

Compte-rendu TP "Linux Xenomai - Partie 1"

Mewen Michel et Sander Ricou - MI11 UTC

Exercice 1 : Tâches

*Dans cet exercice, nous allons créer une application simple de type « Hello World » sous Xenomai. L'objectif principal sera de manipuler les tâches temps réel et d'analyser leur fonctionnement.

Reprenez le code du programme « Hello World » du TP sur Linux embarqué et adaptez le pour afficher le message à intervalle régulier (une fois par seconde par exemple). Cross-compilez ce code, téléchargez-le et exécutez-le sur la carte Devkit8600.*

```
#include <stdio.h>
#include <time.h>

int main() {
    while(1) {
        printf("Hello World\n");
        sleep(1);
    }
    return 1;
}
```

On le compile puis l'exécute comme dans le TP précédent:

```
source /opt/poky/1.7.3/environment-setup-armv7a-vfp-neon-poky-linux-gnueabi
unset LDFLAGS
arm-poky-linux-gnueabi-gcc main.c -o main
scp main root@192.168.7.2:/home/root/
```

Question 1.1 Ce code s'exécute-t-il de façon temps réel ? Comment le vérifier (regarder le fichier de statistiques de Xenomai)?

Non ce code ne s'exécute pas en temps réel, en effet en regardant le fichier de statistiques de Xenomai

(affichant les tâches temps réel), on ne voit pas pas notre tâche "helloworld".

Ce qui est normal car nous n'avons absolument rien fait pour que celle ci le soit.

fichier de statistiques:

```
root@devkit8600-xenomai:~# cat /proc/xenomai/stat
CPU  PID    MSW      CSW      PF    STAT      %CPU  NAME
 0    0       0        0       0    00500080  100.0  ROOT
 0    0       0    22009    0    00000000   0.0  IRQ68: [timer]
```

Créez maintenant une tâche temps réel Xenomai (avec l'API native) qui se chargera d'exécuter le printf et le sleep. Afin de cross-compiler ce programme, il faut indiquer à gcc où se trouvent les headers de Xenomai (option -I) et quelles bibliothèques utiliser (option -l). Vous aurez besoin des

librairies native et xenomai.

Question 1.2 : Donnez la ligne de commande utilisée pour la cross-compilation ainsi que le code du programme.

Ligne de commande utilisée pour la cross-compilation:

```
arm-poky-linux-gnueabi-gcc main_rt.c -o main_rt -I
/opt/poky/1.7.3/sysroots/armv7a-vfp-neon-poky-linux-
gnueabi/usr/include/xenomai/ -l native -l xenomai
```

Code du programme:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <native/task.h>
#define TASK_PRI0 99 /* Highest RT priority */
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */

RT_TASK task_desc;

void task_body()
{
    while (1) {
        printf("Hello world!\n");
        sleep(1);
    }
}

int main (int argc, char *argv[])
{
    int err;
    mlockall(MCL_CURRENT|MCL_FUTURE);
    err = rt_task_create(&task_desc, "hello",
TASK_STKSZ, TASK_PRI0, TASK_MODE);

    if (!err)
        rt_task_start(&task_desc, &task_body, NULL);

    getchar();

    return 0;
}

void cleanup (void)
{
    rt_task_delete(&task_desc);
}
```

Question 1.3 : Le code est-il vraiment temps réel et pourquoi? Que donne le fichier de statistiques de Xenomai (ne pas interpréter ce résultat pour le moment)?

Le code n'est toujours pas temps réel, nous avons une tâche xenomai mais celle ci ne s'exécute pas encore en temps réel.

En effet la tâche n'utilise pas encore des fonctions temps réel de xenomai, notamment pour le sleep et le printf.

Fichier de statistiques:

```
cat /proc/xenomai/stat
CPU  PID    MSW    CSW    PF  STAT    %CPU  NAME
 0    0      0      330323  0  00500080 100.0  ROOT
 0  1337    1        1    0  00300380  0.0  hello
 0    0      0      275656  0  00000000  0.0  IRQ68: [timer]
```

Remplacez maintenant l'appel à la fonction sleep par son équivalent sous Xenomai.

Question 1.4 : *Donnez le code du programme et les statistiques de Xenomai (ne pas interpréter ce résultat pour le moment).*

code :

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <native/task.h>
#define TASK_PRI0 99 /* Highest RT priority */
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */

RT_TASK task_desc;

void task_body()
{
    while (1) {
        printf("Hello world!\n");
        rt_task_sleep(1000000000);
    }
}

int main (int argc, char *argv[])
{
    int err;
    mlockall(MCL_CURRENT|MCL_FUTURE);
    err = rt_task_create(&task_desc, "hello",
TASK_STKSZ, TASK_PRI0, TASK_MODE);

    if (!err)
        rt_task_start(&task_desc, &task_body, NULL);

    getchar();

    return 0;
}

void cleanup (void)
{
    rt_task_delete(&task_desc);
}

```

Fichier statistiques:

```

cat /proc/xenomai/stat
CPU  PID  MSW    CSW    PF  STAT    %CPU  NAME
 0   0    0      330289  0   00500080 100.0  ROOT
 0  1327   7       14    0   00300184  0.0   hello
 0   0    0      272515  0   00000000  0.0   IRQ68: [timer]

```

Remplacez maintenant l'appel à la fonction printf par son équivalent sous Xenomai.

Question 1.5 : *Donnez le code du programme et les statistiques de Xenomai. Interprétez maintenant les résultats des différentes statistiques que vous avez relevées.*

Code:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <native/task.h>
#define TASK_PRI0 99 /* Highest RT priority */
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */

RT_TASK task_desc;

void task_body()
{
    while (1) {
        rt_printf("Hello world!\n");
        rt_task_sleep(1000000);
    }
}

int main (int argc, char *argv[])
{
    // Perform auto-init of rt_print buffers if the task doesn't do so
    rt_print_auto_init(1);

    int err;
    mlockall(MCL_CURRENT|MCL_FUTURE);
    err = rt_task_create(&task_desc, "hello",
TASK_STKSZ, TASK_PRI0, TASK_MODE);

    if (!err)
        rt_task_start(&task_desc, &task_body, NULL);

    getchar();

    return 0;
}

void cleanup (void)
{
    rt_task_delete(&task_desc);
}

```

Fichier statistiques:

```

root@devkit8600-xenomai:~# cat /proc/xenomai/stat
CPU  PID    MSW     CSW     PF  STAT    %CPU  NAME
 0    0        0      330197  0   00500080 100.0  ROOT
 0   1315     0         6      0   00300184  0.0   hello
 0    0        0     269491  0   00000000  0.0   IRQ68: [timer]

```

<<<<<<< HEAD:TP-5to9/TP7.md

Nous avons observé trois cas différents:

- *Cas 1:* Aucune fonction en temps réel utilisée
- *Cas 2:* Seulement la moitié des fonctions en temps réel
- *Cas 3:* Que des fonctions en temps réel utilisées.

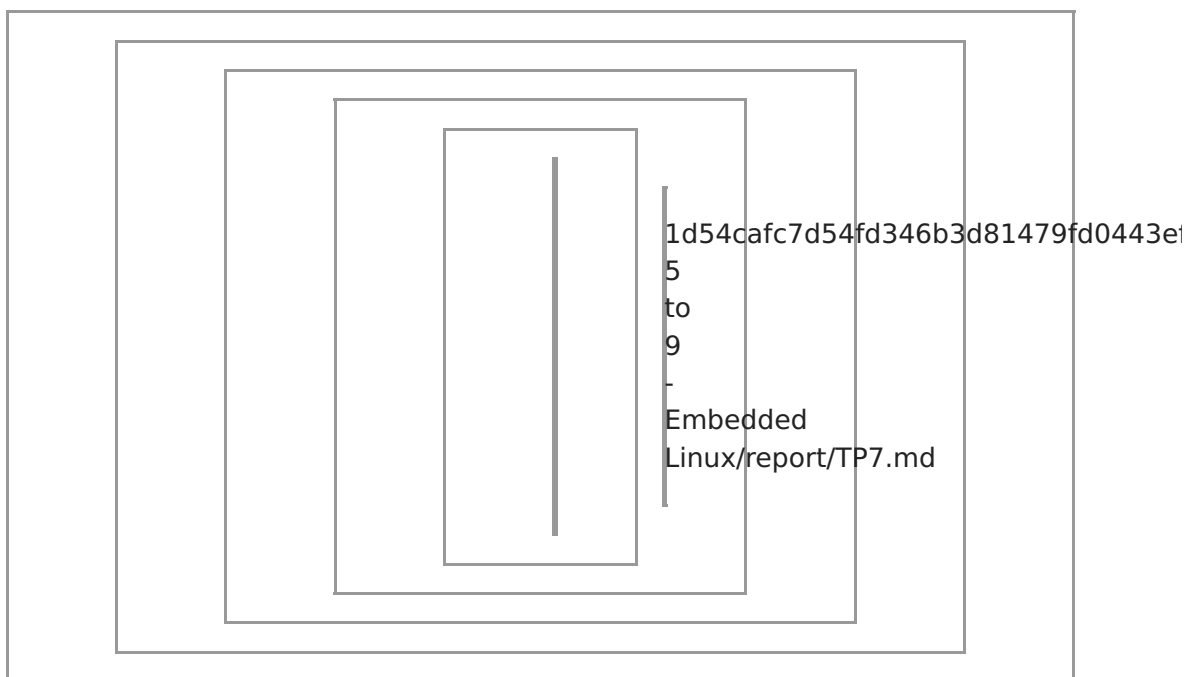
Nous allons étudier le nombre de changements de mode (MSW), qui mettent en évidence le passage d'un mode normal en un mode temps réel ou inversement.

1. On passe une fois en non temps réel et on y reste car aucune fonction temps réel n'est appelée, un seul changement de mode (MSW).
2. On passe d'un état temps réel à un non temps réel en alterné. car seul le sleep et en temps réel. Le nombre de changements de mode grandit avec le temps.
3. Pas de changement de mode car seulement tâche utilisant des fonctions temps réel, on reste donc dans le mode temps réel.

Exercice 2 : Synchronisation

=====

Exercice 2 : Synchronisation



Créez un programme lançant deux tâches Xenomai qui afficheront chacune une partie d'un message (chaque tâche ne doit rien faire d'autre).

Question 2.1 : Donnez le code du programme et le résultat.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <native/task.h>
#define TASK_PRI0 90 /* Highest RT priority */
```

```

#define TASK_PRI02 99
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */

RT_TASK task_desc;
RT_TASK task_desc2;

void task_body1()
{
    while (1) {
        rt_printf("Hello\n");
        rt_task_sleep(1000000000);
    }
}

void task_body2()
{
    while (1) {
        rt_printf(" world!\n");
        rt_task_sleep(1000000000);
    }
}

int main (int argc, char *argv[])
{
    // Perform auto-init of rt_print buffers if the task doesn't do so
    rt_print_auto_init(1);

    mlockall(MCL_CURRENT|MCL_FUTURE);

    int err;
    err = rt_task_create(&task_desc, "hello", TASK_STKSZ, TASK_PRI02, TASK_MODE);

    if (!err)
        rt_task_start(&task_desc, &task_body1, NULL);

    int err2;
    err2 = rt_task_create(&task_desc2, "hello2",
TASK_STKSZ, TASK_PRI0, TASK_MODE);

    if (!err)
        rt_task_start(&task_desc2, &task_body2, NULL);

    getchar();

    return 0;
}

void cleanup (void)
{

```

```
rt_task_delete(&task_desc);  
}
```

Résultat:

```
./main_2tasks  
Hello world!  
Hello world!  
Hello world!  
Hello world!
```

Question 2.2 : *Quelle est l'influence de la priorité des tâches ? Comment faire pour afficher le message dans l'ordre ou le désordre ? Justifiez.*

Il n'y a aucune influence de la priorité des tâches, car celles ci attendent 1s après leur printf ce qui laisse largement le temps à la seconde tâche d'effectuer son travail à chaque fois.

On inverse la création et le démarrage des tâches. Donc dès la création de la première, son body sera exécuté et le message afficher dans le désordre si les tâches sont créées dans le désordre.

Afin d'améliorer le comportement de votre programme, utilisez un sémaphore qui bloquera chacune des tâches dès le début. Les tâches devront être libérées après avoir été toutes lancées.

Question 2.3 : *A quelle valeur faut-il initialiser le sémaphore ?*

Le sémaphore doit être initialisé avec la valeur 0 (nombre de jeton disponible), car nous voulons bloquer toutes les tâches et seulement les libérer avec le broadcast.

Question 2.4 : *Quelle est l'influence du paramètre mode utilisé à la création du sémaphore ?*

Le paramètre mode influence la manière, l'ordre dans lequel les tâches en attente au niveau du sémaphore seront traitées.

La documentation:

```
mode The semaphore creation mode. The following flags can be OR'ed into this  
bitmask, each of them affecting the new semaphore:  
S_FIFO makes tasks pend in FIFO order on the semaphore.  
S_PRIO makes tasks pend in priority order on the semaphore.  
S_PULSE causes the semaphore to behave in "pulse" mode.  
In this mode, the V (signal) operation attempts to release a single waiter each  
time it is called, but without incrementing the semaphore count if no waiter is  
pending.  
For this reason, the semaphore count in pulse mode remains zero.
```

En mettant S_FIFO, la première arrivée sera traitée d'abord.

Nous avons choisi d'utiliser S_PRIO, ce qui nous permet de mettre une priorité plus faible pour la tâche "world" que la tâche "Hello", et ainsi garantir que le world sera affiché en dernier.

Question 2.5 : *Donnez le code du programme et le résultat. Expliquez le fonctionnement du programme.*

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <native/task.h>
#include <native/sem.h>
#define TASK_PRI0 99 /* Highest RT priority */
#define TASK_PRI02 90
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */

RT_TASK task_desc;
RT_TASK task_desc2;

/* semaphore for shared static local variable in thread_func()
 * see the explanation in thread_func() comments
 */
static RT_SEM sem;

void task_body1()
{
    rt_sem_p(&sem, TM_INFINITE);

    while (1) {
        rt_printf("Hello \n");
        rt_task_sleep(1000000000);
    }
}

void task_body2()
{
    rt_sem_p(&sem, TM_INFINITE);

    while (1) {
        rt_printf(" world!\n");
        rt_task_sleep(1000000000);
    }
}

int main (int argc, char *argv[])
{
    // Perform auto-init of rt_print buffers if the task doesn't do so
    rt_print_auto_init(1);

    mlockall(MCL_CURRENT|MCL_FUTURE);

    // int sem_init(sem_t *sem, int pshared, unsigned int initial_value)
    if (rt_sem_create(&sem, "wait plop", 0, S_PRI0) == -1){
```

```

    printf("rt_sem_create: failed: %s\n", strerror(errno));
}

int err2;
err2 = rt_task_create(&task_desc2, "world",
TASK_STKSZ, TASK_PRI02, TASK_MODE);

if (!err2)
    rt_task_start(&task_desc2, &task_body2, NULL);

int err;
err = rt_task_create(&task_desc, "hello", TASK_STKSZ, TASK_PRI0, TASK_MODE);

if (!err)
    rt_task_start(&task_desc, &task_body1, NULL);

rt_sem_broadcast(&sem);

getchar();

return 0;
}

void cleanup (void)
{
    rt_task_delete(&task_desc);
}

```

Résultat:

```

./main_2tasks
Hello
  world!
Hello
  world!
Hello
  world!

```

Ça s'affiche dans le bon sens.

Fonctionnement:

On crée un sémaphore avec 0 jeton et le mode S_PRIO, ensuite nous créons nos deux tâches qui vont demander l'accès au sémaphore (rt_sem_p(&sem, TM_INFINITE)).

Celles ci sont donc bloquées, ce n'est que lorsque l'on utilise la fonction broadcast du sémaphore (dans main, rt_sem_broadcast(&sem)) que les tâches seront traitées, et dû au mode S_PRIO du sémaphore, elles seront traitées selon leur priorité.

Nous allons maintenant faire l'affichage du message en boucle (une fois par seconde), grâce à une troisième tâche qui servira de métronome :

- modifiez les deux tâches d'affichage pour faire l'affichage en boucle
- ajoutez la troisième tâche qui synchronisera les deux autres et réalisera l'attente

Question 2.6 : *Donnez le code du programme et le résultat. Expliquez le fonctionnement du programme.*

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <native/task.h>
#include <native/sem.h>
#define TASK_PRI0 95
#define TASK_PRI02 90
#define TASK_PRI03 99 /* Highest RT priority */
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */

RT_TASK task_desc;
RT_TASK task_desc2;
RT_TASK task_desc3;

/* semaphore for shared static local variable in thread_func()
 * see the explanation in thread_func() comments
 */
static RT_SEM sem;

void task_body1()
{
    while (1) {
        rt_sem_p(&sem, TM_INFINITE);
        rt_printf("Hello \n");
    }
}

void task_body2()
{
    while (1) {
        rt_sem_p(&sem, TM_INFINITE);
        rt_printf(" world!\n");
    }
}

void task_body3()
{
    while(1) {
        rt_sem_broadcast(&sem);
        rt_task_sleep(1000000000);
    }
}

int main (int argc, char *argv[])
```

```

{
    // Perform auto-init of rt_print buffers if the task doesn't do so
    rt_print_auto_init(1);

    mlockall(MCL_CURRENT|MCL_FUTURE);

    // int sem_init(sem_t *sem, int pshared, unsigned int initial_value)
    if (rt_sem_create(&sem, "wait plop", 0, S_PRIO) == -1){
        printf("rt_sem_create: failed: %s\n", strerror(errno));
    }

    int err2;
    err2 = rt_task_create(&task_desc2, "world",
TASK_STKSZ, TASK_PRIO2, TASK_MODE);

    if (!err2)
        rt_task_start(&task_desc2, &task_body2, NULL);

    int err;
    err = rt_task_create(&task_desc, "hello", TASK_STKSZ, TASK_PRIO, TASK_MODE);

    if (!err)
        rt_task_start(&task_desc, &task_body1, NULL);

    int err3;
    err3 = rt_task_create(&task_desc3, "timer",
TASK_STKSZ, TASK_PRIO3, TASK_MODE);

    if (!err3)
        rt_task_start(&task_desc3, &task_body3, NULL);

    getchar();

    return 0;
}

void cleanup (void)
{
    rt_task_delete(&task_desc);
}

```

Résultat:

```
./main_3tasks
Hello
  world!
Hello
  world!
Hello
  world!
Hello
  world!
```

Fonctionnement:

On garde le même principe sauf que cette fois ci la nouvelle tâche (Tâche 3) s'occupe de gérer le timing et de libérer les tâches pour leur traitement selon la priorité des tâches. Comme précédemment le sémaphore est créé dans le main et c'est donc la tâche 3 qui utilise la fonction `rt_sem_broadcast(&sem)` pour libérer les tâches.

Question 2.7 : *Regardez le fichier de statistiques et du scheduler de xenomai. Quelles sont les différentes informations ? Vous pouvez bloquer l'avancement de votre programme en utilisant la fonction `getchar()` par exemple.*

On bloque le timer (tâche 3) en mettant un '`getchar()`' dans son body.

On a le fichier de statistiques suivant:

```
cat /proc/xenomai/stat
CPU  PID      MSW      CSW      PF  STAT      %CPU  NAME
 0   0        0       330813   0   00500080  100.0  ROOT
 0  1537      0        6        0   00300182   0.0   world
 0  1538      0        6        0   00300182   0.0   hello
 0  1539      5       10        0   00300380   0.0   timer
 0   0        0       315735   0   00000000   0.0   IRQ68: [timer]
```

On voit un changement de mode (MSW) dans le timer à cause du '`getchar()`'.

et le fichier du scheduler:

```
cat /proc/xenomai/sched
CPU  PID      CLASS  PRI  TIMEOUT  TIMEBASE  STAT  NAME
 0   0       idle   -1   -        master    R     ROOT
 0  1537     rt     90   -        master    W     world
 0  1538     rt     95   -        master    W     hello
 0  1539     rt     99   -        master    X     timer
```

On remarque que le timer est bien bloqué dans son exécution (X, le `getchar` attend une entrée) et que les autres tâches attendent (W).

Si on met cela en lien avec le fichier de statistiques, on remarque également que 100% du CPU est pour la tâche "ROOT", ce qui est normale car c'est la seule en exécution ("R", running).

Exercice 3 : Latence

*Dans cet exercice, nous allons nous intéresser à la latence de Xenomai, et la comparer avec les résultats du TP précédent.

Il faut donc réécrire un programme réalisant 10 000 fois une attente de 1ms, en utilisant Xenomai.

Ajoutez la mesure des latences minimum, maximum et moyenne. Veillez à utiliser les fonctions Xenomai pour la prise de temps et à utiliser le fichier entête correspondant.*

Question 3.1 & 3.2 : *Donnez le code du programme et les résultats obtenus. Que pouvez vous en conclure ?*

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <native/task.h>
#include <native/timer.h>
#define TASK_PRI0 99 /* Highest RT priority */
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 200000 /* Stack size (use default one) */

RT_TASK task_desc;

void task_body()
{
    const int nbLap = 10000;
    RTIME tvRes;
    RTIME tvLap[nbLap + 1]; // array of ns instants
    RTIME intervals[nbLap + 1]; // array of ns intervals
    int i;
    unsigned long max = 0;
    unsigned long min = 1000000000; // 1s in ns
    unsigned long tmp;
    double mean;

    rt_printf("Start\n");

    tvLap[0] = rt_timer_read();

    // Mesures
    for (i = 1; i < nbLap; i++) {
        rt_task_sleep(1000000);
        tvLap[i] = rt_timer_read();
    }

    rt_printf("End\n");

    // Calculs
    for (i = 0; i < nbLap - 1; i++) {
        intervals[i] = tvLap[i+1] - tvLap[i];

        if (intervals[i] > max) {
            max = intervals[i];
        }
    }
}
```

```

    } else if (intervals[i] < min) {
        min = intervals[i];
    }
}

tvRes = tvLap[nbLap-1] - tvLap[0];
mean = ((double)tvRes) / nbLap;

rt_printf("Total %ld ns\n", tvRes);
rt_printf("Mean %f ns\n", mean);
rt_printf("Min %ld ns\n", min);
rt_printf("Max %ld ns\n", max);
}

int main (int argc, char *argv[])
{
    // Perform auto-init of rt_print buffers if the task doesn't do so
    rt_print_auto_init(1);

    int err;
    mlockall(MCL_CURRENT|MCL_FUTURE);
    err = rt_task_create(&task_desc, "benchmark",
TASK_STKSZ, TASK_PRIO, TASK_MODE);

    if (!err)
        rt_task_start(&task_desc, &task_body, NULL);

    getchar();

    return 0;
}

void cleanup (void)
{
    rt_task_delete(&task_desc);
}

```

Résultats:

```

./main_loop
Start
End
Total 1093572249 ns
Mean 1002316.676000 ns
Min 1000200 ns
Max 1033240 ns

```

Conclusion: L'exécution des attentes d'1ms est correcte, même le maximum est cette fois ci plus proche des 10s. Pour regarder effectivement si Xenomai a de la latence, il faudra charger le CPU.

Question 3.3 : Chargez le CPU et donnez les résultats obtenus. Que pouvez vous en conclure ?

Avec une charge de 600 workers:

```
root@devkit8600-xenomai:~# ./main_loop
Start
End
Total 1093577193 ns
Mean 1004788.620000 ns
Min 1000560 ns
Max 1039920 ns
```

La charge du CPU n'influe pas sur la latence de Xenomai. Il va exécuter ces processus temps réel sans être perturbé car leurs priorités sont plus élevées. Il ne permet pas le stress du CPU de s'effectuer.