Université Polytechnique Hauts-de-France
INSA Hauts-de-France
Année 2024 - 2025

5A ESE - O9MA332
Architecture Temps Réel Pour L'embarqué

TD n°3 : Exemples de Pilote (Driver) temps réel avec Xenomai dans l'espace noyau

Marwane AYAIDA (marwane.ayaida@uphf.fr)

Jusqu'à présent, le code que nous avons étudié s'exécutait dans l'espace utilisateur, et c'est ainsi qu'il est conseillé de développer les tâches temps réel. L'isolation mémoire entre les processus et le noyau garantit la robustesse et la sécurité du système, même en cas de bogue dans une application. Toutefois, il peut s'avérer indispensable d'écrire des portions de programme qui s'exécutent dans l'espace kernel, par exemple pour gérer efficacement les interruptions. Ce TD constitue une brève introduction à l'écriture de code pour le noyau et ne peut se substituer à une documentation plus approfondie. Nous y étudierons le fonctionnement d'un driver dans l'espace noyau Linux et Xenomai.

1 Préparation de l'environnement

Tout d'abord, il faut préparer l'environnement pour compiler un module. Pour cela, vous avez besoin d'inclure les sources que vous avez utilisé pour compiler votre noyau. Ainsi deux cas se présentent :

- soit vous avez compilé directement sur le Raspberry et dans ce cas, il faudrait juste indiquer dans le fichier *Makefile* le dossier ressources "*Linux*" utilisé.
- soit vous avez fait de la cross-compilation et dans ce cas, il faudrait re-télécharger les ressources et appliquer le patch (pas besoin de re-compiler tout le noyau).

Nous allons nous concentrer dans cette section, sur le deuxième cas. En effet, pour le premier cas, c'est assez simple il faudra juste identifier le dossier ressources dans le fichier *Makefile* utilisé.
Dans notre cas, nous allons installer les ressources dans le dossier "*/usr/src/linux*". Pour cela nous allons commencer par l'obtention des ressources.
Premièrement, il faudra récupérer le code source de Linux à partir du PC utilisé dans le TD1 pour la cross-compliation :

```
1 ~/rpi-kernel$ cd pi-kernel
2 ~/rpi-kernel$ scp linux-src.tgz pi@<ipaddress>:/tmp
```

Sinon, il faudra le re-télécharger de nouveau le code source de Linux ce qui peut prendre un certain temps sur le Raspberry Pi. Si c'est le cas, merci de revenir à l'énoncé du TD1 pour refaire les étapes de téléchargement.
Maintenant, passons sur le Raspberry Pi :

```
1 # sudo su
2 # cd /usr/src
3 /usr/src# cp /tmp/linux-src.tgz .
```

```
/usr/src# mkdir linux
/usr/src# tar xzf linux-src.tgz -C linux
/usr/src# wget https://ftp.denx.de/pub/xenomai/ipipe/v4.x/arm/ipipe-
                    core-4.19.82-arm-6.patch
/usr/src# wget wget https://ftp.denx.de/pub/xenomai/xenomai/stable/
                    xenomai-3.1.tar.bz2
/usr/src# tar xjf xenomai-3.1.tar.bz2
/usr/src# wget https://raw.githubusercontent.com/cpb-/xeno-pi/
                    7169258f33fab2b1823a75593130189efbb55979/
                    add-arm-8-a-architecture-to-xenomai-3.1.patch
/usr/src# wget https://raw.githubusercontent.com/cpb-/xeno-pi/
                    7169258f33fab2b1823a75593130189efbb55979/
                    pre-ipipe-core-4.19.82-arm-6.patch
/usr/src# wget https://raw.githubusercontent.com/cpb-/xeno-pi/
                    7169258f33fab2b1823a75593130189efbb55979/
                    post-ipipe-core-4.19.82-arm-6.patch
```

On patche le noyau :

```
/usr/src# cd xenomai-3.1/
/usr/src/xenomai-3.1# patch -p1 < ../add-arm-8-a-architecture-
        to-xenomai-3.1.patch --verbose
/usr/src/xenomai-3.1# cd ../linux
/usr/src/linux# patch -p1 < ../pre-ipipe-core-4.19.82-arm-6.patch
                    --verbose
/usr/src/linux# patch -p1 < ../ipipe-core-4.19.82-arm-6.patch
                    --verbose
/usr/src/linux# patch -p1 < ../post-ipipe-core-4.19.82-arm-6.patch
                    --verbose
/usr/src/linux# cd ..
/usr/src# xenomai-3.1/scripts/prepare-kernel.sh --linux=linux/
                    --arch=arm --ipipe=ipipe-core-4.19.82-arm-6.patch
                    --verbose
```

Copiez de l'ancienne configuration utilisée pour la cross-compilation du PC sur le Raspberry. Revenez sur le PC et tapez cette commande :

```
~/rpi-kernel$ scp linux/.config linux/Module.symvers pi@<ipaddress>:/tmp
```

Revenons sur le Raspberry et utilisons cette configuration :

```
/usr/src# cd linux
/usr/src/linux# cp /tmp/.config /tmp/Module.symvers .
```

Préparez la compilation des modules :

```
/usr/src/linux# apt install libssl-dev bison flex

/usr/src/linux# make prepare
  HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o
YACC scripts/kconfig/zconf.tab.c
LEX scripts/kconfig/zconf.lex.c
HOSTCC scripts/kconfig/zconf.tab.o
HOSTLD scripts/kconfig/conf
scripts/kconfig/conf --syncconfig Kconfig
SYSHDR arch/arm/include/generated/uapi/asm/unistd-common.h
SYSHDR arch/arm/include/generated/uapi/asm/unistd-oabi.h
SYSHDR arch/arm/include/generated/uapi/asm/unistd-eabi.h
UPD include/config/kernel.release
WRAP arch/arm/include/generated/uapi/asm/bitsperlong.h
WRAP arch/arm/include/generated/uapi/asm/bpf_perf_event.h
WRAP arch/arm/include/generated/uapi/asm/errno.h
WRAP arch/arm/include/generated/uapi/asm/ioctl.h
WRAP arch/arm/include/generated/uapi/asm/ipcbuf.h
WRAP arch/arm/include/generated/uapi/asm/msgbuf.h
WRAP arch/arm/include/generated/uapi/asm/param.h
WRAP arch/arm/include/generated/uapi/asm/poll.h
WRAP arch/arm/include/generated/uapi/asm/resource.h
WRAP arch/arm/include/generated/uapi/asm/sembuf.h
WRAP arch/arm/include/generated/uapi/asm/shmbuf.h
WRAP arch/arm/include/generated/uapi/asm/siginfo.h
WRAP arch/arm/include/generated/uapi/asm/socket.h
WRAP arch/arm/include/generated/uapi/asm/sockios.h
WRAP arch/arm/include/generated/uapi/asm/termbits.h
WRAP arch/arm/include/generated/uapi/asm/termios.h
WRAP arch/arm/include/generated/asm/compat.h
WRAP arch/arm/include/generated/asm/current.h
WRAP arch/arm/include/generated/asm/early_ioremap.h
WRAP arch/arm/include/generated/asm/emergency-restart.h
WRAP arch/arm/include/generated/asm/exec.h
WRAP arch/arm/include/generated/asm/extable.h
WRAP arch/arm/include/generated/asm/irq_regs.h
WRAP arch/arm/include/generated/asm/kdebug.h
```

```
39 WRAP arch/arm/include/generated/asm/local.h
40 WRAP arch/arm/include/generated/asm/local64.h
41 WRAP arch/arm/include/generated/asm/mm-arch-hooks.h
42 WRAP arch/arm/include/generated/asm/msi.h
43 WRAP arch/arm/include/generated/asm/parport.h
44 WRAP arch/arm/include/generated/asm/preempt.h
45 WRAP arch/arm/include/generated/asm/rwsem.h
46 WRAP arch/arm/include/generated/asm/seccomp.h
47 WRAP arch/arm/include/generated/asm/segment.h
48 WRAP arch/arm/include/generated/asm/serial.h
49 WRAP arch/arm/include/generated/asm/simd.h
50 WRAP arch/arm/include/generated/asm/sizes.h
51 WRAP arch/arm/include/generated/asm/timex.h
52 WRAP arch/arm/include/generated/asm/trace_clock.h
53 UPD include/generated/uapi/linux/version.h
54 UPD include/generated/utsrelease.h
55 SYSNR arch/arm/include/generated/asm/unistd-nr.h
56 GEN arch/arm/include/generated/asm/mach-types.h
57 SYSTBL arch/arm/include/generated/calls-oabi.S
58 SYSTBL arch/arm/include/generated/calls-eabi.S
59 CC kernel/bounds.s
60 UPD include/generated/bounds.h
61 UPD include/generated/timeconst.h
62 CC arch/arm/kernel/asm-offsets.s
63 UPD include/generated/asm-offsets.h
64 CALL scripts/checksyscalls.sh
65
66 /usr/src/linux# make scripts
67   HOSTCC scripts/dtc/dtc.o
68 HOSTCC scripts/dtc/flattree.o
69 HOSTCC scripts/dtc/fstree.o
70 HOSTCC scripts/dtc/data.o
71 HOSTCC scripts/dtc/livetree.o
72 HOSTCC scripts/dtc/treesource.o
73 HOSTCC scripts/dtc/srcpos.o
74 HOSTCC scripts/dtc/checks.o
75 HOSTCC scripts/dtc/util.o
76 LEX scripts/dtc/dtc-lexer.lex.c
77 YACC scripts/dtc/dtc-parser.tab.h
```

```
78 HOSTCC scripts/dtc/dtc-lexer.lex.o
79 YACC scripts/dtc/dtc-parser.tab.c
80 HOSTCC scripts/dtc/dtc-parser.tab.o
81 HOSTLD scripts/dtc/dtc
82 HOSTCC scripts/genksyms/genksyms.o
83 YACC scripts/genksyms/parse.tab.c
84 HOSTCC scripts/genksyms/parse.tab.o
85 LEX scripts/genksyms/lex.lex.c
86 YACC scripts/genksyms/parse.tab.h
87 HOSTCC scripts/genksyms/lex.lex.o
88 HOSTLD scripts/genksyms/genksyms
89 CC scripts/mod/empty.o
90 HOSTCC scripts/mod/mk_elfconfig
91 MKELF scripts/mod/elfconfig.h
92 HOSTCC scripts/mod/modpost.o
93 CC scripts/mod/devicetable-offsets.s
94 UPD scripts/mod/devicetable-offsets.h
95 HOSTCC scripts/mod/file2alias.o
96 HOSTCC scripts/mod/sumversion.o
97 HOSTLD scripts/mod/modpost
98 HOSTCC scripts/bin2c
99 HOSTCC scripts/kallsyms
100 HOSTCC scripts/pnmtologo
101 HOSTCC scripts/conmakehash
102 HOSTCC scripts/sortextable
103 HOSTCC scripts/asn1_compiler
104 HOSTCC scripts/extract-cert
105
106 /usr/src/linux# depmod -a
```

2 Implémentation de Pilotes (Drivers) avec Xenomai dans l'espace noyau

Maintenant, tout est prêt. Nous pouvons commencer à compiler des modules sur notre système.

2.1 Tâches dans l'espace noyau Linux

Le code que l'on développe pour le noyau Linux est généralement fourni sous forme de module externe (fichier *.ko, kernel object*), qui est inséré dynamiquement avec la commande *insmod* et éventuellement retiré par la suite avec *rmmod*. Ceci offre une certaine souplesse pour le développeur et l'administrateur du système. Le squelette général d'un module du noyau (*my-driver-01.c*) est le suivant :

```
1  #include <linux/module.h>
2
3  static int __init my_driver_01_init (void)
4  {
5          printk(KERN_INFO "%s: chargement du module\n",
6                  THIS_MODULE->name);
7          return 0;
8  }
9
10 static void __exit my_driver_01_exit (void)
11 {
12         printk(KERN_INFO "%s: retrait du module\n",
13                 THIS_MODULE->name);
14 }
15
16 module_init(my_driver_01_init);
17 module_exit(my_driver_01_exit);
18
19 MODULE_LICENSE("GPL");
```

Voici quelques explications concernant ce code :

— Un module du noyau ne peut pas inclure les fichiers d'en-tête système (comme *<stdio.h>*) car ils appartiennent à la bibliothèque *C* de l'espace utilisateur et non à l'espace kernel. Des fichiers d'en-tête spécifiques existent pour le noyau, préfixés par le chemin *<linux/>* ou *<asm/>*. L'inclusion du header *<linux/module.h>* est le minimum requis, mais d'autres nous seront utiles.

— Les macros *module_init()* et *module_exit()*, que l'on place traditionnellement en fin de fichier, indiquent les noms de deux routines, qui sont appelées respectivement au chargement avec *insmod* et au retrait avec *rmmod* du module. La première est responsable de l'initialisation du module : elle doit renvoyer une valeur nulle si cette initialisation s'est bien déroulée ou un code d'erreur négatif dans le cas contraire. La seconde se charge de libérer les éventuelles

ressources tenues par le module.

— Les directives ___*init* et ___*exit* sont spécifiques à Linux et indiquent à quel moment la fonction est utilisée, ce qui permet de la placer sur un segment de mémoire particulier. Une fois le chargement du module terminé, le noyau sait que les fonctions (et variables globales) pré-fixées par ___*init* ne seront plus utilisées et il peut ainsi libérer la mémoire qu'elles occupent. Symétriquement, si le noyau ne permet pas le retrait des modules chargés – c'est une option de compilation du kernel – les fonctions et variables précédées par ___*exit* ne sont jamais employées, et leur segment mémoire n'a pas besoin d'être chargé.

— La fonction *printk()* permet d'envoyer un message dans les traces du noyau. Elle ressemble à son homologue *printf()* de l'espace utilisateur, toutefois il est usuel de précéder le message par une constante d'une de ces valeurs *KERN_INFO*, *KERN_WARNING*, *KERN_DEBUG*, *KERN_ERR*, etc., en fonction de la gravité du message pour qu'il soit traité en conséquence par le système de journalisation des traces. Attention, ces constantes symboliques représentent des chaînes de caractères qui sont concaténées avec le message. Il ne faut donc pas ajouter de virgule entre le niveau de gravité et le message lui-même.

```
1 int printk (const char *format, ...);
```

— *THIS_MODULE* est un pointeur global qui donne accès à la structure représentant le module concerné. Ainsi, nous pouvons afficher son nom dans les traces du noyau.

— Enfin, il est nécessaire d'indiquer avec la macro *MODULE_LICENSE()* la licence d'utilisation du module. Si celle-ci n'est pas *GPL*, *GPLv2* ou quelques autres variantes compatibles avec la *GPL*, le driver est considéré comme *PROPRIETARY*. Il ne pourra pas être intégré statiquement dans le code du noyau et sera obligatoirement chargé comme module. Au chargement, il «tachera» le noyau (qui aura alors l'attribut *tainted*) jusqu'au *reboot*. Certaines fonctionnalités centrales du kernel ne lui seront pas accessibles.

La compilation d'un module nécessite de disposer des fichiers d'en-tête du noyau cible ainsi que de son fichier de configuration *.config*. Le *Makefile* qui permet de générer le module s'appuie également sur la chaîne de construction du noyau cible. Pour notre exemple ci-dessous, nous utilisons, comme dossier des ressources Linux indiquée par la variable *KERNEL_DIR*, le dossier pré-préparé avec les ressources Linux de la section précédente : */usr/src/linux*. Vous devrez le remplacer par votre propre arborescence avec l'emplacement de votre dossier, si ce dernier différent.

```
ifneq ($(KERNELRELEASE),)
        obj-m += my-driver-01.o
        obj-m += my-driver-02.o
        obj-m += rtdm-driver.o


        EXTRA_CFLAGS := -I /usr/xenomai/include/
else

        XENOCONFIG=/usr/xenomai/bin/xeno-config
        CC=$(shell $(XENOCONFIG) --cc)
        CFLAGS=$(shell $(XENOCONFIG) --skin=posix --cflags)
        LDFLAGS=$(shell $(XENOCONFIG) --skin=posix --ldflags)
        LIBDIR=$(shell $(XENOCONFIG) --skin=posix --libdir)

        CROSS_COMPILE ?=
        KERNEL_DIR ?= /usr/src/linux
        MODULE_DIR := $(shell pwd)

.PHONY: all
all:: modules executable

.PHONY: modules
modules:
        $(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(MODULE_DIR)
        CROSS_COMPILE=$(CROSS_COMPILE) modules

.PHONY: executable
executable: rtdm-user-read rtdm-user-write

XENOCONFIG=/usr/xenomai/bin/xeno-config

rtdm-user-read: rtdm-user-read.c
        $(CC) -c -o rtdm-user-read.o rtdm-user-read.c $(CFLAGS)
                $(LDFLAGS)
        /usr/xenomai/bin/wrap-link.sh -v $(CC) -o rtdm-user-read
                rtdm-user-read.o $(LDFLAGS)

rtdm-user-write: rtdm-user-write.c
```

```
40          $(CC) -c -o rtdm-user-write.o rtdm-user-write.c $(CFLAGS)
41                  $(LDFLAGS)
42          /usr/xenomai/bin/wrap-link.sh -v $(CC) -o rtdm-user-write
43                  rtdm-user-write.o $(LDFLAGS)
44
45 .PHONY: clean
46 clean::
47          rm -f *.o .*.o .*.o.* *.ko .*.ko *.mod.* .*.mod.* .*.cmd *~
48          rm -f Module.symvers Module.markers modules.order
49          rm -rf .tmp_versions
50          rm -f rtdm-user-read rtdm-user-write
51 endif
```

Compilons notre module :

```
1 # make
```

Pour pouvoir l'insérer dans notre noyau, il est évidemment nécessaire de disposer des droits *root*. Les messages envoyés dans les traces du kernel sont visibles avec la commande *dmesg*.

```
1 # insmod my-driver-01.ko
2 # dmesg | tail
3 ...
4 [ 354.358779] my_driver_01: chargement du module
5
6 # rmmod my-driver-01
7 # dmesg | tail
8 ...
9 [ 354.358779] my_driver_01: chargement du module
10 [ 398.693857] my_driver_01: retrait du module
```

2.2 Pilotes (Drivers) dans l'espace noyau Linux

Nous pouvons à présent commencer à envisager la construction d'un mini driver très simplifié, qui offre un point d'entrée dans */dev/* sous forme de fichier spécial représentant un périphérique caractère. Les données que nous écrirons dans ce fichier spécial seront stockées dans un buffer interne au noyau que nous pourrons consulter lors d'une lecture du même fichier spécial comme indiqué dans le fichier ci-dessous *my-driver-02.c* :

```
1  #include <linux/device.h>
2  #include <linux/miscdevice.h>
3  #include <linux/module.h>
4  #include <linux/mutex.h>
5  #include <linux/fs.h>
6  #include <linux/uaccess.h>
7  #include <asm/uaccess.h>
8
9  #define MY_DATA_MAX_LENGTH 4096
10 static char my_data [MY_DATA_MAX_LENGTH];
11 static int my_data_length = 0;
12 DEFINE_MUTEX(my_data_mutex);
13
14 static ssize_t my_driver_read(struct file * filp, char * __user u_buffer,
15                               size_t lg, loff_t * offset)
16 {
17         if (mutex_lock_interruptible(&my_data_mutex) != 0)
18                 return -ERESTARTSYS;
19
20         if (lg > my_data_length)
21                 lg = my_data_length;
22         if (lg > 0) {
23                 if (copy_to_user(u_buffer, my_data, lg) != 0) {
24                         mutex_unlock(&my_data_mutex);
25                         return -EFAULT;
26                 }
27                 my_data_length -= lg;
28                 if (my_data_length > 0)
29                         memmove(my_data, & my_data[lg],
30                         my_data_length);
31         }
32         mutex_unlock(&my_data_mutex);
33         return lg;
34 }
35
36 static ssize_t my_driver_write(struct file * filp, const char * u_buffer,
37                               size_t lg, loff_t * offset)
38 {
39         if (mutex_lock_interruptible(&my_data_mutex) != 0)
```

```
40                  return -ERESTARTSYS;
41
42          if (lg > (MY_DATA_MAX_LENGTH - my_data_length))
43                  lg = MY_DATA_MAX_LENGTH - my_data_length;
44
45          if (lg > 0) {
46                  if (copy_from_user(&my_data[my_data_length],
47                                          u_buffer, lg) != 0) {
48                          mutex_unlock(&my_data_mutex);
49                          return -EFAULT;
50                  }
51                  my_data_length += lg;
52          }
53          mutex_unlock(&my_data_mutex);
54          return lg;
55  }
56
57  static struct file_operations my_driver_fops = {
58          .owner = THIS_MODULE,
59          .read = my_driver_read,
60          .write = my_driver_write,
61  };
62
63  static struct miscdevice my_driver_misc = {
64          .minor = MISC_DYNAMIC_MINOR,
65          .name = THIS_MODULE->name,
66          .fops = &my_driver_fops,
67  };
68
69  static int __init my_driver_02_init (void)
70  {
71          return misc_register(&my_driver_misc);
72  }
73
74  static void __exit my_driver_02_exit (void)
75  {
76          misc_deregister(&my_driver_misc);
77  }
78
```

```
79 module_init(my_driver_02_init);
80 module_exit(my_driver_02_exit);
81 MODULE_LICENSE("GPL");
```

Pour comprendre le fonctionnement d'un module du noyau, il est conseillé de commencer à le lire par la fin en remontant vers le début. Nous voyons que la fonction *my_driver_02_init()* inscrit le driver dans la classe misc et que *my_driver_02_exit()* l'en retire. S'inscrire dans une classe de périphériques permet entre autres à notre driver d'apparaître automatiquement dans le répertoire */dev/* grâce au système de fichiers *devtmpfs* du kernel et d'outils comme *udev*, *mdev* ou *systemd*. La classe *misc* est souvent utilisée pour la mise au point de drivers expérimentaux.

```
1 int misc_register (struct miscdevice *misc);
2 void misc_deregister (struct miscdevice *misc);
```

La structure *miscdevice* contient quelques champs dont l'un est particulièrement intéressant, c'est *fops*, Il s'agit d'un pointeur sur une structure *file_operations* qui contient des pointeurs de fonctions vers les méthodes du driver. Dans notre cas, deux méthodes sont fournies : *read* et *write* qui seront appelées respectivement lorsqu'on lit le contenu du fichier de */dev/* ou lorsqu'on y écrit. Les autres méthodes – par exemple *open()* ou *close()* – non renseignées, adoptent un comportement par défaut. Nous voyons une variable globale *my_data* représentant un buffer dans lequel nous viendrons écrire les données que l'espace utilisateur nous envoie avec l'appel système *write()*, et lire les données à renvoyer lors d'un appel *read()*. Le buffer est accompagné par une variable *my_data_ length* indiquant le nombre de caractères qui s'y trouvent, et d'un mutex *my_data_mutex* pour le protéger. Ceci évite les situations de concurrence entre des appels système *read()* ou *write()* se déroulant en parallèle sur deux *CPU* différents ou s'entremêlant sur le même *CPU* avec un noyau compilé en mode préemptible.

Nous pouvons remarquer que les deux appels système verrouillent consciencieusement le mutex avant d'accéder au contenu du buffer (ou même à la variable *my_data_length*) et le libèrent ensuite. La prise du mutex se fait à l'aide de la fonction *mutex_lock_interruptible* :

- si le mutex est initialement libre, l'appel revient immédiatement après l'avoir verrouillé et renvoie zéro ;
- si le mutex est déjà bloqué au moment de *mutex_lock_interruptible()*, cette fonction endort le processus dans un sommeil interruptible. Dès que le mutex devient libre, la routine verrouille le mutex, et se termine en renvoyant zéro ;
- si le processus reçoit un signal alors qu'il est endormi en sommeil interruptible,

il est immédiatement réveillé et *mutex_lock_interruptible()* se termine en renvoyant un code non nul. Par convention, l'appel système doit se terminer le plus vite possible en renvoyant l'erreur *-ERESTARTSYS*.

— Lors d'un sommeil non interruptible (provoqué par *mutex_lock()*, par exemple) un processus ne peut pas être réveillé prématurément par un signal, même s'il s'agit de *SIGKILL* (de numéro *9*). La réception du signal se fera lorsque le sommeil prendra fin – sur libération du mutex.

```
1 int mutex_lock_interruptible (struct mutex * mtx);
2 void mutex_lock (struct mutex * mtx);
3 void mutex_unlock (struct mutex * mtx);
```

Enfin, outre la gestion du buffer qui n'est pas très compliquée, nous pouvons observer dans les méthodes *read()* et *write()*, les invocations de *copy_to_user()* et *copy_from_user()* qui permettent de copier un bloc de données entre l'espace mémoire du kernel et celui du processus appelant. Ces deux fonctions prennent en premier argument le pointeur de destination, puis celui de source et enfin le nombre d'octets à transférer. Elles renvoient le nombre d'octets qui n'ont pas pu être copiés (à cause d'une erreur de gestion mémoire dans l'espace utilisateur). La consigne, si l'une de ces fonctions renvoie une valeur non nulle, est de terminer l'appel système avec l'erreur *-EFAULT*.

```
1 long copy_to_user (void __user *dest,
2                    const void *source,
3                    unsigned long size);
4
5 long copy_from_user (void *dest,
6                       const void __user * source,
7                       unsigned long size);
```

Testons notre module :

```
1 # make
2 # insmod my-driver-02.ko
3 # ls /sys/class/misc/
4 [...] my_driver_02 [...]
5 # ls -l /dev/my*
6 crw------- 1 root root 10, 55 avril 22 00:55 /dev/my_driver_02
7 # echo Hello > /dev/my_driver_02
8 # echo World > /dev/my_driver_02
9 # cat /dev/my_driver_02
10 Hello
```

```
World
# cat /dev/my_driver_02
# rmmod my_driver_02
```

2.3 Pilotes (Drivers) RTDM dans l'espace noyau Xenomai

Avec *Xenomai* la gestion des tâches dédiées à l'espace kernel est effectuée dans le cadre d'une skin, nommée *RTDM* (*Real Time Driver Model*).
Le concept *RTDM*, présenté dans un article de Jan KISZKA en 2005, n'est pas uniquement attaché à *Xenomai*, mais c'est aujourd'hui le seul environnement qui implémente entièrement son *API*. Il sert d'interface entre la partie applicative (dans l'espace utilisateur) et le driver dans l'espace noyau, mais offre aussi des possibilités d'interface entre drivers.
Nous allons voir dans un premier exemple la manipulation des tâches avec l'API *RTDM*. En effet, cet API vous offre un ensemble de méthodes telles que :

- rtdm_task_init() : permet d'initialiser et de lancer une tâche.
- rtdm_task_destroy() : permet de demander l'arrêt d'une tâche RTDM.
- rtdm_task_should_stop() : permet de vérifier si une demande d'arrêt a été envoyée en utilisant la méthode rtdm_task_destroy().
- rtdm_task_sleep() : permet d'endormir la tâche pendant une durée spécifique.
- rtdm_task_set_period() : permet d'ajuster la période d'une tâche périodique.
- rtdm_task_wait_period() : permet de demander de dormir jusqu'à la prochaine période d'une tâche périodique.
- Etc...

Pour plus de détails, merci de vous référer à la page Wiki de Xenomai sur le service de gestion des tâches RTDM qui est accessible via ce lien :
https://xenomai.org/documentation/xenomai-3.1/html/xeno3prm/group___rtdm___task.html.

Nous allons tester sur un exemple dans le fichier *rtdm-example.c* la mise en place d'une tâche temps-réel RTDM périodique :

```
#include <rtdm/driver.h>

static int periode_us = 1000*1000;
static int ret;

```

```
6  module_param(periode_us, int, 0644);
7
8  rtdm_task_t task_desc;
9
10 void task(void *arg){
11
12         int count = 0;
13
14         while(!rtdm_task_should_stop()) {
15                 rtdm_printk(KERN_INFO "%s.%s()␣:␣count␣=␣%d\n",
16                         THIS_MODULE->name, __FUNCTION__, count++);
17
18                 rtdm_task_wait_period(NULL);
19         }
20 }
21
22 static int __init example_init(void) {
23
24         rtdm_printk(KERN_INFO "%s.%s()\n",
25                 THIS_MODULE->name, __FUNCTION__);
26
27         ret = rtdm_task_init(&task_desc, "rtdm-example", task, NULL,
28                                 30, periode_us*1000);
29
30         if (ret) {
31                 rtdm_printk(KERN_INFO "%s.%s()␣:␣error␣rtdm_task_init\n",
32                         THIS_MODULE->name, __FUNCTION__);
33         }
34
35         else rtdm_printk(KERN_INFO "%s.%s()␣:␣success␣rtdm_task_init\n",
36                 THIS_MODULE->name, __FUNCTION__);
37
38         return 0;
39 }
40
41 static void __exit example_exit(void) {
42
43         rtdm_task_destroy(&task_desc);
44
```

```
45          rtdm_printk(KERN_INFO "%s.%s()\n", THIS_MODULE->name,
46                      __FUNCTION__);
47 }
48
49 module_init(example_init);
50 module_exit(example_exit);
51 MODULE_LICENSE("GPL");
```

Maintenant, compilons et exécutons cet exemple :

```
1  # make
2
3  # cat /proc/xenomai/sched/threads
4  CPU PID CLASS  TYPE  PRI  TIMEOUT  STAT NAME
5    0   0    idle   core   -1   -          R    [ROOT/0]
6    1   0    idle   core   -1   -          R    [ROOT/1]
7    2   0    idle   core   -1   -          R    [ROOT/2]
8    3   0    idle   core   -1   -          R    [ROOT/3]
9
10 # insmod rtdm-example.ko
11
12 # cat /proc/xenomai/sched/threads
13 CPU PID CLASS  TYPE  PRI  TIMEOUT  STAT NAME
14   0   0    idle   core   -1   -          R    [ROOT/0]
15   1   0    idle   core   -1   -          R    [ROOT/1]
16   2   0    idle   core   -1   -          R    [ROOT/2]
17   3   0    idle   core   -1   -          R    [ROOT/3]
18   1  262   rt     core    30   29ms75us   D    [rtdm-example]
19
20 # dmesg | tail
21 ...
22 [ 7242.736751] rtdm_example.example_init()
23 [ 7242.737799] rtdm_example.task() : count = 0
24 [ 7242.737808] rtdm_example.example_init() : success rtdm_task_init
25 [ 7243.953570] rtdm_example.task() : count = 1
26 [ 7245.169338] rtdm_example.task() : count = 2
27 [ 7246.385103] rtdm_example.task() : count = 3
28 [ 7247.600871] rtdm_example.task() : count = 4
29 [ 7248.816639] rtdm_example.task() : count = 5
30
```

```
# rmmod rtdm-example

# dmesg | tail
...
[ 7271.916236] rtdm_example.task() : count = 24
[ 7273.132007] rtdm_example.task() : count = 25
[ 7274.347774] rtdm_example.task() : count = 26
[ 7275.563545] rtdm_example.task() : count = 27
[ 7275.841119] rtdm_example.example_exit()

# cat /proc/xenomai/sched/threads
CPU PID CLASS  TYPE  PRI  TIMEOUT  STAT NAME
  0   0    idle     core   -1   -        R    [ROOT/0]
  1   0    idle     core   -1   -        R    [ROOT/1]
  2   0    idle     core   -1   -        R    [ROOT/2]
  3   0    idle     core   -1   -        R    [ROOT/3]
```

On remarque bien ici que la tâche lancée avec *rtdm_task_init()* est bien lancée avec la période d'une seconde qui a été renseignée par défaut. À chaque fois que cette dernière est lancée, on vérifie s'il y a eu une demande d'arrêt par l'appel de la méthode *rtdm_task_destroy()* en testant le retour de la méthode *rtdm_task_should_stop()*. Sinon, on la fait dormir jusqu'à la prochaine période à l'aide de l'appel de la méthode *rtdm_task_wait_period()*.

En plus de la gestion des tâches, l'API RTDM permet principalement le développement de pilotes de périphériques gérés par RTDM. Ces derniers peuvent être de deux types :

- périphériques nommés : ceux auxquels on accède par les appels système *open()*, *close()*, r*ead()*, *write()*, *ioctl()*, etc. Ils disposent d'un espace de noms indépendant de celui de Linux (qui emploie les fichiers spéciaux de */dev*) ;
- périphériques protocoles : ces périphériques sont accessibles par *socket()* et *close()*, et implémentent la gestion de messages via *sendto()* et *recvfrom()*.

Nous allons voir un exemple dans le module *rtdm-driver.c* – similaire au module *my_driver_02.c* de la section précédente – nous permettant d'observer les mécanismes de communication entre l'espace utilisateur et l'espace noyau.

```
#include <linux/version.h>
#include <linux/device.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/cdev.h>
```

```
6  #include <linux/fs.h>
7  #include <asm/uaccess.h>
8
9  #include <rtdm/rtdm.h>
10 #include <rtdm/driver.h>
11
12 #define MY_DATA_SIZE 4096
13 static char my_data [MY_DATA_SIZE];
14 static int my_data_end = 0;
15 static rtdm_mutex_t my_data_mtx;
16
17 static int my_open_function(struct rtdm_fd *fd, int flags)
18 {
19         rtdm_printk(KERN_INFO "%s.%s()\n", THIS_MODULE->name,
20                     __FUNCTION__);
21         return 0;
22 }
23
24 static void my_close_function(struct rtdm_fd *fd)
25 {
26         rtdm_printk(KERN_INFO "%s.%s()\n", THIS_MODULE->name,
27                     __FUNCTION__);}
28
29 static int my_read_nrt_function (struct rtdm_fd *fd, void __user *buffer,
30                                  size_t lg)
31 {
32         rtdm_printk(KERN_INFO "%s.%s()\n", THIS_MODULE->name,
33                     __FUNCTION__);
34
35         rtdm_mutex_lock(&my_data_mtx);
36
37         if (lg > my_data_end)
38                 lg = my_data_end;
39
40         if (lg > 0) {
41                 if (rtdm_safe_copy_to_user(fd, buffer, my_data, lg) != 0) {
42                         rtdm_mutex_unlock(&my_data_mtx);
43                         return -EFAULT;
44                 }
```

```
45                  my_data_end -= lg;
46                  if (my_data_end > 0)
47                          memmove(my_data, &my_data[lg], my_data_end);
48          }
49
50          rtdm_printk("%s: sent %d bytes, \"%.*s\"\n", __FUNCTION__,
51                          lg, lg, buffer);
52
53          rtdm_mutex_unlock(&my_data_mtx);
54          return lg;
55  }
56
57  static int my_write_nrt_function(struct rtdm_fd *fd, const void __user
58                                  *buffer, size_t lg)
59  {
60          rtdm_printk(KERN_INFO "%s.%s()\n", THIS_MODULE->name,
61                          __FUNCTION__);
62
63          rtdm_mutex_lock(&my_data_mtx);
64
65          if (lg > (MY_DATA_SIZE - my_data_end))
66                  lg = MY_DATA_SIZE - my_data_end;
67
68          if (lg > 0) {
69                  if (rtdm_safe_copy_from_user(fd,
70                                          &my_data[my_data_end],
71                                          buffer, lg) != 0) {
72                          rtdm_mutex_unlock(&my_data_mtx);
73                          return -EFAULT;
74                  }
75                  my_data_end += lg;
76          }
77
78          rtdm_printk("==> Received from Linux %d bytes : %.*s\n",
79                          lg, lg, buffer);
80
81          rtdm_mutex_unlock(&my_data_mtx);
82          return lg;
83  }
```

```
84
85 static struct rtdm_driver my_rt_driver = {
86
87         .profile_info = RTDM_PROFILE_INFO(my_example,
88                                           RTDM_CLASS_TESTING, 1, 1),
89
90         .device_flags = RTDM_NAMED_DEVICE,
91         .device_count = 1,
92         .context_size = 0,
93
94         .ops = {
95                 .open = my_open_function,
96                 .close = my_close_function,
97                 .read_nrt = my_read_nrt_function,
98                 .write_nrt = my_write_nrt_function,
99         },
100 };
101
102 static struct rtdm_device my_rt_device = {
103
104         .driver = &my_rt_driver,
105         .label = "rtdm_driver_%d",
106 };
107
108 static int __init my_module_init (void)
109 {
110         rtdm_mutex_init(&my_data_mtx);
111         rtdm_dev_register(&my_rt_device);
112         return 0;
113 }
114
115 static void __exit my_module_exit (void)
116 {
117         rtdm_dev_unregister(&my_rt_device);
118         rtdm_mutex_destroy(&my_data_mtx);
119 }
120
121 module_init(my_module_init);
122 module_exit(my_module_exit);
```

```
123 MODULE_LICENSE("GPL");
```

Dans cet exemple, nous pouvons observer quelques éléments nouveaux par rapport aux drivers Linux. Tout d'abord, la structure *rtdm_device* intègre tout l'aspect spécifique à *RTDM* en ce qui concerne le module. Ceci permet de l'enregistrer de manière simple, comme nous le faisons avec la classe *misc*. Outre le nom du driver, la structure *rtdm_device* contient un pointeur sur une structure *rtdm_driver*.
Dans cette structure, nous voyons un champ nommé *ops* qui regroupe les méthodes – les appels système – implémentées par le driver.
Chaque méthode existe en version temps réel (invocable depuis le mode primaire de *Xenomai*) et en version non temps réel (avec le suffixe *_nrt*). L'implémentation de certaines méthodes (*open* et *close*, par exemple) est obligatoire en version non temps réel. Si l'implémentation temps réel n'est pas fournie, l'implémentation non temps réel peut être invoquée à sa place. La structure *rtdm_fd* que ces méthodes reçoivent en argument est un descripteur de fichier, c'est-à-dire une instance d'ouverture du driver par un processus de l'espace utilisateur. Depuis l'appel *open()* jusqu'au *close()* correspondant, les appels-systèmes invoqués recevront en argument la même structure *rtdm_fd*.
Nous voyons qu'il existe des objets de synchronisation spécifiques (ici les *rtdm_mutex*, mais on peut trouver des sémaphores, des verrous, des événements, etc.).
L'utilisation des *rtdm_mutex* est équivalente à celle des mutex de Linux. Nous remarquons également la présence de fonctions de copie de données depuis ou vers l'espace utilisateur, *rtdm_safe_copy_from_user()* et *rtdm_safe_copy_to_user()* qui fonctionnent comme leurs homologues du noyau Linux.
Chargeons notre module :

```
1 # make
2 # insmod rtdm-driver.ko
3 # ls -l /dev/rtdm
4 ...
5 crw-------- 1 root root 237, 0 avril 22 01:01 rtdm_driver_0
6 ...
```

Notre driver est bien chargé. Le périphérique est nommé *rtdm_driver_0* et il apparaît bien dans la liste des périphériques de la classe *RTDM*. Pour accéder à ce périphérique, il ne suffit pas de faire des *read()* ou *write()* à travers les commandes cat ou echo comme auparavant, il nous faut utiliser l'*API* spécifique de *RTDM*.
Voici un petit programme *rtdm-user-read.c* approximativement équivalent à *cat*.

```
1 #include <fcntl.h>
```

```
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  #define BUFFER_SIZE 128
8
9  int main (int argc, char * argv[])
10 {
11         int fd;
12         int i;
13         char buffer[BUFFER_SIZE];
14
15         if (argc < 2) {
16                 fprintf(stderr, "usage: %s rtdm-device\n", argv[0]);
17                 exit(EXIT_FAILURE);
18         }
19
20         fd = open(argv[1], O_RDONLY);
21         if (fd < 0) {
22                 fprintf(stderr, "%s: opening %s failed with error: %s\n",
23                                 argv[0], argv[1], strerror(-fd));
24                 exit(EXIT_FAILURE);
25         }
26
27         while ((i = read(fd, buffer, BUFFER_SIZE - 1)) > 0) {
28                 buffer[i] ='\0';
29                 printf("%s\n", buffer);
30         }
31
32         close(fd);
33         return EXIT_SUCCESS;
34 }
```

À première vue, ce programme ne se distingue pas d'une implémentation minimale de cat, les appels système employés sont tout à fait classiques. La différence apparaît lors de la phase finale de sa compilation. Grâce aux directives du fichier *Makefile*, l'édition des liens se fait en utilisant une bibliothèque spécifique de *Xenomai*, qui remplace les appels systèmes *open()*, *close()*, *read()* etc., traditionnels par leurs équivalents dans l'*API RTDM*.

Nous pouvons ajouter un petit utilitaire *rtdm-user-write.c* d'écriture en modifiant essentiellement les lignes centrales du programme :

```
1  #include <fcntl.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  int main (int argc, char * argv[])
8  {
9          int err;
10         int fd;
11         int i;
12
13         if (argc < 2) {
14                 fprintf(stderr, "usage: %s rtdm-device message...\n",
15                         argv[0]);
16                 exit(EXIT_FAILURE);
17         }
18
19         fd = open(argv[1], O_WRONLY);
20         if (fd < 0) {
21                 fprintf(stderr, "%s: opening %s failed with error: %s\n",
22                                 argv[0], argv[1], strerror(-fd));
23                 exit(EXIT_FAILURE);
24         }
25
26         for (i = 2; i < argc; i ++) {
27                 err = write(fd, argv[i], strlen(argv[i]));
28                 if (err < 0) {
29                         fprintf(stderr, "%s: write into %s failed with error: %s\n",
30                                         argv[0], argv[1], strerror(-err));
31                         exit(EXIT_FAILURE);
32                 }
33         }
34
35         close(fd);
36         exit(EXIT_SUCCESS);
37 }
```

Ce programme fonctionne bien entendu comme son homologue de l'espace Linux :

```
# ./rtdm-user-write
usage: ./rtdm-user-write rtdm-device message...

# ./rtdm-user-write /dev/rtdm/rtdm_driver_0 Hello

# dmesg | tail
...
[ 3535.935129] rtdm_driver.my_open_function()
[ 3535.935183] rtdm_driver.my_write_nrt_function()
[ 3535.935193] ==> Received from Linux 5 bytes : Hello
[ 3535.935209] rtdm_driver.my_close_function()

# ./rtdm-user-write /dev/rtdm/rtdm_driver_0 World
# dmesg | tail
...
[ 3622.648124] rtdm_driver.my_open_function()
[ 3622.648180] rtdm_driver.my_write_nrt_function()
[ 3622.648189] ==> Received from Linux 5 bytes : World
[ 3622.648204] rtdm_driver.my_close_function()

 ./rtdm-user-read /dev/rtdm/rtdm_driver_0
HelloWorld
# dmesg | tail
...
[ 3697.099610] rtdm_driver.my_open_function()
[ 3697.099666] rtdm_driver.my_read_nrt_function()
[ 3697.099676] my_read_nrt_function: sent 10 bytes, "HelloWorld"
[ 3697.099789] rtdm_driver.my_read_nrt_function()
[ 3697.099796] my_read_nrt_function: sent 0 bytes, ""
[ 3697.099811] rtdm_driver.my_close_function()

# ./rtdm-user-read /dev/rtdm/rtdm_driver_0
# dmesg | tail
...
[ 3747.802980] rtdm_driver.my_open_function()
[ 3747.803036] rtdm_driver.my_read_nrt_function()
[ 3747.803045] my_read_nrt_function: sent 0 bytes, ""
[ 3747.803061] rtdm_driver.my_close_function()
```

```
39
40 # rmmod rtdm-driver
```

3 Conclusion sur la programmation Xenomai dans l'espace noyau

Dans ce TD, nous avons observé les rudiments de la gestion des modules dans le noyau, que ce soit sous Linux standard ou en utilisant l'*API RTDM* pour *Xenomai*. La mise au point d'un driver pour Linux n'est pas très compliquée, mais nécessite une bonne rigueur d'écriture et une certaine connaissance de l'*API* interne du noyau. L'emploi de *RTDM* permet d'utiliser une interface plus standardisée, plus uniforme et mieux documentée. Nous n'avons fait qu'effleurer le contenu de *RTDM*, cette *API* offre des fonctionnalités de gestion des timers, des tâches, des outils de synchronisation, etc., que l'on retrouvera dans la documentation disponible à l'adresse suivante : http ://www.xenomai.org. Vous en aurez certainement besoin de quelques unes pour votre Projet.