
# MI11
# TP Linux Temps Réel

Jeanneau Louis, Schulster Alex

Printemps 2022

# Table des matières

# 1 Mise en place de Xenomai

## 1.1 Construction de l'image

Afin de construire une nouvelle `core-image-base` utilisant *Xenomai*, nous modifions le fichier `/opt/mi11/poky/build/conf/local.conf` en indiquant que la machine serait cette fois-ci un `joypinote-xenomai` (Cf. Fig 1).

```
#
# Machine Selection
#
# You need to select a specific
# of emulated machines available
#
MACHINE ?= "joypinote-xenomai"
#MACHINE ?= "qemuarm"
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemumips"
#MACHINE ?= "qemumips64"
```

FIGURE 1 – Modification du fichier `local.conf`

Il nous faut ensuite nous déplacer dans le directoire `/opt/mi11/poky/`, puis source `../poky-dunfell-23.0.16/oe-init-build-env` afin de pouvoir utiliser `bitbake`. Enfin, nous pouvons lancer la compilation de la nouvelle image avec la commande `bitbake core-image-base`.

## 1.2 Faire démarrer la cible

Afin de faire démarrer la cible sur notre nouvelle image, nous devons nous rendre dans le dossier `/opt/mi11/poky/build/tmp/deploy/images/joypinote-xenomai/`. On copie alors l'image (`zImage`), ainsi que le *device tree*, `cmdline.txt`, `config.txt`, et `start4.elf` dans le répertoire `tftpboot/`*device_id*`/`.

De plus, nous devons extraire le tout nouveau système de fichiers dans le répertoire `tftpboot/rootfs`.

On démarre la cible et on s'assure qu'elle utilise bien xenomai en lançant la commande `latency`, qui retourne le résultat suivant :

```
joypinote-xenomai login: root
root@joypinote-xenomai:~# latency
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01 (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|    -3.113|    -2.679|     1.832|      0|     0|    -3.113|     1.832
RTD|    -3.113|    -2.742|     0.332|      0|     0|    -3.113|     1.832
RTD|    -2.910|    -2.576|     0.423|      0|     0|    -3.113|     1.832
```

```
RTD|    -2.892|    -2.605|     0.552|      0|     0|    -3.113|     1.832
RTD|    -3.004|    -2.689|     0.422|      0|     0|    -3.113|     1.832
RTD|    -3.023|    -2.700|     1.144|      0|     0|    -3.113|     1.832
RTD|    -2.986|    -2.729|     0.384|      0|     0|    -3.113|     1.832
RTD|    -3.080|    -2.725|     0.532|      0|     0|    -3.113|     1.832
RTD|    -3.024|    -2.759|     0.494|      0|     0|    -3.113|     1.832
RTD|    -3.044|    -2.772|     0.475|      0|     0|    -3.113|     1.832
RTD|    -3.045|    -2.733|     0.974|      0|     0|    -3.113|     1.832
RTD|    -3.045|    -2.766|     0.510|      0|     0|    -3.113|     1.832
RTD|    -2.916|    -2.689|     0.491|      0|     0|    -3.113|     1.832
RTD|    -2.972|    -2.746|     0.435|      0|     0|    -3.113|     1.832
RTD|    -3.010|    -2.768|     0.472|      0|     0|    -3.113|     1.832
RTD|    -2.992|    -2.595|     2.397|      0|     0|    -3.113|     2.397
RTD|    -3.011|    -2.758|     2.712|      0|     0|    -3.113|     2.712
RTD|    -3.011|    -2.772|     0.415|      0|     0|    -3.113|     2.712
RTD|    -3.104|    -2.587|     0.507|      0|     0|    -3.113|     2.712
RTD|    -2.883|    -2.582|     0.524|      0|     0|    -3.113|     2.712
RTD|    -3.106|    -2.767|     1.246|      0|     0|    -3.113|     2.712
RTT|  00:00:22 (periodic user-mode task, 1000 us period, priority 99)
```

(On remarque par ailleurs que la machine porte le nom `joypinote-xenomai`, indice supplémentaire de l'utilisation de l'image `xenomai`).

# 2 Hello world temps réel

Nous exécutons le programme compilé lors du dernier TP sur le `Joy-Pi-Note` puis nous étudions les statistiques `xenomai` et nous obtenons :

```
root@joypinote-xenomai:/usr# cat /proc/xenomai/sched/stat
CPU  PID    MSW       CSW       XSC       PF    STAT      %CPU  NAME
   0  0      0         51963     0         0     00018000 100.0  [ROOT]
   0  495    11        19        43        0     00060044   0.0  hello
   0  0      0         132597    0         0     00000000   0.0  [IRQ18: [timer]]
root@joypinote-xenomai:/usr# cat /proc/xenomai/sched/threads
CPU  PID    CLASS  TYPE     PRI   TIMEOUT      STAT      NAME
   0  0      idle   core      -1   -            R         [ROOT]
   0  495    rt     cobalt     0   868ms369us   D         hello
```

Nous ne voyons pas notre programme `hello`, signe que celui-ci n'est pas exécuté en temps réel. Nous allons donc le modifier pour corriger cela.

## 2.1 Utilisation de tâches

Voici le code mis à jour de notre programme :

```c
// Importing neede libraries
#include <stdio.h>
#include <unistd.h>
#include <alchemy/task.h>

// Defining constants
#define TASK_PRIO 99
#define TASK_MODE T_JOINABLE
#define TASK_STKSZ 0

// Task responsible for saying ‘hello world‘
void say_hello_task() {
```

```
   // Infinite loop
   for (;;)
   {
      // Print ‘hello world‘
      printf("Hello, World!\n");
      // Sleep for one second
      sleep(1);
   }
}

int main() {
   // Defining variables needed for task creation
   int err;
   RT_TASK task_desc;

   // Creating task
   err = rt_task_create(&task_desc, "hello", TASK_STKSZ, TASK_PRIO, TASK_MODE);
   // If an error occured, stop the program
   if (err != 0) {
      printf("error rt_task_create\n");
      return 1;
   }

   // Start the task
   rt_task_start(&task_desc, &say_hello_task, NULL);
   // Wait for the task to finish
   rt_task_join(&task_desc);
   // Delete the task
   rt_task_delete(&task_desc);

   return 0;
}
```

Analyse des statistiques `xenomai` :

```
root@joypinote-xenomai:/usr# cat /proc/xenomai/sched/stat
CPU  PID    MSW        CSW        XSC        PF    STAT       %CPU  NAME
   0  0      0          52097      0          0     00018000 100.0  [ROOT]
   0  549    8          10         54         0     000680c0   0.0  hello
   0  551    3          5          12         0     000480c0   0.0  hello
   0  0      0          153663     0          0     00000000   0.0  [IRQ18: [timer]]
root@joypinote-xenomai:/usr# cat /proc/xenomai/sched/threads
CPU  PID    CLASS  TYPE     PRI   TIMEOUT      STAT       NAME
   0  0      idle   core      -1   -            R          [ROOT]
   0  549    rt     cobalt     0   -            X          hello
   0  551    rt     cobalt    99   -            X          hello
```

Cette fois-ci notre programme apparaît bien.

## 2.2 rt_task_sleep

```
// Task responsible for saying ‘hello world‘
void say_hello_task() {
   // Infinite loop
   for (;;)
   {
      // Print ‘hello world‘
      printf("Hello, World!\n");
```

```
    // Sleep for one second
    rt_task_sleep(1000000000);
  }
}
```

Analyse des statistiques `xenomai` :

```
root@joypinote-xenomai:~# cat /proc/xenomai/sched/stat
CPU  PID    MSW       CSW       XSC       PF    STAT      %CPU  NAME
   0  0      0         52126     0         0     00018000 100.0  [ROOT]
   0  558    8         10        54        0     000680c0   0.0  hello
   0  560    11        22        20        0     00048044   0.0  hello
   0  0      0         156192    0         0     00000000   0.0  [IRQ18: [timer]]
root@joypinote-xenomai:~# cat /proc/xenomai/sched/threads
CPU  PID    CLASS  TYPE     PRI   TIMEOUT      STAT      NAME
   0  0      idle   core      -1   -            R         [ROOT]
   0  558    rt     cobalt     0   -            X         hello
   0  560    rt     cobalt    99   874ms291us   D         hello
```

## 2.3 rt_printf

```
// Task responsible for saying ‘hello world‘
void say_hello_task() {
  // Infinite loop
  for (;;)
  {
    // Print ‘hello world‘
    rt_printf("Hello, World!\n");
    // Sleep for one second
    rt_task_sleep(1000000000);
  }
}
```

Analyse des statistiques `xenomai` :

```
root@joypinote-xenomai:~# cat /proc/xenomai/sched/stat
CPU  PID    MSW       CSW       XSC       PF    STAT      %CPU  NAME
   0  0      0         52148     0         0     00018000 100.0  [ROOT]
   0  568    8         10        54        0     000680c0   0.0  hello
   0  571    2         6         13        0     00048044   0.0  hello
   0  0      0         157058    0         0     00000000   0.0  [IRQ18: [timer]]
root@joypinote-xenomai:~# cat /proc/xenomai/sched/threads
CPU  PID    CLASS  TYPE     PRI   TIMEOUT      STAT      NAME
   0  0      idle   core      -1   -            R         [ROOT]
   0  568    rt     cobalt     0   -            X         hello
   0  571    rt     cobalt    99   624ms341us   D         hello
```

## 2.4 Analyse

On remarque que, à partir du moment où nous utilisons une tâche, un deuxième processus apparaît dans les statistiques `xenomai`. De plus, lorsque nous utilisons la fonction `rt_task_sleep` mais le `printf` classique, le nombre de `Mode Switch (MSW)` de la tache augmente avec le temps, alors que ce n'est pas le cas lorsque nous utilisons uniquement ou aucune fonction rendant la tâche temps réel.

# 3 Synchronisation

## 3.1 Code source

```c
// Importing needed libraries
#include <stdio.h>
#include <unistd.h>
#include <alchemy/task.h>

// Defining constants
#define TASK_PRIO_1 99
#define TASK_PRIO_2 2
#define TASK_MODE T_JOINABLE
#define TASK_STKSZ 0

// Task responsible for saying ‘hello‘
void say_hello_task() {
   rt_printf("Hello,");
}

// Task responsible for saying ‘world‘
void say_world_task() {
   rt_printf("World!");
}

int main() {
   // Defining variables needed for task creation
   int err;
   RT_TASK task_hello;
   RT_TASK task_world;

   // Creating task
   err = rt_task_create(&task_hello, "hello", TASK_STKSZ, TASK_PRIO_2, TASK_MODE);
   // If an error occured, stop the program
   if (err != 0) {
      printf("error rt_task_create hello\n");
      return 1;
   }

   // Creating task
   err = rt_task_create(&task_world, "world", TASK_STKSZ, TASK_PRIO_1, TASK_MODE);
   // If an error occured, stop the program
   if (err != 0) {
      printf("error rt_task_create world\n");
      rt_task_delete(&task_hello);
      return 1;
   }

   // Start the tasks
   rt_task_start(&task_world, &say_world_task, NULL);
   rt_task_start(&task_hello, &say_hello_task, NULL);
   // Wait for the tasks to finish
   rt_task_join(&task_hello);
   rt_task_join(&task_world);
   // Delete the tasks
   rt_task_delete(&task_hello);
   rt_task_delete(&task_world);
   return 0;
}
```

## 3.2 Priorité des tâches

En tentant de modifier la priorité des 2 tâches, nous remarquons que cela n'a pas d'influence sur l'ordre d'exécution des 2 tâches. Cependant, changer l'ordre de démarrage des tâches a bien une influence sur l'ordre d'exécution. Ceci est dû au fait que la première tâche démarrée aura le temps de s'exécuter avant que la deuxième ne soit démarrer (peu importes leurs priorités respectives donc).

## 3.3 Sémaphore

Afin que le sémaphore soit bloquant, nous allons initialiser sa valeur à 0. Ainsi, les deux tâches démarrées seront bloquées toutes les deux dans le sémaphore sans pouvoir continuer leur exécution. Une fois les deux tâches démarrées, on relâchera une fois le sémaphore dans le programme principal.

Pour ce qui est du mode du sémaphore, nous avons le choix entre `S_FIFO` et `S_PRIO` qui permettent d'ordonnancer différemment l'accès au sémaphore pour les tâches en attentes. Ici, nous souhaitons faire en sorte que la tâche plus prioritaire passe en premier (même si elle est arrivée après) : on utilise donc le mode `S_PRIO`.

Voici le code du programme résultant :

```
// Importing needed libraries
#include <stdio.h>
#include <unistd.h>
#include <alchemy/task.h>
#include <alchemy/sem.h>

// Defining constants
#define TASK_PRIO_1 99
#define TASK_PRIO_2 2
#define TASK_MODE T_JOINABLE
#define TASK_STKSZ 0

RT_SEM semaphore;

// Task responsible for saying ‘hello‘
void say_hello_task() {
   rt_sem_p(&semaphore, TM_INFINITE);
   rt_printf("Hello,");
   rt_sem_v(&semaphore);
}

// Task responsible for saying ‘world‘
void say_world_task() {
   rt_sem_p(&semaphore, TM_INFINITE);
   rt_printf("World!");
   rt_sem_v(&semaphore);
}

int main() {
// Defining variables needed for task creation
int err;
RT_TASK task_hello;
RT_TASK task_world;

// Creating semaphore
```

```
36 err = rt_sem_create(&semaphore, "semaph", 0, S_PRIO);
37 // If an error occured, stop the program
38 if (err != 0) {
39    printf("error rt_sem_create\n");
40    return 1;
41 }
42
43 // Creating task
44 err = rt_task_create(&task_hello, "hello", TASK_STKSZ, TASK_PRIO_1, TASK_MODE);
45 // If an error occured, stop the program
46 if (err != 0) {
47    printf("error rt_task_create hello\n");
48    rt_sem_delete(&semaphore);
49    return 1;
50 }
51 // Creating task
52 err = rt_task_create(&task_world, "world", TASK_STKSZ, TASK_PRIO_2, TASK_MODE);
53 // If an error occured, stop the program
54 if (err != 0) {
55    printf("error rt_task_create world\n");
56    rt_task_delete(&task_hello);
57    rt_sem_delete(&semaphore);
58    return 1;
59 }
60
61 // Start the tasks
62 rt_task_start(&task_hello, &say_hello_task, NULL);
63 rt_task_start(&task_world, &say_world_task, NULL);
64 // Create a slot in the semaphore
65 rt_sem_v(&semaphore);
66 // Wait for the tasks to finish
67 rt_task_join(&task_hello);
68 rt_task_join(&task_world);
69 // Delete the tasks
70 rt_task_delete(&task_hello);
71 rt_task_delete(&task_world);
72 // Delete the semaphore
73 rt_sem_delete(&semaphore);
74 printf("\n");
75 return 0;
76 }
```

Le programme `main` se charge de créer le sémaphore et les deux tâches. Il initialise le sémaphore à 0 en mode `S_RPIO` (comme justifié précédemment), puis démarre les deux tâches qui vont alors toutes les deux être en attente sur le sémaphore. La fonction main libère alors une place sur le sémaphore pour que la tâche la plus prioritaire puisse s'exécuter (qui cèdera sa place à la deuxième une fois son exécution complétée...). Pendant, ce temps, le programme `main` attend l'exécution complète des deux tâches avant de les supprimer et de terminer le progamme.

## 3.4 Métronome

```
1 // Importing needed libraries
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <alchemy/task.h>
5 #include <alchemy/sem.h>
```

```
6
7  // Defining constants
8  #define TASK_PRIO_1 99
9  #define TASK_PRIO_2 50
10 #define TASK_MODE T_JOINABLE
11 #define TASK_STKSZ 0
12
13 RT_SEM semaphore;
14
15 // Task responsible for saying ‘hello‘
16 void say_hello_task() {
17    for (;;)
18    {
19       rt_sem_p(&semaphore, TM_INFINITE);
20       rt_printf("Hello,");
21    }
22 }
23
24 // Task responsible for saying ‘world‘
25 void say_world_task() {
26    for (;;)
27    {
28       rt_sem_p(&semaphore, TM_INFINITE);
29       rt_printf("World!");
30    }
31 }
32
33 // Task responsible for the timing
34 void metronome_task() {
35    for(;;) {
36       rt_sem_v(&semaphore);
37       rt_sem_v(&semaphore);
38       rt_printf("\n");
39       rt_task_sleep(1000000000);
40    }
41 }
42
43 int main() {
44    // Defining variables needed for task creation
45    int err;
46    RT_TASK task_hello;
47    RT_TASK task_world;
48    RT_TASK task_metro;
49
50    // Creating semaphore
51    err = rt_sem_create(&semaphore, "semaph", 0, S_FIFO);
52    // If an error occured, stop the program
53    if (err != 0) {
54       printf("error rt_sem_create\n");
55       return 1;
56    }
57
58    // Creating task
59    err = rt_task_create(&task_hello, "hello", TASK_STKSZ, TASK_PRIO_1, TASK_MODE);
60    // If an error occured, stop the program
61    if (err != 0) {
62       printf("error rt_task_create hello\n");
63       rt_sem_delete(&semaphore);
64       return 1;
65    }
66
```

```
67    // Creating task
68    err = rt_task_create(&task_world, "world", TASK_STKSZ, TASK_PRIO_2, TASK_MODE);
69    // If an error occured, stop the program
70    if (err != 0) {
71       printf("error rt_task_create world\n");
72       rt_task_delete(&task_hello);
73       rt_sem_delete(&semaphore);
74       return 1;
75    }
76
77    // Creating task
78    err = rt_task_create(&task_metro, "metro", TASK_STKSZ, 2, TASK_MODE);
79    // If an error occured, stop the program
80    if (err != 0) {
81       printf("error rt_task_create metronome\n");
82       rt_task_delete(&task_hello);
83       rt_task_delete(&task_world);
84       rt_sem_delete(&semaphore);
85       return 1;
86    }
87
88    // Start the tasks
89    rt_task_start(&task_hello, &say_hello_task, NULL);
90    rt_task_start(&task_world, &say_world_task, NULL);
91    rt_task_start(&task_metro, &metronome_task, NULL);
92    // Wait for the tasks to finish
93    rt_task_join(&task_hello);
94    rt_task_join(&task_world);
95    rt_task_join(&task_metro);
96    // Delete the tasks
97    rt_task_delete(&task_hello);
98    rt_task_delete(&task_world);
99    rt_task_delete(&task_metro);
100   // Delete the semaphore
101   rt_sem_delete(&semaphore);
102   printf("\n");
103   return 0;
104 }
```

Nous créons une tâche supplémentaire responsable de la synchronisation des deux tâches d'affichage. Nous passons le sémaphore en mode `S_FIFO` pour assurer un roulement dans l'exécution des tâches (autrement, ce serait toujours la tache la plus prioritaire qui s'exécuterait 2 fois par cycle).

Le principe de fonctionnement est assez simple : les deux tâches d'affichage veulent entrer dans un sémaphore qui est initialisé à 0 pour afficher leur partie du message. Le métronome quant à lui libère 2 places dans le sémaphore avant de dormir pendant 1 seconde (et ce à l'infini).

```
root@joypinote-xenomai:~# cat /proc/xenomai/sched/stat
CPU  PID    MSW        CSW        XSC        PF    STAT       %CPU  NAME
   0  0      0          708        0          0     00018000 100.0  [ROOT]
   0  413    18         24         97         0     000680c0   0.0  sem
   0  415    2          186        193        0     00048042   0.0  hello
   0  416    2          186        193        0     00048042   0.0  world
   0  417    2          547        554        0     00048044   0.0  metro
   0  0      0          19215      0          0     00000000   0.0  [IRQ18: [timer]]
root@joypinote-xenomai:~# cat /proc/xenomai/sched/threads
CPU  PID    CLASS  TYPE     PRI   TIMEOUT      STAT       NAME
```

```
  0  0      idle   core      -1   -           R         [ROOT]
  0  413    rt     cobalt     0   -           X         sem
  0  415    rt     cobalt    99   -           W         hello
  0  416    rt     cobalt    50   -           W         world
  0  417    rt     cobalt     2   688ms522us  D         metro
```

On remarque dans les statistiques `xenomai` que les différentes tâches sont bien en temps réel continuellement étant donné que le nombre de *Mode switch* reste à 2 tout le long du déroulement du programme. De plus, la seule tâche possédant un *TIMEOUT* est le métronome (la seule utilisant la fonction `rt_trask_sleep`)

# 4 Latence

## 4.1 Code

```c
// Importing needed libraries
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <alchemy/task.h>
#include <alchemy/sem.h>

// Defining constants
#define TASK_PRIO 99
#define TASK_MODE T_JOINABLE
#define TASK_STKSZ 0

// Main task
void task() {
   // Defining time variables
   RTIME beginning;
   RTIME ending;
   RTIME delta;
   RTIME loop = rt_timer_read();
   RTIME loop_2 = rt_timer_read();
   RTIME loop_delta = loop_2 - loop;
   RTIME loop_min = 9999999999;
   RTIME loop_max = 0;

   // Getting starting time
   beginning = rt_timer_read();
   // 10 000 iterations loop
   for (int i = 0; i < 10000; i++)
   {
      // sleep for 1ms
      rt_task_sleep(1000000);

      // Get time at the beginning of the loop
      loop_2 = rt_timer_read();
      // Get the time between previous iteration and current one
      loop_delta = loop_2 - loop;

      // If new min delta, save it
      if (loop_delta < loop_min) {
         loop_min = loop_delta;
      }

```

```
43      // If new max delta, save it
44      else if (loop_delta > loop_max) {
45         loop_max = loop_delta;
46      }
47
48      loop = loop_2;
49   }
50   // Get ending time
51   ending = rt_timer_read();
52   // Calculate delta between start and end
53   delta = ending - beginning;
54
55   // Time intervals calculations
56   int usec_moyen = delta / (1000 * 10000);
57   int delta_ms = delta/1000000;
58   int loop_min_ms = loop_min/1000;
59   int loop_max_ms = loop_max/1000;
60
61   // Printing results
62   rt_printf("Temps d'execution: %dms\n", delta_ms);
63   rt_printf("Intervalle minimal: %dus\n", loop_min_ms-1000);
64   rt_printf("Intervalle maximal: %dus\n", loop_max_ms-1000);
65   rt_printf("Intervalle moyen: %dus\n", usec_moyen-1000);
66 }
67
68 int main() {
69   // Defining variables needed for task creation
70   int err;
71   RT_TASK task_desc;
72
73   // Creating semaphore
74   err = rt_task_create(&task_desc, "task", TASK_STKSZ, TASK_PRIO, TASK_MODE);
75   // If an error occured, stop the program
76   if (err != 0) {
77      printf("error rt_task_create task\n");
78      return 1;
79   }
80
81   // Start the task
82   rt_task_start(&task_desc, &task, NULL);
83   // Wait for the task to finish
84   rt_task_join(&task_desc);
85   // Delete the task
86   rt_task_delete(&task_desc);
87
88   return 0;
89 }
```

## 4.2 Analyse des performances

```
1 root@joypinote-xenomai:/usr# ./latency
2 Temps d execution: 10066ms
3 Intervalle minimal: 5us
4 Intervalle maximal: 12us
5 Intervalle moyen: 6us
```

On remarque que les erreurs de gestion du temps sont bien plus minimes que lors d'une

utilisation non-temps réel. Ci-dessous l'exécution du même programme tout en stressant le processeur :

```
root@joypinote-xenomai:/usr# ./latency
Temps d execution: 10070ms
Intervalle minimal: 4us
Intervalle maximal: 10us
Intervalle moyen: 7us
```

On voit que cela n'a aucune influence sur les performances de notre programme temps réel. Ceci est dû au fait que le programme `stress` n'est pas temps réel et est donc géré différemment par l'ordonnanceur de `xenomai` qui gère d'abord les tâches temps réelles.