiliseront le nombre magique ’N’ (oui, c’est un nombre) et des numéros de séquence commençant à 0.

La définition des numéro de requête a 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 driver 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 un 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 driver de périphérique caractères a bien été enregistré en analysant le contenu du fichier /proc/devices.*

*Réponse:*