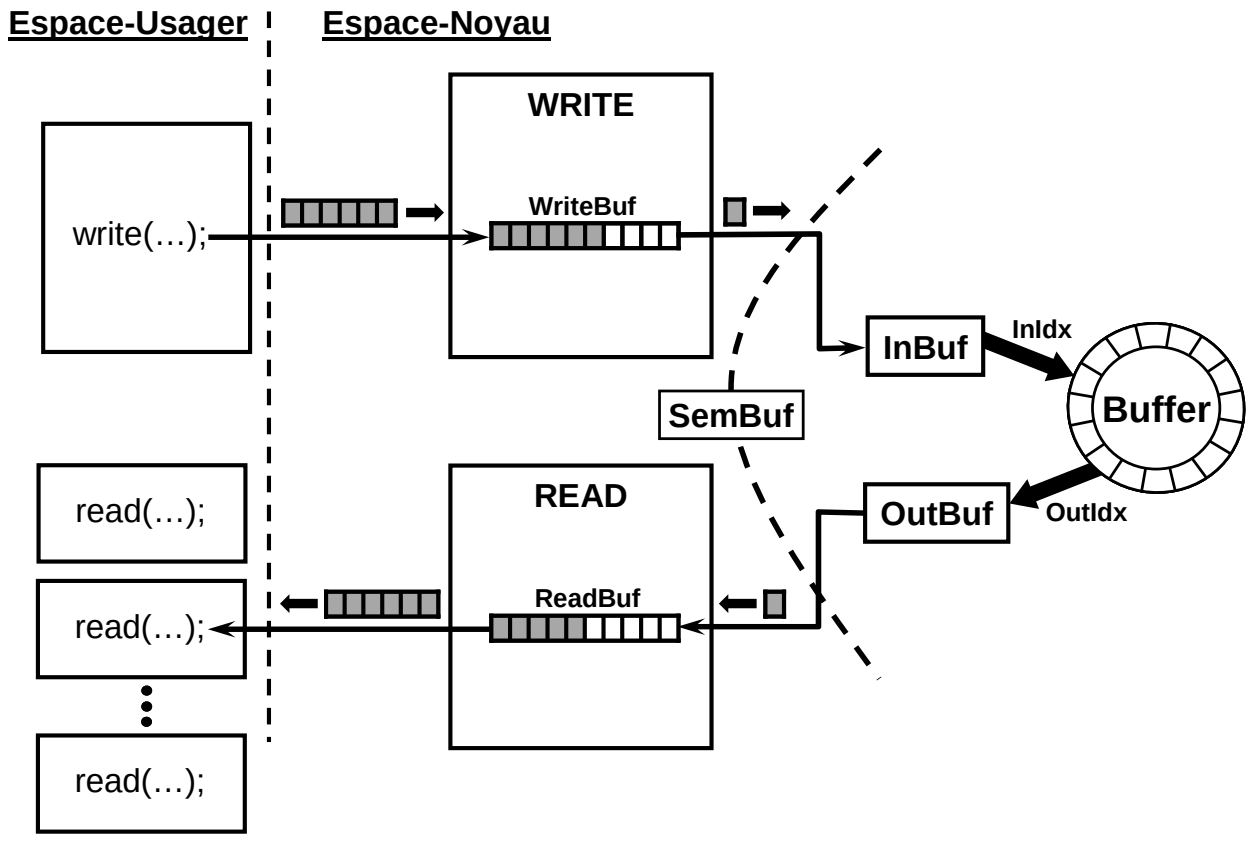
1. Objectif

L'objectif de ce laboratoire est d'initier l'étudiant à la programmation d'un pilote simple de type caractère et de l'amener à explorer et utiliser les différents mécanismes de gestion offert à l'intérieur du noyau de Linux.

2. Énoncé

2.1. Mise en contexte

On désire concevoir un pilote-caractère simple qui permet gérer une zone de mémoire, servant pour échanger des données entre des applications. La figure ci-dessous décrit grossièrement le principe de fonctionnement des lectures/écritures (Read/Write) du pilote :



La zone de mémoire (**Buffer**) est gérée selon le principe d'un tampon circulaire à l'aide de deux (2) fonctions, **InBuf** et **OutBuf**. L'accès à **Buffer**, à l'aide de **InBuf** ou **OutBuf**, doit être protégé par un sémaphore (**SemBuf**). De plus, les fonctions **InBuf** et **OutBuf** ne peuvent traiter qu'une seule donnée à la fois.

Il ne peut y avoir qu'un seul « **écrivain** » à la fois, c'est-à-dire que le **Pilote** ne peut être ouvert qu'une seule fois en « **écriture** », à n'importe quel instant. Par contre, il peut y avoir autant de « **lecteurs** » que désiré, d'un même usager ou de différents usagers. Finalement, un usager peut ouvrir le **Pilote** en « **lecture/écriture** » sauf si le **Pilote** est déjà ouvert en « **écriture** ». Dans ce dernier cas, sa requête d'ouverture lui sera refusée complètement, c'est-à-dire ni l'ouverture en **lecture** ni l'ouverture en **écriture** ne sera permise, et un code d'erreur lui sera retourné.

Les **lectures/écritures** peuvent être « **bloquantes** » ou « **non-bloquantes** », selon le choix de l'usager. Bloquantes ou non, les lectures/écritures sont effectuées au travers de « **tampons locaux** » indépendants, un pour l'écriture et un pour la lecture. Ces tampons ont une taille limitée et servent à contenir temporairement les données qui transitent entre l'application-usager et la zone de mémoire (**Buffer**).

Le code suivant vous est fourni :

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/fcntl.h>
#include <linux/wait.h>
#include <linux/spinlock.h>
#include <linux/device.h>

#include <asm/atomic.h>
#include <asm/uaccess.h>

#define READWRITE_BUFSIZE     16
#define DEFAULT_BUFSIZE       256

MODULE_LICENSE("Dual BSD/GPL")

int buf_init(void);
void buf_exit(void);
int buf_open (struct inode *inode, struct file *flip);
int buf_release (struct inode *inode, struct file *flip);
ssize_t buf_read (struct file *flip, char __user *ubuf, size_t count,
                  loff_t *f_ops);
ssize_t buf_write (struct file *flip, const char __user *ubuf, size_t count,
                   loff_t *f_ops);
long buf_ioctl (struct file *flip, unsigned int cmd, unsigned long arg);

module_init(buf_init);
module_exit(buf_exit);
```

NOTE
La liste de déclaration n'est probablement pas complète pour votre pilote. Ajouter toutes déclarations nécessaires.

```
struct BufStruct {
   unsigned int    InIdx;
   unsigned int    OutIdx;
   unsigned short  BufFull;
   unsigned short  BufEmpty;
   unsigned int    BufSize;
   unsigned short  *Buffer;
} Buffer;

struct Buf_Dev {
   unsigned short       *ReadBuf;
   unsigned short       *WriteBuf;
   struct semaphore     SemBuf;
   unsigned short       numWriter;
   unsigned short       numReader;
   dev_t                dev;
   struct cdev          cdev;
} BDev;

struct file_operations Buf_fops = {
   .owner          =     THIS_MODULE,
   .open           =     buf_open,
   .release        =     buf_release,
   .read           =     buf_read,
   .write          =     buf_write,
   .unlocked_ioctl=      buf_ioctl,
};

int BufIn (struct BufStruct *Buf, unsigned short *Data) {
   if (Buf->BufFull)
       return -1;
   Buf->BufEmpty = 0;
   Buf->Buffer[Buf->InIdx] = *Data;
   Buf->InIdx = (Buf->InIdx + 1) % Buf->BufSize;
   if (Buf->InIdx == Buf->OutIdx)
       Buf->BufFull = 1;
   return 0;
}

int BufOut (struct BufStruct *Buf, unsigned short *Data) {
   if (Buf->BufEmpty)
       return -1;
   Buf->BufFull = 0;
   *Data = Buf->Buffer[Buf->OutIdx];
   Buf->OutIdx = (Buf->OutIdx + 1) % Buf->BufSize;
   if (Buf->OutIdx == Buf->InIdx)
       Buf->BufEmpty = 1;
   return 0;
}
```

Fin du code fourni.

2.2. Description des fonctions principales du pilote

À la fin de ce laboratoire, les fonctions principales de votre pilote devront minimalement respecter les spécifications suivantes :

Init : - En plus d'initialiser le Pilote, la fonction **Init** doit initialiser au complet les deux structures de données essentielles **Buffer** et **BDev**.

- C'est-à-dire initialiser le sémaphore **SemBuf**, créer les tampons **Buffer**, **ReadBuf** et **WriteBuf**, ainsi qu'initialiser toutes les autres variables contenues dans les deux structures **Buffer** et **BDev**.

Exit : - Avant de désinstaller le Pilote, la fonction **Exit** doit s'assurer que tous les tampons ont été détruits.

Open : - Vérifie le mode d'ouverture : (voir **filp->f_flags**)
Lecture seule (**O_RDONLY**)
Écriture seule (**O_WRONLY**)
Lecture/Écriture (**O_RDWR**)

- Le Pilote ne peut être ouvert en « **écriture** » (**O_WRONLY** ou **O_RDWR**) qu'une seule fois.

- Toute nouvelle ouverture en écriture sera refusée et un code d'erreur (**ENOTTY**) sera retourné au demandeur.

Release : - Vérifie le mode d'ouverture : (**O_RDONLY**, **O_WRONLY**, **O_RDWR**).

- Si le mode d'ouverture était celui d'un « écrivain » (**O_WRONLY** ou **O_RDWR**), alors le Pilote sera de nouveau disponible pour un nouvel écrivain.

Read : - Doit supporter les lectures **bloquantes** et **non-bloquantes** (voir **filp->f_flags**).

- Doit protéger l'accès au tampon **Buffer** à l'aide du sémaphore **SemBuf**.

- Les données sont retirées une à une du **Buffer**, à l'aide de la fonction **BufOut**, et placées dans le tampon **ReadBuf**, avant d'être transmises en bloc à l'usager.

- Ce processus se répète autant de fois que nécessaire, tant qu'il y a des données disponibles et que la requête n'est pas entièrement satisfaite (nombre de données demandées).

- S'il manque de données pour satisfaire la requête (**Buffer** vide), seules les données disponibles sont retournées à l'usager.

- En **mode bloquant** : bloque si le **Buffer** est momentanément non disponible (voir **SemBuf**).

- En **mode non-bloquant** : retourne immédiatement un code d'erreur (**EAGAIN**) à l'usager, si le **Buffer** est momentanément non disponible (voir **SemBuf**).

Write : - Doit supporter les écritures **bloquantes** et **non-bloquantes** (voir **filp->f_flags**).

- Doit protéger l'accès au tampon **Buffer** à l'aide du sémaphore **SemBuf**.

- Les données sont récupérées en bloc et placées dans le tampon **WriteBuf**. Elles sont ensuite transmises une à une dans le **Buffer**, à l'aide de la fonction **BufIn**.

- Ce processus se répète autant de fois que nécessaire, tant qu'il y a de la place disponibles dans le **Buffer** et que la requête n'est pas entièrement satisfaite (nombre de données fournies).

- S'il manque de la place pour satisfaire la requête (**Buffer** plein), seules les places disponibles sont utilisées.

- En **mode bloquant** : bloque si le **Buffer** est momentanément non disponible (voir **SemBuf**).

- En **mode non-bloquant** : retourne immédiatement un code d'erreur (**EAGAIN**) à l'usager, si le **Buffer** est momentanément non disponible (voir **SemBuf**).

La fonction **IOCTL** permet d'envoyer différentes commandes à votre pilote pour soit obtenir ou configuter certains paramètres de celui-ci. Elle devra minimalement supporter les commandes suivantes :

GetNumData : - Retourne le nombre de données actuellement présentes dans le **Buffer**.

GetNumReader : - Retourne le nombre actuel de « lecteurs ».

GetBufSize : - Retourne la taille du **Buffer**.

SetBufSize : - Redimensionne le **Buffer**, tout en s'assurant que les données sont placées correctement dans le nouveau **Buffer** et en s'assurant de réajuster les index **InIdx** et **OutIdx** (voir structure **BufStruct**).

- Un code d'erreur est retourné si l'ancien **Buffer** contient trop de données pour la nouvelle taille et l'opération n'est pas effectuée.

- Retourne immédiatement un code d'erreur si le **Buffer** est momentanément non disponible (voir **SemBuf**).

- Cette commande n'est accessible que pour un usager ayant des permissions d'administrateur (**CAP_SYS_ADMIN**).

2.3. Scripts d'allocation/désallocation

Avant de pouvoir tester votre pilote à l'aide d'une application exécutée de l'espace usager, vous devez charger votre pilote et créer un nœud d'unité-matériel pour que celui-ci soit accessible par l'application. Anciennement, ce nœud était créé manuellement en invoquant, soit manuellement ou à l'aide de scripts, la commande *mknod* directement dans la ligne de commande. C'est la façon la plus élémentaire pour créer un nœud.

Cependant, il est possible de créer ce nœud directement à l'initialisation de notre pilote. Il suffit de créer une classe pour notre pilote avec *class_create()* et d'utiliser la fonction *device_create()* qui générera un « device » avec le nom qu'on lui fournit dans l'arborescence de **sysfs**. Ensuite, c'est **udev**, à l'aide de certaines règles préétablies pour le cadre du cours, qui détectera la création de nouveau « device » et s'occupera de créer le nœud dans ***/dev/***.

L'étudiant motivé en quête de connaissances approfondies sur le sujet trouvera satisfaction en se documentant d'avantage sur **sysfs** (plus particulièrement sur la création de « device » dans le « kernel space ») et en se documentant sur **udev** qui est l'application du côté « user space » qui gère les « devices » qui sont ajoutés dans ***/dev/*** et mis à la disposition de l'utilisateur.

Dans le cadre du laboratoire, je vous invite à essayer les deux méthodes, mais si vous en choisissez seulement une, ce sera celle avec *device_create()*.

2.3.1. Méthode avec mknod

Pour utiliser **mknod**, vous devrez créer deux scripts qui permettront de charger votre module et de le décharger tout en gérant les nœuds d'unités-matériel. Référez-vous aux notes de cours présentées dans les acétates du cours #3 pour avoir un exemple de comment créer ses scripts.

> *Note : Dans le cadre du laboratoire, les commandes insmod, mknod et rmmod doivent être précédées de la commande sudo. Vous devez également remplacer les fonctions chgrp et chmod par l'option –mode=0777 de mknod.*

2.3.2. Méthode avec device_create()

Pour utiliser *device_create()*, nous avons besoins de la librairie <linux/device.h>. Ensuite il faut se définir une classe qui sera utilisée pour notre « device ». Cette classe peut être définie soit dans une variable globale ou dans une structure personnelle, l'important est d'y avoir accès à l'initialisation. Il faut ensuite initialiser la classe avec la fonction *class_create()*. Une fois la classe créée, il est possible de créer le « device » avec la fonction *device_create()*.

Évidemment, il faut détruire le « device » et la classe lorsque le pilote est retiré soit dans la fonction de sortie. Les fonctions d'intérêt sont respectivement *device_destroy()* et *class_destroy()*.

Attention : il faut toujours s'assurer que la classe et le « device » ont été créés avant de poursuivre dans l'initialisation, sinon on doit défaire tout ce qui a été fait avant de quitter. Ce conseil est d'ailleurs bon pour tout ce qui doit être créé ou alloué dynamiquement. Une attention particulière sur ce point sera portée lors de la correction des laboratoires.

Voici un aperçu très simplifié de ce donne la méthode avec *device_create()* :

```
#include <linux/device.h>

[…]
struct class *my_class;
[…]

int __init buf_init(void){

[…]
my_class = class_create(THIS_MODULE,”ETSELE”);
// ETSELE peut être remplacé par n’importe quel nom de classe
[…]
device_create(my_class, NULL, BDev.dev, &BDev, “etsele_cdev”);
// Il est essentiel d’utiliser “etsele_cdev” pour les ordinateurs du
// laboratoire pour que vous puissiez avoir les droits suffisants pour
// avoir accès à votre module.
[…]

}

void __exit buf_exit(void){

[…]
device_destroy(my_class, BDev.dev);
class_destroy(myclass);
[…]

}
```

2.4. Application usager

Vous devrez développer une application usager permettant de tester exhaustivement les fonctionnalités votre pilote. Celle-ci devra être :

- Interactive;
- Permettre de changer/lire les paramètres du tampon circulaire;
- Lire/écrire dans le tampon circulaire (le contenu à mettre dans le tampon peut être une chaîne de caractères entrée par l'usager par exemple) de manière bloquante ou non;
- Supporter le démarrage de plusieurs instances d'elle-même.

Vous n'êtes pas obligé de développer une interface graphique élaborée; une simple application dans un terminal est suffisante, le but du laboratoire étant de développer un pilote et non de faire une application. Les fonctions de base à utiliser sont (voir notes de cours pour les paramètres) :

- open();
- close();
- read();
- write();
- ioctl().

Lors de la remise du laboratoire, un fichier README devra accompagner l'application pour expliquer comment votre application doit être utilisé et s'il y a des particularités avec celle-ci (e.g. problèmes connus, démarche particulière, etc.).

3. Déroulement

Ce laboratoire doit être fait en équipe de deux seulement. Vous aurez 3 séances de 2 heures pour effectuer le travail. Le laboratoire peut être abordé en 3 étapes :

- En premier lieu, développez le squelette du pilote-caractère, les scripts d'allocation/désallocation et une application simple permettant d'accéder à chacune des fonctions du pilote. Aidez-vous de la fonction printk() pour vérifier que chaque fonction s'exécute correctement;
- Ensuite, implémentez les fonctions pour la gestion du tampon circulaire tel que mentionnées dans l'énoncé. Vous pourrez ensuite mettre à jour votre application pour tester les nouvelles fonctionnalités;
- Finalement, implémentez les fonctionnalités relative aux accès bloquant/non-bloquant et à la gestion d'accès.

4. Remise

La remise du laboratoire doit être effectué électroniquement en envoyant une archive zippé au chargé de laboratoire à l'adresse indiquée par celui-ci au début du laboratoire. Le nom du fichier doit comporter le code permanent des 2 membres de l'équipe (e.g. ABCD01234567_EFGH89012345.zip). L'archive doit avoir exactement l'arborescence suivante :

- Dossier « bin »
  - Tout ce qui sert à l'exécution de votre laboratoire, c'est-à-dire module compilé, exécutable de l'application et script pour l'insertion du module.
- Dossier « src »
  - Toutes les sources de votre module et application (aucun fichier objet ou résidu de compilation). Séparez les dans 2 sous-dossiers (module et application) avec tout ce qui est nécessaire pour la compilation de chacun.

Finalement, un fichier de type README doit être fourni, détaillant le nom et l'utilité de chacun des fichiers (pour simple référence) et expliquant comment l'application fonctionne et s'il y a des particularités avec celle-ci ou votre pilote. Vous pouvez le placer à la base de votre archive.

La date de remise vous sera communiquée au laboratoire.