Université Polytechnique Hauts-de-France

INSA Hauts-de-France

Année 2024 - 2025

5A ESE - O9MA332

Architecture Temps Réel Pour L'embarqué

TD n°4 : Communications inter-processus entre le domaine secondaire (Linux) et le domaine primaire (Xenomai)

Marwane AYAIDA (marwane.ayaida@uphf.fr)

Dans les deux derniers TDs, nous avons vu des exemples d'implémentation de tâches temps-réel qui peuvent être définies depuis l'espace utilisateur et le noyau. Comme nous l'avons spécifié auparavant, les tâches temps-réel doivent se limiter aux fonctionnalités temps-réel et doivent rapidement donner la main à la tâche Linux afin d'effectuer le travail non temps-réel (IHM, sauvegarde des données, envoi réseau, etc.). Pour cette raison, Xenomai propose un ensemble de fonctionnalités qui permettent d'échanger entre les tâches non temps-réel dans le domaine secondaire (Linux) et les tâches temps-réel dans le domaine primaire (Xenomai). Dans ce TD, nous allons explorer quelques possibilités d'échange de données. Il est à noter que les méthodes traitées dans ce TD ne sont pas exhaustives et qu'il existe d'autres méthodes non abordées dans ce TD.

1 Communications dans l'espace utilisateur

Dans cette section, nous allons voir deux exemples de moyens de communication entre deux tâches, une dans le domaine primaire et l'autre dans le domaine secondaire, toutes les deux définies dans l'espace utilisateur.

1.1 Utilisation des PIPEs

Comme vu en TD, dans l'espace utilisateur, il est possible d'utiliser les *PIPEs* pour échanger entre les tâches Linux et Xenomai. Un *PIPE* est un tube qui permet la communication entre tâches multi-domaines.

Les principales fonctions d'utilisation d'un *PIPE* du côté Xenomai sont :

- rt_pipe_create() : permet de créer une file de messages de type *PIPE*. Cette fonction ne doit être appelée que depuis le mode primaire.
- rt_pipe_delete() : permet de supprimer une file de messages de type *PIPE*.
- rt_pipe_read() : permet de lire un message de type *PIPE*.
- rt_pipe_write() : permet d'écrire un message de type *PIPE*.

Les principales fonctions d'utilisation d'un *PIPE* du côté Linux sont :

- open() : permet d'ouvrir le pseudo-fichier géré par le *PIPE*.
- close() : permet de fermer le pseudo-fichier géré par le *PIPE*.
- read() : permet de lire un message de type *PIPE*.
- write() : permet d'écrire un message de type *PIPE*.

Pour plus de détails sur les différentes fonctions d'utilisation des *PIPEs*, merci de consulter ce lien.

Reprenons le même exemple vu en TD. C'est un simple exemple qui permet d'écrire

une chaîne de caractères d'une tâche Xenomai (*demo_pipe_xeno.c*) vers une tâche Linux (*demo_pipe_linux.c*).
La tâche Xenomai *demo_pipe_xeno.c* se présente comme suit :

```
1  #include <stdio.h>
2  #include <sys/mman.h>
3  #include <alchemy/task.h>
4  #include <alchemy/pipe.h>
5
6  static RT_PIPE my_pipe;
7
8  static void task(void *arg)
9  {
10   char *s="Hello world!";
11   unsigned int sz=strlen(s)+1;
12
13   if ( rt_pipe_write( &my_pipe, s, sz, P_NORMAL) != sz )
14   {
15     printf("rt_pipe_write error\n");
16   }
17 }
18
19 int main (void)
20 {
21   RT_TASK task_desc;
22
23   /* disable memory swap */
24   if ( mlockall( MCL_CURRENT | MCL_FUTURE ) != 0 )
25   {
26     printf("mlockall error\n");
27     return 1;
28   }
29
30   if ( rt_pipe_create( &my_pipe, "rtp0", P_MINOR_AUTO, 0 ) != 0 )
31   {
32     printf("rt_pipe_create error\n");
33     return 1;
34   }
35
36   if (rt_task_spawn( &task_desc, /* task descriptor */
```

```
37                          "my task", /* name */
38                           0 /* 0 = default stack size */,
39                           99 /* priority */,
40                           T_JOINABLE, /* needed to call rt_task_join after */
41                           &task, /* entry point (function pointer) */
42                           NULL /* function argument */ )!=0)
43   {
44     printf("rt_task_spawn error\n");
45     return 1;
46   }
47
48   /* wait for task function termination */
49   rt_task_join(&task_desc);
50
51   return 0;
52 }
```

La tâche Xenomai *demo_pipe_xeno.c* se présente comme suit :

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <unistd.h>
6
7  int main(void)
8  {
9    char buf[128];
10   int fd = open( "/dev/rtp0", O_RDWR );
11
12   if (fd < 0)
13   {
14     printf("open error (%d)\n",fd);
15     return 1;
16   }
17   if ( 0 < read(fd, buf, sizeof(buf)) )
18   {
19     printf("==> %s\n",buf);
20   }
21 }
```

Maintenant, exécutons cet exemple. Pour cela ouvrez deux terminaux différents. Sur le premier terminal, exécutez :

```
# sudo su
# cd pipe/
# make
# ./demo_pipe_linux
==> Hello world!
```

Sur le deuxième terminal, exécutez :

```
# sudo su
# cd pipe/
# ./demo_pipe_xeno
```

Ainsi, cela vous montre un exemple de communication en utilisant un *PIPE* entre les deux domaines : primaire avec (Xenomai) et secondaire (Linux).

1.2 Utilisation du protocole XDDP

Le framework *Real Time Interprocess Communications* (*RTIPC*) a été rajouté dans Xenomai 2.5.x et a été pris comme base dans Xenomai 3. L'objectif est de généraliser le fonctionnement des *PIPEs* qui devraient être supprimés progressivement. *RTIPC* est un "méta-pilote" basé sur *RTDM*, sur lequel on peut empiler des pilotes de protocole, exportant une interface de type socket vers les tâches temps réel s'exécutant en mode principal dans le domaine Xenomai. L'intérêt de *RTIPC* est que leurs utilisateurs n'ont pas à switcher de mode temps réel pour envoyer / recevoir des données vers / depuis d'autres destinations / sources.
Ce framework intègre deux familles de protocoles :

— *Cross-Domain Datagram Protocol* (*XDDP*) : il permet d'échanger les datagrammes (messages) entre le domaine temps réel Xenomai (principal) et le domaine Linux (secondaire). Fondamentalement, il connecte un port *RTDM* en temps réel à l'un des pseudo-périphériques */dev/rtp**. Le port réseau utilisé du côté du socket correspond au numéro de périphérique mineur utilisé du côté non temps-réel (Linux).

— *Intra-Domain Datagram Protocol* (*IDDP*) : il permet d'échanger les messages via un canal de datagrammes temps réel de Xenomai à Xenomai. Ce protocole exporte une interface socket pour échanger des messages. L'idée de base derrière tout cela est que tout ce que vous pourriez faire sur la base des sockets *AF_UNIX* dans le domaine Linux devrait être aussi faisable avec

AF_RTIPC et *IDDP* dans le domaine Xenomai. Cependant, il utilise plutôt des numéros des port numériques ou des chaînes d'étiquettes (label), et non des chemins de socket pour lier des sockets dans l'espace des noms de Xenomai.

Maintenant regardons un exemple d'un peu plus près. Pour cela, soit un exemple d'une tâche Xenomai définie dans le fichier *demo_xddp_xeno.c* :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
5  #include <string.h>
6  #include <malloc.h>
7  #include <pthread.h>
8  #include <fcntl.h>
9  #include <errno.h>
10 #include <rtdm/ipc.h>
11
12 pthread_t rt;
13
14 /* [0..CONFIG_XENO_OPT_PIPE_NRDEV - 1] and /dev/rtpX */
15 #define XDDP_PORT 0
16
17 static const char *msg = "Hello world!";
18
19 static void *realtime_thread(void *arg)
20 {
21         struct sockaddr_ipc saddr;
22         int ret, s, len;
23         struct timespec ts;
24         size_t poolsz;
25         char buf[128];
26
27         s = socket(AF_RTIPC, SOCK_DGRAM, IPCPROTO_XDDP);
28         if (s < 0) {
29                 printf("socket error\n");
30         }
31
32         poolsz = 16384; /* bytes */
33         ret = setsockopt(s, SOL_XDDP, XDDP_POOLSZ,
```

```
34                              &poolsz, sizeof(poolsz));
35          if (ret)
36                  printf("setsockopt␣error\n");
37
38          memset(&saddr, 0, sizeof(saddr));
39          saddr.sipc_family = AF_RTIPC;
40          saddr.sipc_port = XDDP_PORT;
41          ret = bind(s, (struct sockaddr *)&saddr, sizeof(saddr));
42          if (ret)
43                  printf("bind␣error\n");
44
45          len = strlen(msg);
46
47          printf("%s:␣sent␣%d␣bytes,␣\"%.*s\"\n", __FUNCTION__, len, len, msg);
48
49          ret = sendto(s, msg, len, 0, NULL, 0);
50          if (ret != len)
51                  printf("sendto␣error\n");
52
53          ret = recvfrom(s, buf, sizeof(buf), 0, NULL, 0);
54          if (ret <= 0)
55                  printf("recvfrom␣error\n");
56
57          printf("==>␣Received␣from␣Linux␣%d␣bytes␣:␣%.*s\n", ret, ret, buf);
58
59          return NULL;
60  }
61
62  int main(int argc, char **argv)
63  {
64          struct sched_param rtparam = { .sched_priority = 42 };
65          pthread_attr_t rtattr;
66
67          pthread_attr_init(&rtattr);
68          pthread_attr_setdetachstate(&rtattr, PTHREAD_CREATE_JOINABLE);
69          pthread_attr_setinheritsched(&rtattr, PTHREAD_EXPLICIT_SCHED);
70          pthread_attr_setschedpolicy(&rtattr, SCHED_FIFO);
71          pthread_attr_setschedparam(&rtattr, &rtparam);
72
```

```
73          errno = pthread_create(&rt, &rtattr, &realtime_thread, NULL);
74          if (errno)
75                  printf("pthread_create error\n");
76
77          pthread_join(rt, NULL);
78
79          return 0;
80  }
```

Nous avons aussi un exemple d'une tâche Linux définie dans le fichier *demo_xddp_linux.c* :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <unistd.h>
8
9  int main(void)
10 {
11          char buf[128];
12          char *resp="Message Received!";
13          int fd, ret, len;
14
15          fd = open( "/dev/rtp0", O_RDWR );
16          if (fd < 0)
17                  printf("open error\n");
18
19          /* Get the next message from realtime_thread. */
20          ret = read(fd, buf, sizeof(buf));
21          if (ret <= 0)
22                  printf("read error\n");
23          else
24                  printf("==> Received from Realtime %d bytes : %.*s\n", ret, ret, buf);
25
26          /* Echo the message back to realtime_thread. */
27          len = strlen(resp);
28          printf("%s: sent %d bytes, \"%.*s\"\n", __FUNCTION__, len, len, resp);
29          ret = write(fd, resp, len);
30          if (ret <= 0)
```

```
31                      printf("write␣error\n");
32 }
```

Voyons maintenant ce que l'on obtient en exécutant ces deux exemples dans deux terminaux différents.
Sur le premier terminal, exécutez :

```
1 # cd ../xddp-user/
2 # ./demo_xddp_xeno
3 realtime_thread: sent 12 bytes, "Hello␣world!"
4 ==> Received from Linux 17 bytes : Message Received!
```

Sur le deuxième terminal, exécutez :

```
1 # cd ../xddp-user/
2 # ./demo_xddp_linux
3 ==> Received from Realtime 12 bytes : Hello world!
4 main: sent 17 bytes, "Message␣Received!"
```

Ainsi, cela vous montre un exemple de communication en utilisant les sockets *XDDP* entre les deux domaines : primaire avec (Xenomai) et secondaire (Linux) dans l'espace utilisateur.

2 Communications entre l'espace utilisateur et l'espace noyau

Dans cette section, nous allons voir un moyen simple de communication entre un pilote temps-réel qui s'exécute dans l'espace noyau Xenomai et qui utilise l'API *RTDM* et une tâche Linux non temps-réel qui s'exécute dans l'espace utilisateur de Linux. Dans cet exemple, nous allons principalement utiliser les deux fonctions de copie temps-réel *rtdm_safe_copy_to_user()* et *rtdm_safe_copy_from_user()* qui permettent respectivement de copier un buffer dans l'espace utilisateur d'une manière sûre (sans risque de changer de domaine) et de copier un buffer depuis l'espace utilisateur d'une manière sûre aussi.
Ainsi, nous prenons l'exemple d'un simple driver (pilote) *RTDM* défini dans le fichier *rtdm-dev.c* qui permet de récupérer / envoyer une chaîne de caractères depuis / vers l'espace utilisateur :

```
1 #include <linux/version.h>
2 #include <linux/device.h>
3 #include <linux/module.h>
```

```
4  #include <linux/sched.h>
5  #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 128
13 static char my_data [MY_DATA_SIZE];
14
15 static const char *msg = "Hello␣world!";
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
30 static int my_read_nrt_function (struct rtdm_fd *fd, void __user *buffer,
31         size_t lg)
32 {
33         rtdm_printk(KERN_INFO "%s.%s()\n", THIS_MODULE->name,
34                         __FUNCTION__);
35
36         if (lg > 0) {
37                 if (rtdm_safe_copy_to_user(fd, buffer, msg, lg) != 0) {
38                         return -EFAULT;
39                 }
40         }
41
42         rtdm_printk("%s:␣sent␣%d␣bytes,␣\"%.*s\"\n", __FUNCTION__,
```

```
43                          strlen(msg), strlen(msg), msg);
44
45          return lg;
46  }
47
48  static int my_write_nrt_function(struct rtdm_fd *fd, const void __user *buffer,
49          size_t lg)
50  {
51          rtdm_printk(KERN_INFO "%s.%s()\n", THIS_MODULE->name,
52                  __FUNCTION__);
53
54          if (lg > 0) {
55                  if (rtdm_safe_copy_from_user(fd,
56                                  &my_data,
57                                  buffer, lg) != 0) {
58                          return -EFAULT;
59                  }
60          }
61
62          rtdm_printk("==> Received from Linux %d bytes : %.*s\n",
63                          lg, lg, my_data);
64
65          return lg;
66  }
67
68  static struct rtdm_driver my_rt_driver = {
69
70          .profile_info = RTDM_PROFILE_INFO(my_example,
71                          RTDM_CLASS_TESTING, 1, 1),
72
73          .device_flags = RTDM_NAMED_DEVICE,
74          .device_count = 1,
75          .context_size = 0,
76
77          .ops = {
78                  .open = my_open_function,
79                  .close = my_close_function,
80                  .read_nrt = my_read_nrt_function,
81                  .write_nrt = my_write_nrt_function,
```

```
82          },
83 };
84
85 static struct rtdm_device my_rt_device = {
86
87          .driver = &my_rt_driver,
88          .label = "rtdm_driver_%d",
89 };
90
91 static int __init my_module_init (void)
92 {
93          rtdm_dev_register(&my_rt_device);
94          return 0;
95 }
96
97 static void __exit my_module_exit (void)
98 {
99          rtdm_dev_unregister(&my_rt_device);
100 }
101
102 module_init(my_module_init);
103 module_exit(my_module_exit);
104 MODULE_LICENSE("GPL");
```

Du côté de l'espace utilisateur, on utilisera la tâche Linux définie dans le fichier *demo_dev_linux.c* :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <unistd.h>
8
9  int main(void)
10 {
11          char buf[128];
12          char *resp="Message␣Received!";
13          int fd, ret, len;
14
```

```
15          fd = open( "/dev/rtdm/rtdm_driver_0", O_RDWR );
16          if (fd < 0)
17                  printf("open error\n");
18
19          /* Get the next message from realtime device. */
20          ret = read(fd, buf, sizeof(buf));
21          if (ret <= 0)
22                  printf("read error\n");
23          else
24                  printf("==> Received from Realtime %d bytes : %.*s\n",
25                          strlen(buf), strlen(buf), buf);
26
27          /* Echo the message back to realtime device. */
28          len = strlen(resp);
29          printf("%s: sent %d bytes, \"%.*s\"\n", __FUNCTION__,
30                          len, len, resp);
31          ret = write(fd, resp, len);
32          if (ret <= 0)
33                  printf("write error\n");
34  }
```

Maintenant, exécutons cet exemple :

```
1  # cd ../dev/
2  # make
3  # insmod rtdm-dev.ko
4  # ./demo_dev_linux
5  # rmmod rtdm-dev
6  # dmesg | tail
7  ...
8   [10234.109083] rtdm_dev.my_open_function()
9  [10234.109137] rtdm_dev.my_read_nrt_function()
10 [10234.109145] my_read_nrt_function: sent 12 bytes, "Hello world!"
11 [10234.109306] rtdm_dev.my_write_nrt_function()
12 [10234.109312] ==> Received from Linux 17 bytes : Message Received!
13 [10234.109534] rtdm_dev.my_close_function()
```

Ainsi les messages reçus et envoyés par le pilote *RTDM* sur le log du noyau. Cet exemple sera un bon départ si vous voulez développer votre propre pilote pour votre projet.

3 Conclusion sur la communication inter-processus

Dans ce TD, nous avons exploré quelques exemples de mécanismes de communication entre les tâches temps réel définies dans l'espace utilisateur et une tâche Linux en utilisant les *PIPEs* et le protocole *XDDP* ainsi qu'entre un pilote *RTDM* et une tâche Linux. Bien sûr ces moyens de communication ne sont pas exhaustifs, il en existe d'autres qui ne sont pas présentés ici comme les mémoires partagés pour la communication inter-domaines. De plus, la communication intra-domaine entre tâches temps-réel de Xenomai à Xenomai n'a pas été traitée dans ce TD. Pour cela, on pourra utiliser par exemple les files de messages (message queue) ou les sockets définies par le protocole *IDDP*.
N'hésitez pas à vous inspirer et à utiliser ces codes à volonté pour vos différents projets.