Systèmes d'exploitation **GLO-2001**
Patrick Dallaire H17

Travail Pratique Bonus
Modules noyau et pilote en mode caractère sous Linux

Le rapport

Le rapport consiste en un compte rendu des manipulations et les exercices demandés. Le travail est divisé en 3 sections composées parfois de plusieurs points. Pour les questions ayant rapport à des commandes sur Linux, la réponse doit comprendre la commande telle que vous l'avez entrée (tapée) suivie de la réponse du système. Si la réponse excède 10 ou 15 lignes, il faut sélectionner seulement les lignes les plus pertinentes et les inclure dans le rapport, évitant ainsi au correcteur à avoir à digérer des informations superfétatoires. Il faut également inclure les réponses aux questions et vos explications ou commentaires lorsqu'ils sont demandés.

À la fin, sauvez ou imprimez dans le format **PDF**. Dans AbiWord, vous pouvez choisir le format PDF dans "sauvegarder sous" ou dans "imprimer dans un fichier". Lors de votre remise sur PIXEL, incluez ce rapport ainsi que les codes sources, zippés dans un seul fichier. Notez que le code doit compiler sur la machine virtuelle fournie, puisque cette machine est celle du correcteur. Des erreurs de compilations entraîneront des pénalités de 50 %.

Ce travail est individuel. **Écrivez votre nom** au début du document.

Encore une fois, n'oubliez pas de joindre tous les codes sources pour les exercices de programmation.

Pour la machine virtuelle, utilisez le compte **etu2** avec le mot de passe suivant : « **etudiant** » (sans accents)

Notes

❶ Insérez dans le rapport la commande telle qu'entrée dans le shell et son résultat.

❷ Si la réponse du système est longue, sélectionnez les lignes les plus pertinentes, par exemple, quelques lignes du début et/ou de la fin.

1. Insertion d'un module dans le noyau (10 pts)

Vous devez créer un module qui affichera le message « hello world! » dans le journal du noyau. Cet exercice vous permettra de comprendre comment insérer/retirer un module dans le noyau, définir les points d'entrée/sortie d'un module et comprendre les contraintes qu'impose la programmation noyau.

Voici le code du programme **helloworld.c** :

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

static int __init(void)
{
    printk(KERN_ALERT "Hello world!\n");
    return 0;
}

static void __exit(void)
{
    printk(KERN_ALERT "See you next time!\n");
}

module_init(__init);
module_exit(__exit);
```

Les modules ne dispose pas de point d'entré par défaut tel que la fonction spéciale `main()` avec laquelle nous sommes familiers. Il faut plutôt spécifier les points entrée et sortie du programme avec des macros définies dans `linux/init.h`. La fonction `_init` sera appelée lors de l'insertion du module dans le noyau et la fonction `_exit` lors du retrait.

Comme les modules s'exécutent dans le noyau, ils n'ont pas accès à la bibliothèque standard du C. Par conséquent, la commande d'affichage à l'écran `printf` n'est pas disponible. Pour la remplacer, nous pouvons utiliser la fonction `printk` qui copie dans le journal du noyau les messages à afficher. Pour voir ces messages du noyau, tapez la commande **dmesg** dans l'invite de commande. Il est possible d'afficher les messages avec divers niveau de priorité, mais nous utiliserons `KERN_ALERT` uniquement.

Pour compiler le programme, vous utiliserez le fichier **Makefile** suivant :

```
obj-m := helloworld.o
KERNEL_DIR = /usr/src/linux-headers-2.6.32-33-generic
all:
     $(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD) modules
clean:
     rm -rf *.o *.ko *.mod.* *.symvers *.order *~
```

Pour lancer la compilation, tapez la commande **make** dans l'invite de commande. Cette opération produit un objet noyau dans le répertoire nommé **helloworld.ko** ainsi que d'autres fichiers. Cet objet peut

maintenant être inséré dans le noyau. Pour ce faire, tapez **insmod helloworld.ko** en mode superutilisateur ou ajouter **sudo** avant votre commande. Pour vérifier que le module à bel et bien été chargé dans le noyau, vous pouvez afficher la liste des modules en tapant la commande **lsmod**. Pour retirer le module du noyau, tapez **rmmod helloworld.ko** en mode superutilisateur. Ces opérations d'insertion et retrait devrait avoir affichées certains messages dans le noyau. Tapez **dmesg** pour voir ces messages.

Incluez dans le rapport i) une copie d'écran montrant que le module est chargé et ii) une copie d'écran montrant les messages retournés par le module.

2. Enregistrer un pilote en mode caractère dans le noyau (10 pts)

Sous Linux, les pilotes en mode caractère reposent sur l'abstraction du fichier, ce qui permet de les manipuler facilement. Ils sont accessibles par leur nom via le système de fichier et généralement placés dans le répertoire **/dev**. Par conséquent, un pilote se verra associé à un *i*-node et devra implémenter certaines opérations typiques des fichiers telles que l'ouverture, la fermeture, la lecture et l'écriture.

Un pilote peut être conçu pour gérer un modèle spécifique de périphérique ou possiblement une classe de périphérique partageant certaines similitudes. Pour identifier un pilote dans le noyau, nous utilisons généralement un numéro dit *majeur*. Pour identifier un périphérique particulier qu'un pilote doit gérer, nous utilisons en guise d'index un numéro dit *mineur*. Une fonction du noyau permet d'obtenir dynamiquement les identifiants nécessaires pour un pilote et ses périphériques. Voici le prototype de cette fonction :

```
int alloc_chrdev_region(dev_t *pdev_num, unsigned int firstminor,
                        unsigned int count, char *name);
```

Le pointeur `dev` est un paramètre de sortie utilisé afin de retourner les identifiants majeur et mineur alloués, `name` est le nom du pilote et la variable `count` spécifie le nombre d'identifiants mineurs demandés dont le premier est `firstminor`. Pour récupérer les identifiants de la variable `*pdev_num`, nous pouvons utiliser les macros `MAJOR()` et `MINOR()` qui retourne les numéros respectifs. Pour annuler l'enregistrement des numéros associés avec notre pilote, il faut utiliser la fonction :

```
void unregister_chrdev_region(dev_t dev_num, unsigned int count);
```

où `dev_num` contient les numéros majeur et mineur et `count` indique le nombre de mineur à retirer.

Une fois que les numéros majeur et mineur ont été réservés dans le noyau, nous pouvons définir la structure de données qui sert à représenter les pilotes en mode caractère dans le noyau. Cette structure est de type `struct cdev`. Pour initialiser une telle structure, vous pouvez appeler la fonction suivante :

```
struct cdev * cdev_alloc(void);
```

qui alloue dynamiquement la structure et retourne un pointeur sur celle-ci. Il faut ensuite spécifier le propriétaire du pilote de caractère comme suit :

```
p_vircdev->owner = THIS_MODULE;
```

Finalement, l'enregistrement du pilote dans le noyau se fait avec la fonction suivante :

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

Cette fonction nécessite simplement le pointeur sur la structure qui représentera le pilote dans le noyau ainsi que les numéros majeur et mineur réservés. Dès que cette fonction retourne, le pilote est actif et les instructions que le module implémente peuvent être appelées par le noyau. Il faut donc appeler cette fonction à la toute fin de l'initialisation du pilote. Pour retirer le pilote du noyau, il faut appeler :

```
void cdev_del(struct cdev *p);
```

Cette fonction retire le pilote du noyau, libère la mémoire utilisée par sa structure et libère aussi les numéros mineur et majeur associés au pilote qui pourront être réutilisés éventuellement.

Pour cet exercice, vous devez :

- Compléter le code du fichier reg_driver.c aux endroits spécifiés par des commentaires.
- Insérer le module dans le noyau.
- Créer le nœud du périphérique utilisant la commande **mknod** indiquée par le pilote.
- Accorder les droits d'accès complets du fichier **my_virtual_device** à tous les utilisateurs.

Incluez dans le rapport i) le code du fichier **reg_driver.c** avec vos modifications, ii) une copie d'écran de la commande **dmesg** affichant les messages d'insertion ET de retrait du module, iii) une copie d'écran du contenu du fichier **/proc/devices** montrant votre pilote et iv) une copie d'écran montrant les attributs du fichier **/dev/my_virtual_device** (avec la commande **ls –l)**.

3. Implémentation des appels systèmes (40 pts)

À la question précédente, nous avons vu comment rendre un périphérique disponible via le système de fichier. Les appels systèmes liés aux fichiers pourront donc être utilisés pour interagir avec le matériel. Ainsi, le rôle du pilote de périphérique est de répondre aux différents appels systèmes qui seront éventuellement traduits en communication avec le matériel. Pour ce travail pratique, le matériel sera représenté par une structure en mémoire noyau, essentiellement une file FIFO.

Avant d'implémenter les opérations sur les fichiers que le pilote supportera, il faut déclarer une structure spécifiant les liens entre fonctions du module et appels systèmes :

```
struct file_operations fops = {
   .owner = THIS_MODULE,
   .open = device_open,
   .release = device_release,
   .write = device_write,
   .read = device_read,
};
```

Cette structure est utilisée par le pilote de caractère pour déterminer les fonctions à appeler en fonction de l'appel système. Par conséquent, il faut la passer au pilote en question. Lors de l'initialisation du module, il est donc important de fournir l'adresse de cette structure au pilote,

```
p_vircdev->ops = &fops;
```

Votre travail consistera ensuite à implémenter dans le module les quatre fonctions `devices_` nécessaires au fonctionnement du périphérique. Le périphérique virtuel sera un *buffer circulaire* dans lequel nous pourrons lire et écrire. Le buffer circulaire est une structure de données basée sur un tableau, mais où le début et la fin sont connectés. Ainsi, lorsque l'index atteint le dernier élément de la file, incrémenter cet index nous ramène au premier élément de cette file. Cependant, la position d'un élément dans le buffer circulaire est différente de sa position réelle dans le tableau. Pour déterminer la position d'un élément, nous avons besoin de savoir où se trouve la tête du buffer circulaire dans le tableau pour ensuite déterminer la distance de l'index par rapport à la tête. Similairement à la tête, un pointeur de queue vient indiquer où se trouve la fin du buffer circulaire et permet par le fait même de calculer la taille de celui-ci.

Pour représenter le périphérique virtuel, nous utiliserons la structure suivante :

```
struct CBuffDevice {
   char data[DISK_SIZE];
   unsigned long head;
   unsigned long tail;
   unsigned long size;
   struct semaphore sem;
} virtual_device;
```

Le champ `data` constitue la mémoire du périphérique et c'est à cet endroit que les données seront lues et écrites. La taille `DISK_SIZE` est une constante devant être fixée afin d'allouer 1 Mo de mémoire. Les champs `head` et `tail` indique où débute le buffer circulaire et où il se termine dans le tableau. Le champ `size` représente la quantité de mémoire utilisées, soit la distance entre la tête et la queue. Le sémaphore sert à protéger le périphérique virtuel des accès concurrents qui pourraient corrompre les données.

Lors de l'initialisation du module ou lors de la première ouverture, vous devrez initialiser la structure du périphérique virtuel de sorte qu'elle représente un buffer circulaire vide prêt à l'emploi. Vous devez implémenter les fonctions d'ouverture et de relâchement (~fermeture) de fichier dont les prototypes sont :

```
int device_open(struct inode *inode, struct file *filp);
int device_release(struct inode *inode, struct file *filp);
```

Ces fonctions devront respectivement afficher les messages suivants :

```
my_virtual_device: opened device on inode 12116
my_virtual_device: released device on inode 12116
```

où 12116 est le numéro d'*i*-node du fichier **/dev/my_virtual_device** sur ma machine. Vous devez récupérer le numéro d' *i*-node via le pointeur sur l' *i*-node et afficher le numéro qui lui aura été assigné.

Les fonctions de lecture et d'écriture devront interagir avec le buffer circulaire. Il n'est pas nécessaire de concevoir une interface pour ce buffer. L'ajout et le retrait de valeur ainsi que la mise à jour de la tête, queue et taille peuvent très bien être fait localement dans les fonctions du pilote. Voici le prototype de la fonction d'écriture à implémenter :

```
ssize_t device_write(struct file* filp, const char* bufUserData,
                     size_t count, loff_t* f_pos);
```

Cette fonction reçoit l'adresse `bufUserData` du buffer utilisateur à copier dans le périphérique virtuelle, ainsi que la quantité de données à écrire `count`. La fonction retourne le nombre d'octets copiés avec succès sur le périphérique, un nombre qui peut être inférieur à `count` pour une copie partielle. Lorsqu'elle retourne 0, rien n'a été copié, et les valeurs négatives correspondent à une erreur. Par exemple, s'il n'y a plus d'espace sur le périphérique, il faut retourner `-ENOSPC` pour en informer l'appelant. Un message doit être affiché lors de chaque appel afin de montre l'état du périphérique suite aux changements apportés par la fonction.

```
printk(KERN_ALERT "%s: head:%lu tail:%lu size:%lu\n", DEVICE_NAME,
virtual_device.head, virtual_device.tail, virtual_device.size);
```

Ce message permet d'observer le comportement du buffer circulaire lors de vos appels. Le prototype de la fonction de lecture est le suivant :

```
ssize_t device_read(struct file* filp, char* bufUserData,
                    size_t count, loff_t* f_pos);
```

Pour cette fonction, la variable `count` spécifie le nombre d'octets à lire dans le périphérique. Tout comme pour la lecture, il faut retourner le nombre d'octets ayant été lus avec succès. La lecture avec succès des données contenues dans le périphérique virtuelle aura pour effet de les effacer du buffer circulaire. Par exemple, écrire « ABC », écrire « DEF » et lire 4 octets, retourne « ABCD », ce qui a pour effet de laisser uniquement « EF » dans le buffet. Donc lire fait avancer le pointeur de tête, ce qui rend les données inaccessibles. Écrire, à l'inverse, fait avancer le pointeur de queue.

Lorsque nous programmons dans le noyau, il faut utiliser les fonctions du noyau pour réaliser certaines opérations. Par exemple, `malloc` devient `kalloc` et `free` devient `kfree`, lorsqu'il est temps d'allouer dynamiquement la (précieuse) mémoire du noyau. Ces dernières fonctions pourraient vous être utiles afin de copier un buffer localement et le traiter ensuite. Par contre, vous allez devoir utiliser les fonctions de copie entre espace utilisateur et noyau. Ces fonctions sont :

```
unsigned long copy_from_user(void *to, const void *from, unsigned long n);
unsigned long copy_to_user(void *to, const void *from, unsigned long n);
```

et permettent de copier sécuritairement de l'information en évitant, par exemple, les défauts de pages lorsqu'une page en espace utilisateur est *swapper* sur disque. Vous devrez aussi utiliser le sémaphore du périphérique virtuel pour protéger les données qu'il contient. Pour le `up`, rien ne change. Pour ce qui est du `down`, il faut utiliser la fonction suivante :

```
int down_interruptible(struct semaphore *sem);
```

Pour tester votre pilote de périphérique, vous pouvez utiliser des fonctions permettant d'écrire dans un fichier à partir du shell tel que **echo mon texte /dev/my_virtual_device**, ou afficher le contenu d'un fichier dans le shell avec **cat /dev/my_virtual_device**. Vous pouvez aussi coder un programme utilisateur qui ouvre le fichier et le manipule avec plusieurs threads afin de vérifier les conditions de concurrences.

Assurez-vous de faire une gestion des erreurs dans votre module. Un plantage dans le noyau pourrait nécessiter le redémarrage de la machine virtuelle. Par ailleurs, ne laissez pas de données importantes sur la machine virtuelle au risque de les perdre en cas de crash majeur. Faite aussi des sauvegardes de votre code à l'extérieur de la machine virtuelle.

Incluez dans le rapport i) votre code avec commentaires dans le fichier **file_ops.c**, ii) un court texte explicatif d'environ une dizaine de lignes décrivant l'approche utilisée par vos fonctions de lecture et d'écriture (ex : copie complète du buffer, transfert caractère par caractère, transfert par morceau, etc) et iii) une copie d'écran de la commande **dmesg** après avoir fait un **echo** « bonjour » dans le fichier et un **cat** pour l'afficher.