Programmation système – TP4

Objectifs du TP :

1. Utiliser un *character device* pour transférer des données entre noyau et applications (fin du TP3) ;
2. Rendre une zone quelconque de mémoire physique accessible en espace utilisateur ;
3. Dessiner à l'écran en écrivant directement en mémoire vidéo.

1 Création du character device dans le noyau

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

struct file_operations {
  ssize_t (*read) (struct file *, char *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
  int (*open) (struct inode *, struct file *);
  int (*release) (struct inode *, struct file *);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
};

int register_chrdev(unsigned int major, char * name,
                    struct file_operations * fops);
void unregister_chrdev(unsigned int major, char * name);
```

2 Copie de données entre espace utilisateur et noyau

Les méthodes `read` et `write` du *character device* permettent de transférer des données quelconques entre le noyau et des zones de mémoire utilisateur.

```
#include <asm/uaccess.h>
unsigned long copy_from_user(void *kto, const void * ufrom, unsigned long size);
unsigned long copy_to_user(void *uto, const void *kfrom, unsigned long size);
```

Transfert de données entre espace utilisateur (`uto` ou `ufrom`) et espace noyau (`kfrom` ou `kto`).

```
int put_user(val, void *uptr);
int get_user(val, const void *uptr);
```

C'est le type du pointeur cible en espace utilisateur (`*uptr`) qui définit la quantité de données copiées (1, 2, 4 ou 8 octets).

3 Mapping en espace utilisateur

```
#include <linux/mm.h>
int remap_page_range(struct vm_area_struct *vma, unsigned long uvaddr,
                     unsigned long paddr, unsigned long size, pgprot_t prot)
```

Cette fonction peut-être utilisée dans la méthode `mmap` du *character device*. Tous ses paramètres sont issues de la VMA sauf `paddr` qui doit être l'adresse physique de la zone à remapper. On peut ainsi rendre visible à l'espace utilisateur des pages quelconques du système.

```
int fd = open("/dev/mymodule", O_RDWR);
void *map = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
/* on peut toucher aux adresses qui suivent map */
```

4 Mapping de la mémoire vidéo

Au lieu de mapper de la mémoire normale, on peut mapper de la mémoire des périphériques. Pour cela, il faut d'abord trouver les adresses physiques correspondantes, ce qui dépend fortement de l'architecture et du périphérique. Heureusement, pour les cartes graphiques sur architecture x86, c'est facile.

`http://perso.ens-lyon.fr/brice.goglin/progsyst/tp/xmem/` : vous trouverez à cette adresse un programme utilisant les librairies XFree86 pour donner l'adresse physique à utiliser.

Il faudra inscrire cette valeur en dur dans le code source du module noyau... ou le passer en paramètre au chargement du module.

```
#include <linux/moduleparam.h>
static unsigned long iomem = 0;
module_param(iomem, ulong, 0444);

insmod mymodule.ko iomem=0x48000000
```

Le mapping utilisateur de la mémoire vidéo permet alors de dessiner directement à l'écran depuis votre programme en déréférençant les adresses virtuelles mappées !