
# MI11
# TP2 Linux Xenomai

Jeanneau Louis, Schulster Alex

Printemps 2022

# Table des matières

# 1 Pathfinder

## 1.1 Principe des fonctions

### 1.1.1 Question 1

La fonction `create_and_start_rt_task` permet de créer, rendre périodique et lancer une tâche temps réel Xenomai. Les informations sur la tâche à démarrer (nom, priorité, périodicité, durée, fonction lancée) sont fournies via un paramètre `struct task_descriptor` qui encapsule les informations.

```
typedef struct task_descriptor{
  RT_TASK task;
  void (*task_function)(void*);
  RTIME period;
  RTIME duration;
  int priority;
  bool use_resource;
} task_descriptor;
```

Le `status` est retourné pour informer du bon déroulé de la fonction : 0 pour un succès, autre pour un échec.

La fonction `rt_task` permet quant à elle de simuler une exécution de tâche qui prend du temps (et optionnellement une ressource de bus) et de disposer d'informations sur la tâche lancée. La tâche attend qu'on libère son sémaphore `start_sem` pour démarrer.

### 1.1.2 Question 2

`rt_task_name` retourne le nom de la tâche temps réel actuellement en cours d'exécution, elle n'a donc pas d'argument.

`RT_TASK_INFO` contient :

- la priorité
- le statut stocké dans une structure `threadobj_stat`
- le nom de la tâche
- le numéro de processus de la tâche

```
int   prio
   // Task priority.

struct threadobj_stat  stat
   // Task status.

char  name [XNOBJECT_NAME_LEN]
   // Name of task.

pid_t   pid
   // Host pid.
```

Pour `threadobj_stat`, le contenu est le suivant :

```
struct threadobj_stat {
        ticks_t xtime; // Temps total passe dans la tache
        ticks_t timeout; // Temps avant lequel la tache s'arrete
        uint64_t msw; // Nombre de changements de mode
        uint64_t csw; // Nombre de changements de contexte
        uint64_t xsc;
        int cpu; // Coeur sur lequel est la tache
        int schedlock;
        unsigned int status; // Status de la tache
        uint32_t pf;
 \};
```

### 1.1.3 Question 3

Une fois toutes les tâches démarrées, on réalise un `broadcast` sur le sémaphore `start_sem` pour lancer leur exécution.

## 1.2 Busy-wait

### 1.2.1 Question 4

Au lancement de la fonction `busy_wait`, le champs `xtime` de la structure `threadobj_stat` nous donne le temps total passé dans une tâche. Nous sauvegardons la valeur de `xtime` à l'entrée dans la fonction et nous comparons cette valeur sauvegardée à la valeur du `xtime` actuel dans une boucle *while* jusqu'à ce qu'il y ait entre les deux un écart de la durée souhaitée.

### 1.2.2 Question 5

Le timing a l'air correct lorsque l'on vérifie l'exécution en utilisant la fonction `ms_time_since_start`.

```
root@joypinote-xenomai:/usr# ./pathfinder
started main program at 0.000ms
started task ORDO_BUS, period 125ms, duration 25ms, use resource 0
doing ORDO_BUS (1.071000 ms)
doing ORDO_BUS ok (26.086500 ms)
doing ORDO_BUS (125.954277 ms)
doing ORDO_BUS ok (150.975220 ms)
doing ORDO_BUS (250.950012 ms)
doing ORDO_BUS ok (275.967804 ms)
doing ORDO_BUS (375.949768 ms)
doing ORDO_BUS ok (400.966919 ms)
doing ORDO_BUS (500.949097 ms)
doing ORDO_BUS ok (525.966064 ms)
doing ORDO_BUS (625.948608 ms)
doing ORDO_BUS ok (650.965515 ms)
doing ORDO_BUS (750.948730 ms)
doing ORDO_BUS ok (775.965149 ms)
doing ORDO_BUS (875.948975 ms)
doing ORDO_BUS ok (900.965271 ms)
doing ORDO_BUS (1000.948853 ms)
doing ORDO_BUS ok (1025.966187 ms)
doing ORDO_BUS (1125.948486 ms)
doing ORDO_BUS ok (1150.965820 ms)
doing ORDO_BUS (1250.948608 ms)
doing ORDO_BUS ok (1275.966309 ms)
doing ORDO_BUS (1375.948486 ms)
doing ORDO_BUS ok (1400.965942 ms)
doing ORDO_BUS (1500.949585 ms)
doing ORDO_BUS ok (1525.966187 ms)
doing ORDO_BUS (1625.948364 ms)
doing ORDO_BUS ok (1650.964233 ms)
doing ORDO_BUS (1750.948975 ms)
doing ORDO_BUS ok (1775.966919 ms)
doing ORDO_BUS (1875.947998 ms)
doing ORDO_BUS ok (1900.964478 ms)
```

## 1.3 Sémaphore d'accès au bus

### 1.3.1 Question 6

Pour l'accès au bus 1553, un sémaphore initialisé à 1 en mode priorité est créée. Une tâche qui prend l'accès appelle `acquire_resource`, qui elle-même tente de prendre le sémaphore. Pour libérer l'accès au bus, on libère le sémaphore à travers la fonction `release_resource`.

```
root@joypinote-xenomai:/usr# ./pathfinder
started main program at 0.000ms
started task ORDO_BUS, period 125ms, duration 25ms, use resource 0
started task DISTRIB_DONNES, period 125ms, duration 25ms, use resource 1
started task PILOTAGE, period 250ms, duration 25ms, use resource 1
started task RADIO, period 250ms, duration 25ms, use resource 0
started task CAMERA, period 250ms, duration 25ms, use resource 0
started task MESURES, period 5000ms, duration 50ms, use resource 1
started task METEO, period 5000ms, duration 50ms, use resource 1
doing ORDO_BUS (9.271537 ms)
doing ORDO_BUS ok (34.290424 ms)
doing DISTRIB_DONNES (34.308723 ms)
doing DISTRIB_DONNES ok (59.319149 ms)
doing PILOTAGE (59.332352 ms)
doing PILOTAGE ok (84.343704 ms)
doing RADIO (84.357094 ms)
doing RADIO ok (109.367516 ms)
doing CAMERA (109.380127 ms)
doing ORDO_BUS (125.946571 ms)
doing ORDO_BUS ok (150.961243 ms)
doing DISTRIB_DONNES (150.976395 ms)
doing DISTRIB_DONNES ok (175.987015 ms)
doing CAMERA ok (184.453400 ms)
doing MESURES (184.466568 ms)
doing MESURES ok (234.476456 ms)
doing METEO (234.488327 ms)
doing ORDO_BUS (250.943558 ms)
doing ORDO_BUS ok (275.963593 ms)
doing RADIO (275.992523 ms)
doing RADIO ok (301.003601 ms)
doing CAMERA (301.017822 ms)
doing CAMERA ok (326.029694 ms)
doing METEO ok (326.041901 ms)
doing DISTRIB_DONNES (326.054932 ms)
doing DISTRIB_DONNES ok (351.066833 ms)
```

Les tâches se déroulent correctement.

## 1.4 Mécanisme de sécurité

### 1.4.1 Question 7

Pour faire le blocage CPU, nous utilisons un sémaphore de sécurité qui est initialisé à 1. A la fin de son exécution, `ORDO_BUS` met le sémaphore à 0 pour indiquer qu'il a fini son exécution. De son côté, `DISTRIB_DONNEE` libère une place dans le sémaphore (le met à 1) à la fin de son exécution pour signifier que les données ont été distribué.

Ainsi, lorsque `ORDO_BUS` commence son exécution, il vérifie que le sémaphore vaut 1 pour continuer. Si le sémaphore vaut autre chose que 1, c'est qu'entre deux exécutions d'`ORDO_BUS` il n'y a pas eu de distribution.

Ce sémaphore est un peu comme un `mutex` de sécurité.

### 1.4.2 Question 8

Lorsque la durée de la tâche `METEO` est de 40 ms, tout s'exécute sans soucis. Si la tâche dure 50 ou 60 ms, un plantage finit par arriver car `ORDO_BUS` peut s'exécuter 2 fois sans que `DISTRIB_DONNEE` entre temps ne s'exécute.

```
root@joypinote-xenomai:/usr# ./pathfinder
started main program at 0.000ms
started task ORDO_BUS, period 125ms, duration 25ms, use resource 0
started task DISTRIB_DONNES, period 125ms, duration 25ms, use resource 1
started task PILOTAGE, period 250ms, duration 25ms, use resource 1
started task RADIO, period 250ms, duration 25ms, use resource 0
started task CAMERA, period 250ms, duration 25ms, use resource 0
started task MESURES, period 5000ms, duration 50ms, use resource 1
started task METEO, period 5000ms, duration 50ms, use resource 1
doing ORDO_BUS (8.925 ms)
doing ORDO_BUS ok (33.946 ms)
doing DISTRIB_DONNES (33.965 ms)
doing DISTRIB_DONNES ok (58.976 ms)
doing PILOTAGE (58.989 ms)
doing PILOTAGE ok (84.001 ms)
doing RADIO (84.014 ms)
doing RADIO ok (109.026 ms)
doing CAMERA (109.039 ms)
doing ORDO_BUS (125.941 ms)
doing ORDO_BUS ok (150.955 ms)
doing DISTRIB_DONNES (150.970 ms)
doing DISTRIB_DONNES ok (175.982 ms)
doing CAMERA ok (184.109 ms)
doing MESURES (184.122 ms)
doing MESURES ok (234.133 ms)
doing METEO (234.145 ms)
doing ORDO_BUS (250.937 ms)
doing ORDO_BUS ok (275.958 ms)
doing RADIO (275.986 ms)
doing RADIO ok (300.997 ms)
doing CAMERA (301.010 ms)
doing CAMERA ok (326.020 ms)
doing METEO ok (359.255 ms)
doing DISTRIB_DONNES (359.267 ms)
doing ORDO_BUS (375.937 ms)
security triggered! Security semaphore count: 0
```

Cela s'explique par un phénomène d'inversion de priorité.

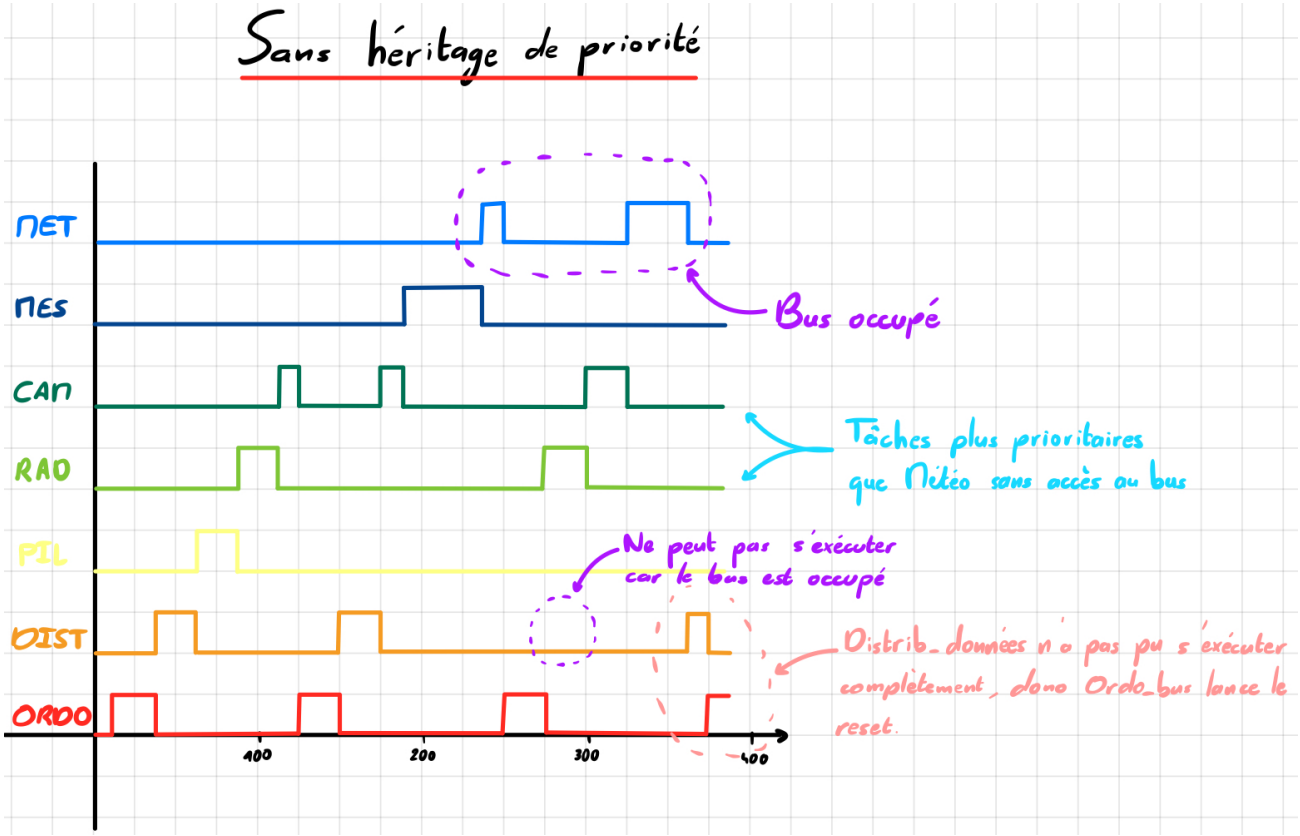


FIGURE 1 – Chronogrammes avec le soucis d'inversion de priorité

### 1.4.3 Question 9

Notre solution consiste à utiliser un `mutex` qui intègre l'héritage de priorité au lieu d'un sémaphore qui ne l'intègre pas. Cela évite ainsi le cas de l'inversion de priorités.

### 1.4.4 Question 10

L'utilisation du `mutex` règle le problème.

```
root@joypinote-xenomai:/usr# METEO = 60ms
root@joypinote-xenomai:/usr# ./pathfinder
started main program at 0.000ms
started task ORDO_BUS, period 125ms, duration 25ms, use resource 0
started task DISTRIB_DONNES, period 125ms, duration 25ms, use resource 1
started task PILOTAGE, period 250ms, duration 25ms, use resource 1
started task RADIO, period 250ms, duration 25ms, use resource 0
started task CAMERA, period 250ms, duration 25ms, use resource 0
started task MESURES, period 5000ms, duration 50ms, use resource 1
started task METEO, period 5000ms, duration 60ms, use resource 1
doing ORDO_BUS (9.375 ms)
doing ORDO_BUS ok (34.394 ms)
doing DISTRIB_DONNES (34.413 ms)
doing DISTRIB_DONNES ok (59.427 ms)
doing PILOTAGE (59.441 ms)
```

```
doing PILOTAGE ok (84.453 ms)
doing RADIO (84.467 ms)
doing RADIO ok (109.479 ms)
doing CAMERA (109.491 ms)
doing ORDO_BUS (125.934 ms)
doing ORDO_BUS ok (150.950 ms)
doing DISTRIB_DONNES (150.966 ms)
doing DISTRIB_DONNES ok (175.977 ms)
doing CAMERA ok (184.565 ms)
doing MESURES (184.579 ms)
doing MESURES ok (234.590 ms)
doing METEO (234.602 ms)
doing ORDO_BUS (250.931 ms)
doing ORDO_BUS ok (275.953 ms)
doing METEO ok (319.665 ms)
doing DISTRIB_DONNES (319.679 ms)
doing DISTRIB_DONNES ok (344.690 ms)
doing PILOTAGE (344.704 ms)
doing PILOTAGE ok (369.717 ms)
doing RADIO (369.730 ms)
doing ORDO_BUS (375.930 ms)
doing ORDO_BUS ok (400.944 ms)
doing DISTRIB_DONNES (400.958 ms)
doing DISTRIB_DONNES ok (425.969 ms)
...
doing ORDO_BUS (5000.937 ms)
doing ORDO_BUS ok (5025.969 ms)
doing DISTRIB_DONNES (5025.987 ms)
doing DISTRIB_DONNES ok (5050.998 ms)
doing PILOTAGE (5051.012 ms)
doing PILOTAGE ok (5076.024 ms)
doing RADIO (5076.039 ms)
doing RADIO ok (5101.050 ms)
doing CAMERA (5101.063 ms)
doing ORDO_BUS (5125.931 ms)
doing ORDO_BUS ok (5150.945 ms)
doing DISTRIB_DONNES (5150.960 ms)
doing DISTRIB_DONNES ok (5175.973 ms)
doing CAMERA ok (5175.985 ms)
doing MESURES (5176.000 ms)
doing MESURES ok (5226.012 ms)
doing METEO (5226.027 ms)
doing ORDO_BUS (5250.930 ms)
doing ORDO_BUS ok (5275.951 ms)
doing METEO ok (5311.087 ms)
doing DISTRIB_DONNES (5311.103 ms)
doing DISTRIB_DONNES ok (5336.114 ms)
doing PILOTAGE (5336.128 ms)
doing PILOTAGE ok (5361.140 ms)
doing RADIO (5361.153 ms)
doing ORDO_BUS (5375.930 ms)
doing ORDO_BUS ok (5400.946 ms)
doing DISTRIB_DONNES (5400.961 ms)
doing DISTRIB_DONNES ok (5425.971 ms)
```

Nous pouvons visualiser que le problème est réglé avec le chronogramme suivant :

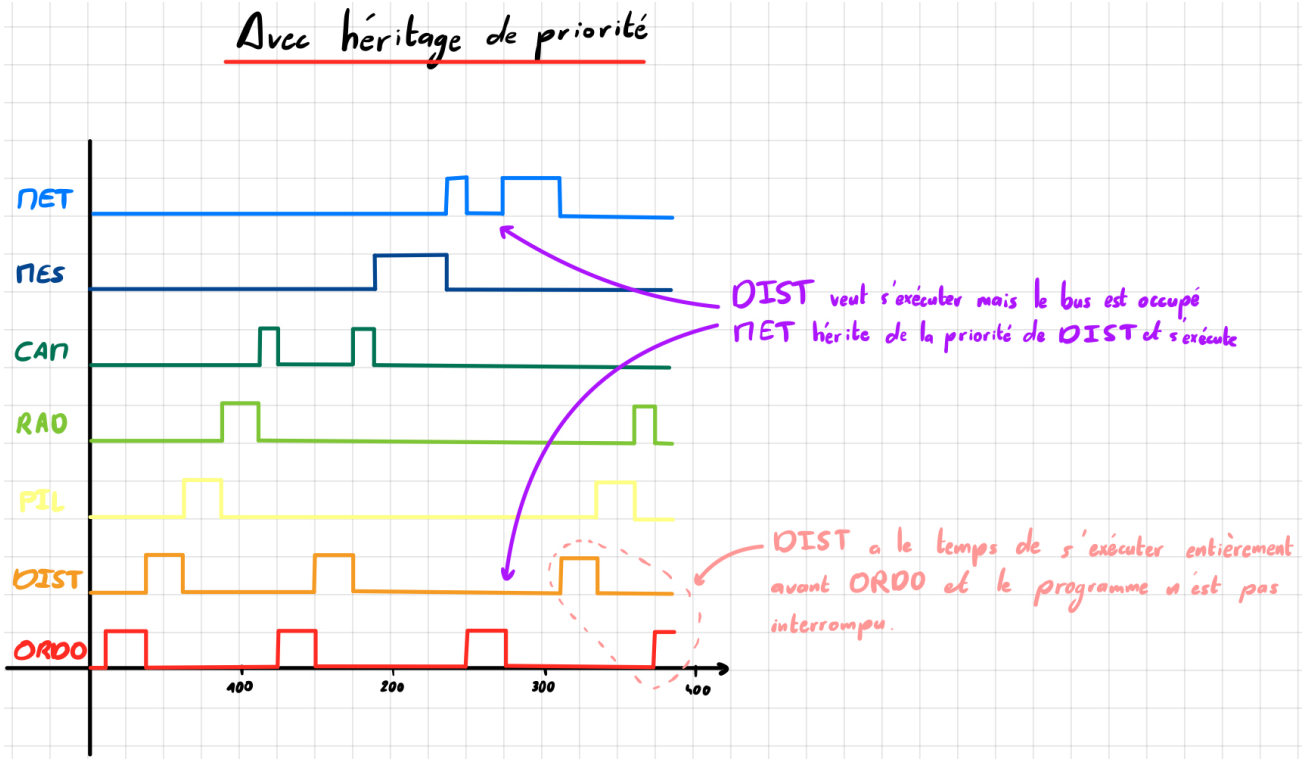


FIGURE 2 – Chronogrammes avec héritage de priorité. Il n'y a plus de problèmes

# 2 Code complet

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include <alchemy/task.h>
#include <alchemy/timer.h>
#include <alchemy/sem.h>
#include <alchemy/mutex.h>

#define TASK_MODE T_JOINABLE
#define TASK_STKSZ 0

// Comment to use semaphore for bus, uncomment for mutex
#define USE_MUTEX

// Semaphore variables
RT_SEM start_sem;
RT_SEM bus_sem;
RT_MUTEX bus_mutex;
RT_SEM security_sem;

typedef struct task_descriptor{
  RT_TASK task;
  // Pointer on function executed by the task
  void (*task_function)(void*);
  // Period of the task
  RTIME period;
  // Execution time of the task
  RTIME duration;
  // Priority of the task
  int priority;
  // Boolean telling if the task uses bus 1553
  bool use_resource;
} task_descriptor;

////////////////////////////////////////////////////////
/**
 * Get the name of the currently running task
 */
char* rt_task_name(void) {
  static RT_TASK_INFO info;
  rt_task_inquire(NULL,&info);

  return info.name;
}

////////////////////////////////////////////////////////
/**
 * Returns the number of ms elapsed since the first
 * execution of this function
 */
float ms_time_since_start(void) {
  static RTIME init_time=0;

  if(init_time==0) init_time=rt_timer_read();

  return (rt_timer_read()-init_time)/1000000.;
}
```

```
59
60 #ifdef USE_MUTEX
61 ////////////////////////////////////////////////////////
62 void acquire_resource(void) {
63     rt_mutex_acquire(&bus_mutex, TM_INFINITE);
64 }
65
66 ////////////////////////////////////////////////////////
67 void release_resource(void) {
68     rt_mutex_release(&bus_mutex);
69 }
70 #else
71 ////////////////////////////////////////////////////////
72 void acquire_resource(void) {
73     rt_sem_p(&bus_sem,TM_INFINITE);
74 }
75
76 ////////////////////////////////////////////////////////
77 void release_resource(void) {
78     rt_sem_v(&bus_sem);
79 }
80 #endif
81
82 ////////////////////////////////////////////////////////
83 void busy_wait(RTIME time) {
84     // Get info about the currently running task
85     static RT_TASK_INFO info;
86     rt_task_inquire(NULL,&info);
87
88     // Get execution time when starting
89     RTIME start = info.stat.xtime;
90
91     // While execution time is inferior to waiting time, burn cpu cycles
92     while (info.stat.xtime - start < time) {
93         // Update info about the task
94         rt_task_inquire(NULL,&info);
95     }
96 }
97
98 ////////////////////////////////////////////////////////
99 /**
100  * Simulate the execution of a task with possible bus 1553
101  * acquirement and fake execution time (done with a busy wait).
102  */
103 void rt_task(void *cookie) {
104     // Get the task descriptor given as argument.
105     struct task_descriptor* params = (struct task_descriptor*)cookie;
106
107     // Print information about executing task
108     rt_printf("started task %s, period %ims, duration %ims, use resource
        %i\n",rt_task_name(),(int)(params->period/1000000),(int)(params->duration/1000000),params->use_resource)
109     // Wait start for semaphore to be available
110     rt_sem_p(&start_sem,TM_INFINITE);
111
112     // Infinite loop
113     while(1) {
114         // If the task needs the bus, aqcuire it
115         if(params->use_resource) acquire_resource();
116         // Print the name of running task
117         rt_printf("doing %s (%.3f ms)\n",rt_task_name(), ms_time_since_start());
118         // Fake program execution duration
```

```
119         busy_wait(params->duration);
120         // Print the fact that execution is done
121         rt_printf("doing %s ok (%.3f ms)\n",rt_task_name(), ms_time_since_start());
122         // Release the bus if acquired
123         if(params->use_resource) release_resource();
124         // Wait for the next period of the task
125         rt_task_wait_period(NULL);
126     }
127 }
128
129 void ordo_bus_task(void *cookie) {
130     // Get the task descriptor given as argument.
131     struct task_descriptor* params = (struct task_descriptor*)cookie;
132
133     // Print information about executing task
134     rt_printf("started task %s, period %ims, duration %ims, use resource
            %i\n",rt_task_name(),(int)(params->period/1000000),(int)(params->duration/1000000),params->use_resource)
135     // Wait for start semaphore to be available
136     rt_sem_p(&start_sem,TM_INFINITE);
137
138     RT_SEM_INFO security_sem_info;
139
140     // Infinite loop
141     while(1) {
142         // Print the name of running task
143         rt_printf("doing %s (%.3f ms)\n",rt_task_name(), ms_time_since_start());
144         // Get info about security semaphore
145         rt_sem_inquire(&security_sem, &security_sem_info);
146         // If count is different from 1, task did not run properly
147         if (security_sem_info.count != 1) {
148             rt_printf("security triggered! Security semaphore count: %ld\n",
                    security_sem_info.count);
149             return;
150         }
151         // Put the semaphore to 0
152         rt_sem_p(&security_sem, TM_INFINITE);
153
154         // Fake program execution duration
155         busy_wait(params->duration);
156         // Print the fact that execution is done
157         rt_printf("doing %s ok (%.3f ms)\n",rt_task_name(), ms_time_since_start());
158         // Wait for the next period of the task
159         rt_task_wait_period(NULL);
160     }
161 }
162
163 void distrib_donnees_task(void *cookie) {
164     // Get the task descriptor given as argument.
165     struct task_descriptor* params = (struct task_descriptor*)cookie;
166
167     // Print information about executing task
168     rt_printf("started task %s, period %ims, duration %ims, use resource
            %i\n",rt_task_name(),(int)(params->period/1000000),(int)(params->duration/1000000),params->use_resource)
169     // Wait start for semaphore to be available
170     rt_sem_p(&start_sem,TM_INFINITE);
171
172     // Infinite loop
173     while(1) {
174         // If the task needs the bus, aqcuire it
175         if(params->use_resource) acquire_resource();
176         // Print the name of running task
```

```
177         rt_printf("doing %s (%.3f ms)\n",rt_task_name(), ms_time_since_start());
178         // Fake program execution duration
179         busy_wait(params->duration);
180         // Release the security semaphore to prove execution
181         rt_sem_v(&security_sem);
182         // Print the fact that execution is done
183         rt_printf("doing %s ok (%.3f ms)\n",rt_task_name(), ms_time_since_start());
184         // Release the bus if acquired
185         if(params->use_resource) release_resource();
186         // Wait for the next period of the task
187         rt_task_wait_period(NULL);
188     }
189 }
190 
191 ////////////////////////////////////////////////////////
192 /**
193  * Create a task, make it periodic, and start it.
194  */
195 int create_and_start_rt_task(struct task_descriptor* desc,char* name){
196     // Create the task
197     int status=rt_task_create(&desc->task,name,TASK_STKSZ,desc->priority,TASK_MODE);
198     // Ensure that creation went well
199     if(status!=0) {
200         printf("error creating task %s\n",name);
201         return status;
202     }
203 
204     // Make the task periodic with first release point now
205     status=rt_task_set_periodic(&desc->task,TM_NOW,desc->period);
206     // Ensure that setting went well
207     if(status!=0) {
208         printf("error setting period on task %s\n",name);
209         return status;
210     }
211 
212     // Start the task and give the task_descriptor structure as argument
213     status=rt_task_start(&desc->task,desc->task_function,desc);
214     // Ensure that starting went well
215     if(status!=0) {
216         printf("error starting task %s\n",name);
217     }
218     return status;
219 }
220 
221 ////////////////////////////////////////////////////////
222 int main(void) {
223     // Create start_sem to ensure proper priority task starting
224     if(rt_sem_create(&start_sem,"start_semaphore",0,S_PRIO)!=0) {
225         printf("error creating start_semaphore\n");
226         return EXIT_FAILURE;
227     }
228     // Create bus_sem
229     if(rt_sem_create(&bus_sem,"bus_semaphore",1,S_PRIO)!=0) {
230         printf("error creating bus_semaphore\n");
231         rt_sem_delete(&start_sem);
232         return EXIT_FAILURE;
233     }
234     // Create security_sem
235     if(rt_sem_create(&security_sem,"security_semaphore",1,S_PRIO)!=0) {
236         printf("error creating security_semaphore\n");
237         rt_sem_delete(&start_sem);
```

```
238         rt_sem_delete(&bus_sem);
239         return EXIT_FAILURE;
240     }
241     // Create bus_mutex
242     if (rt_mutex_create(&bus_mutex, "bus_mutex") != 0) {
243         printf("error creating bus_mutex\n");
244         rt_sem_delete(&start_sem);
245         rt_sem_delete(&bus_sem);
246         rt_sem_delete(&security_sem);
247         return EXIT_FAILURE;
248     }
249 
250     // Print start time of the main program
251     rt_printf("started main program at %.3fms\n",ms_time_since_start());
252 
253     // Create ORDO_BUS task descriptor
254     task_descriptor ORDO_BUS;
255     ORDO_BUS.duration = 25000000;
256     ORDO_BUS.period = 125000000;
257     ORDO_BUS.priority = 7;
258     ORDO_BUS.use_resource = false;
259     ORDO_BUS.task_function = &ordo_bus_task;
260     // Create and start the task
261     int status = create_and_start_rt_task(&ORDO_BUS, "ORDO_BUS");
262     // Ensure that everything went well
263     if(status!=0) {
264         printf("error starting ORDO_BUS\n");
265         rt_sem_delete(&start_sem);
266         rt_sem_delete(&bus_sem);
267         rt_sem_delete(&security_sem);
268         rt_mutex_delete(&bus_mutex);
269         return EXIT_FAILURE;
270     }
271 
272     // Create DISTRIB_DONNES task descriptor
273     task_descriptor DISTRIB_DONNES;
274     DISTRIB_DONNES.duration = 25000000;
275     DISTRIB_DONNES.period = 125000000;
276     DISTRIB_DONNES.priority = 6;
277     DISTRIB_DONNES.use_resource = true;
278     DISTRIB_DONNES.task_function = &distrib_donnees_task;
279     // Create and start the task
280     status = create_and_start_rt_task(&DISTRIB_DONNES, "DISTRIB_DONNES");
281     // Ensure that everything went well
282     if(status!=0) {
283         printf("error starting DISTRIB_DONNES\n");
284         rt_sem_delete(&start_sem);
285         rt_sem_delete(&bus_sem);
286         rt_sem_delete(&security_sem);
287         rt_mutex_delete(&bus_mutex);
288         rt_task_delete(&ORDO_BUS.task);
289         return EXIT_FAILURE;
290     }
291 
292     // Create PILOTAGE task descriptor
293     task_descriptor PILOTAGE;
294     PILOTAGE.duration = 25000000;
295     PILOTAGE.period = 250000000;
296     PILOTAGE.priority = 5;
297     PILOTAGE.use_resource = true;
298     PILOTAGE.task_function = &rt_task;
```

```
299     // Create and start the task
300     status = create_and_start_rt_task(&PILOTAGE, "PILOTAGE");
301     // Ensure that everything went well
302     if(status!=0) {
303         printf("error starting PILOTAGE\n");
304         rt_sem_delete(&start_sem);
305         rt_sem_delete(&bus_sem);
306         rt_sem_delete(&security_sem);
307         rt_mutex_delete(&bus_mutex);
308         rt_task_delete(&ORDO_BUS.task);
309         rt_task_delete(&DISTRIB_DONNES.task);
310         return EXIT_FAILURE;
311     }
312 
313     // Create RADIO task descriptor
314     task_descriptor RADIO;
315     RADIO.duration = 25000000;
316     RADIO.period = 250000000;
317     RADIO.priority = 4;
318     RADIO.use_resource = false;
319     RADIO.task_function = &rt_task;
320     // Create and start the task
321     status = create_and_start_rt_task(&RADIO, "RADIO");
322     // Ensure that everything went well
323     if(status!=0) {
324         printf("error starting RADIO\n");
325         rt_sem_delete(&start_sem);
326         rt_sem_delete(&bus_sem);
327         rt_sem_delete(&security_sem);
328         rt_mutex_delete(&bus_mutex);
329         rt_task_delete(&ORDO_BUS.task);
330         rt_task_delete(&DISTRIB_DONNES.task);
331         rt_task_delete(&PILOTAGE.task);
332         return EXIT_FAILURE;
333     }
334 
335     // Create CAMERA task descriptor
336     task_descriptor CAMERA;
337     CAMERA.duration = 25000000;
338     CAMERA.period = 250000000;
339     CAMERA.priority = 3;
340     CAMERA.use_resource = false;
341     CAMERA.task_function = &rt_task;
342     // Create and start the task
343     status = create_and_start_rt_task(&CAMERA, "CAMERA");
344     // Ensure that everything went well
345     if(status!=0) {
346         printf("error starting CAMERA\n");
347         rt_sem_delete(&start_sem);
348         rt_sem_delete(&bus_sem);
349         rt_sem_delete(&security_sem);
350         rt_mutex_delete(&bus_mutex);
351         rt_task_delete(&ORDO_BUS.task);
352         rt_task_delete(&DISTRIB_DONNES.task);
353         rt_task_delete(&PILOTAGE.task);
354         rt_task_delete(&RADIO.task);
355         return EXIT_FAILURE;
356     }
357 
358     // Create MESURES task descriptor
359     task_descriptor MESURES;
```

```
360     MESURES.duration = 50000000;
361     MESURES.period = 5000000000;
362     MESURES.priority = 2;
363     MESURES.use_resource = true;
364     MESURES.task_function = &rt_task;
365     // Create and start the task
366     status = create_and_start_rt_task(&MESURES, "MESURES");
367     // Ensure that everything went well
368     if(status!=0) {
369         printf("error starting MESURES\n");
370         rt_sem_delete(&start_sem);
371         rt_sem_delete(&bus_sem);
372         rt_sem_delete(&security_sem);
373         rt_mutex_delete(&bus_mutex);
374         rt_task_delete(&ORDO_BUS.task);
375         rt_task_delete(&DISTRIB_DONNES.task);
376         rt_task_delete(&PILOTAGE.task);
377         rt_task_delete(&RADIO.task);
378         rt_task_delete(&CAMERA.task);
379         return EXIT_FAILURE;
380     }
381 
382     // Create METEO task descriptor
383     task_descriptor METEO;
384     METEO.duration = 60000000;
385     METEO.period = 5000000000;
386     METEO.priority = 1;
387     METEO.use_resource = true;
388     METEO.task_function = &rt_task;
389     // Create and start the task
390     status = create_and_start_rt_task(&METEO, "METEO");
391     // Ensure that everything went well
392     if(status!=0) {
393         printf("error starting METEO\n");
394         rt_sem_delete(&start_sem);
395         rt_sem_delete(&bus_sem);
396         rt_sem_delete(&security_sem);
397         rt_mutex_delete(&bus_mutex);
398         rt_task_delete(&ORDO_BUS.task);
399         rt_task_delete(&DISTRIB_DONNES.task);
400         rt_task_delete(&PILOTAGE.task);
401         rt_task_delete(&RADIO.task);
402         rt_task_delete(&CAMERA.task);
403         rt_task_delete(&MESURES.task);
404         return EXIT_FAILURE;
405     }
406 
407     // Release the semaphore
408     rt_sem_broadcast(&start_sem);
409 
410     // Wait for tasks to finish
411     rt_task_join(&ORDO_BUS.task);
412 
413     // Delete the start semaphore
414     rt_sem_delete(&start_sem);
415     rt_sem_delete(&bus_sem);
416     rt_sem_delete(&security_sem);
417     rt_mutex_delete(&bus_mutex);
418     rt_task_delete(&ORDO_BUS.task);
419     rt_task_delete(&DISTRIB_DONNES.task);
420     rt_task_delete(&PILOTAGE.task);
```

```
421     rt_task_delete(&RADIO.task);
422     rt_task_delete(&CAMERA.task);
423     rt_task_delete(&MESURES.task);
424     rt_task_delete(&METEO.task);
425 
426     // Exit the program
427     return EXIT_SUCCESS;
428 }
```