FreeRTOS sur Texas Instruments Stellaris LM4F120

Texas Instruments propose une gamme de microcontrôleurs centrée sur le Cortex M4 nommée LM4F. Cette plateforme est supportée par `libopencm3` en se liant sur les bibliothèques fournies dans le sous-répertoire `lm4f` et nativement par la version de `qemu` émulant le comportement de processeurs ARM `qemu-system-arm`.

Nous avons vu à l'issue du premier semestre que nous pouvions calculer et tracer les motifs de structures fractales sur microcontroleurs, en particulier la fractale de Mandelbrot qui affiche la vitesse de divergence de la suite $z_{n+1} = z_n^2 + c$ avec $z_n, c \in \mathbb{C}$ initialisées avec $z_0 = c$. Dans le cas qui nous avait intéressé lors de la programmation en C ne traitant que d'entiers, puisque les microcontrôleurs ne supportent généralement pas le calcul sur les nombres à virgule flottante, nous avions balayé le plan complexe c séquentiellement.

Nous allons nous intéresser à la parallélisation du calcul. En effet, chaque sous ensemble de c peut se calculer indépendamment des voisins, et nous pouvons donc lancer une multitude de tâches en parallèles chargées chacune de calculer un petit bout de la fractale, visant ainsi à accélérer le calcul et réduire le temps nécessaire à aboutir à l'affichage des motifs.

Toutes les démonstrations expérimentales se feront avec la version officielle du paquet Debian/GNU Linux de qemu-system-arm en choisissant l'émulation de la machine `lm3s6965evb` au moyen de la commande `qemu-system-arm -machine lm3s6965evb -nographic -serial mon:stdio -kernel exec.elf` pour exécuter le binaire `exec.elf`. Noter que nous quittons `qemu` par `CTRL-a` puis `x` dans le terminal dans lequel l'émulateur a été lancé.

Nous allons déjà fournir l'infrastructure de travail avec une unique tâche dans un premier temps, en nous inpirant du `Makefile.stellaris` disponible à `https://github.com/jmfriedt/tp_freertos/tree/master/1basic`, le script d'édition de liens présentant l'organisation de la mémoire dans `https://github.com/jmfriedt/tp_freertos/tree/master/ld`, et de FreeRTOS dans sa version Long Term Support disponible à `https://github.com/FreeRTOS/FreeRTOS-LTS`

1. Proposer une façon de représenter un complexe en langage C dont la partie réelle et imaginaire sont définies par des entiers positifs ou négatifs.
   struct cpl {signed int re ; signed int im ;} ;
2. Proposer une architecture de programme permettant de lancer une tâche FreeRTOS qui reçoit en argument deux complexes et possède un espace mémoire libre pour récupérer le résultat de l'opération : déclaration des variables, définition de la fonction telle qu'elle sera appelée dans `main()` de FreeRTOS, incluant le passage d'argument lors de l'ordonnancement de la tâche.

   ```
   struct cpl c[3];

   void mandel(void* dummy)
   {struct cpl *c=(struct cpl*)dummy;
    struct cpl c1=c[0];
    struct cpl c2=c[1];
    struct cpl cres=c[2];
   ...
   }
   ```

   et lors du lancement de la tâche dans `main()` :

   ```
    static struct cpl c={{.re=1.,.im=2.},{.re=3.,.im=4.}};
    xTaskCreate( mandel, (const char*) "mandel",64,&c,1,NULL );
   ```

3. Nous désirons multiplier deux complexes entre eux, puis les sommer, et finalement calculer leur module, au moyen des fonctions que nous nommerons `cmul()`, `cadd()` et `cmod()` sous FreeRTOS. Agrémenter le programme précédent de ces trois fonctions qui prennent en argument deux complexes et calculent la somme et le produit de ces complexes, et une troisième fonction qui prend en argument un complexe et calcule son module.

```
struct cpl cmul(struct cpl c1,struct cpl c2)
{struct cpl cres;
 cres.re=c1.re*c2.re-c2.im*c2.im;
 cres.im=c1.re*c2.im+c1.im*c2.re;
 return(cres);
}

struct cpl cadd(struct cpl c1,struct cpl c2)
{struct cpl cres;
 cres.re=c1.re+c2.re;
 cres.im=c1.im+c2.im;
 return(cres);
}
```

4. Démontrer comment vous passez, depuis le `main()`, à une tâche FreeRTOS les deux arguments $a, b \in \mathbb{C}$ lors de son enregistrement auprès de l'ordonnanceur, et que vous stockez dans l'espace disponible le résultat du calcul $c = (a \times b) + a$. Pour ce faire, compiler le programme que vous avez commencé à écrire dans la séquence de questions précédentes, et exécutez le dans `qemu` pour aboutir au résultat. Détailler les étapes (compilation, exécution, résultat)

```
void mandel(void* dummy)
{struct cpl *c=(struct cpl*)dummy;
 struct cpl c1=c[0];
 struct cpl c2=c[1];
 struct cpl cres=c[2];
 cres=cmul(c1,c2);
 cres=cadd(cres,c1);
...
}
```

5. Vérifiez sur un exemple simple que chaque fonction (`cadd()`, `cmul()` et `cmod()` fournit le résultat attendu, par exemple en prenant $a = 1 + 2j$ et $b = 3 + 4j$ ($j^2 = -1$). Pour ce faire, exécuter le programme de démonstration compilé au moyen de FreeRTOS dans `qemu`.

   Voir solution 7 mais sur un exemple simple de a et b

Maintenant que nous avons une tâche capable d'effectuer les opérations, nous désirons étendre le calcul de la solution à $N > 1$ tâches exécutées en parallèle sur le microcontrôleur

6. Compléter la définition du paramètre fourni comme argument lors de l'enregistrement (question 4) de la tâche afin de lui passer l'indice de la tâche : ainsi, nous pourrons ordonner les tâches. Suite à cette modification, expliciter les modifications amenées au programme précédent pour passer le nouveau paramètre lors de son ordonnancement dans `main()`.

```
struct argument{struct cpl v[3];int indice;};

void mandel(void* dummy)
{struct argument *c=(struct argument*)dummy;
 struct cpl c1=c->v[0];
 struct cpl c2=c->v[1];
 struct cpl cres=c->v[2];
 int indice=c->indice;
...
```

et lors du lancement de la tâche dans main

```
 static struct argument c[N];
 xTaskCreate( mandel, (const char*) "mandel",64,&c[k],1,NULL );
```

7. Démontrer que la tâche k effectue la somme de $a = k + j \times k$ avec $b = 1 + 2j$ pour retrouver c, mais cette fois avec une valeur de a qui dépend de l'indice de la tâche. Ainsi, nous pourrons calculer des segments différents de la fractale de Mandelbrot en fonction de la valeur de k. Ici encore, l'exécution dans `qemu()` permet de visualiser le résultat.

```
$ qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -vga none -net none \
  -serial mon:stdio -kernel output/main.bin
4Hello: 12+44=56
0Hello: 12+00=12
1Hello: 12+11=23
2Hello: 12+22=34
3Hello: 12+33=45
```

Le calcul de chaque segment de la fractale est indépendant de ses voisins et l'ordre n'importe pas. Supposons que nous ayons un problème dans lequel l'ordre des solutions importe, par exemple parce que chaque calcul dépend du résultat de la tâche précédente.

8. Compléter le programme précédent avec les éléments nécessaires pour garantir l'ordre d'exécution, *i.e.* que la tâche $k + 1$ ne s'exécute qu'une fois la tâche k achevée, pour k allant de 0 à $N = 5$.
   Nous ajoutons N mutex pour garantir que chaque tâche libère le mutex qui bloque son successeur lorsque son calcul est achevé :

```
#include "semphr.h"
#define N 5

xSemaphoreHandle xMutex[N-1];

void mandel(void* dummy)
{struct argument *c=(struct argument*)dummy;
 struct cpl c1=c->v[0];
 struct cpl c2=c->v[1];
 struct cpl cres=c->v[2];
 int indice=c->indice;
 char resultat[32];
 if (indice!=0) xSemaphoreTake(xMutex[indice-1],500/portTICK_RATE_MS);
...
  xSemaphoreGive(xMutex[indice]);
}

int main(void){
 int k;
 static struct argument c[N];
 for (k=0;k<N-1;k++)
    {xMutex[k] = xSemaphoreCreateMutex(); xSemaphoreTake(xMutex[k],500/portTICK_RATE_MS);}
...
}
```

   permet bloquer toutes les tâches sauf celle d'indice 0, puis chaque tâche débloque la suivante pour donner

```
$ qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -vga none -net none \
  -serial mon:stdio -kernel output/main.bin
0Hello: 12+00=12
1Hello: 12+11=23
2Hello: 12+22=34
3Hello: 12+33=45
4Hello: 12+44=56
```

9. Compte tenue de la documentation technique du LM4F120 à `https://www.ti.com/lit/ds/symlink/tm4c1233h6pm.pdf?ts=1616854593130&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FTM4C1233H6PM` et en particulier la description de son cœur de calcul en section 2, quel bénéfice attendez vous de cette parallélisation ? Argumenter en justifiant les affirmations.

   Aucun bénéfice puisqu'il n'y a que un processeur pour effectuer les calculs et nous ne ferons que perdre du temps avec l'ordonnanceur sans bénéficier d'aucune parallélélisation matérielle qui ne saurait être atteinte qu'en présence d'au moins un second cœur de processeur.

Nous désirons maintenant reproduire ces résultats sur Red Pitaya, System-On-Chip Zynq de Xilinx muni de deux cœurs de calcul ARM A9 aux côtés du FPGA dont nous ne nous servirons pas ici. GNU/Linux est installé et s'exécute sur ce système d'exploitation embarqué. La chaîne de compilation croisée pour la Red Pitaya se trouve dans l'arborescence Buildroot de `/home/jmfriedt/buildroot-2020.08.1_RPi` et l'adresse IP de la Red Pitaya est 192.168.2.200.

10. Reprendre le programme FreeRTOS de la question 6 et le modifier pour exécuter N threads en parallèle, équivalent sous GNU/Linux des tâches de FreeRTOS, sur la Red Pitaya. Fournir le programme et son mode de compilation pour être exécutable sur Red Pitaya : démontrer son bon fonctionnement sur cette plateforme.

    `pthread_create() et pthread_join() au lieu de xTaskCreate()`

11. Remettre en place les mécanismes d'ordonnancement de la question 8 vus plus haut pour garantir que le `thread` d'indice 0 s'exécute en premier et $N-1$ en dernier sur la Red Pitaya : quel est-il, fournir le programme et son mécanisme de compilation.

    Mutex de pthreads sous GNU/Linux

12. Quel bénéfice attendez vous de cette parallélisation ? Argumenter en justifiant les affirmations.

    Cette fois nous avons deux cœurs de calcul donc sous réserve que le système d'exploitation n'occupe pas trop de ressources, nous pouvons espérer diviser par deux le temps d'exécution.

13. Conclure en compilant le même programme sur PC (comment ? quel résultat ?) et développer le même argumentaire concernant un PC de la salle 235B.

```
$ cat /proc/cpuinfo  | grep cores
cpu cores       : 4
```