Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2: Travail écrit no 3. 3,4

Nom : Zambon **Prénom :** Yannick **Classe : T-2/I-2** **Date : 26.04.2018**

8

Problème n° 1 (interfaçage C - assembleur)

a) Lors de l'implémentation de fonction réentrante, il est interdit d'utiliser des variables globales non-constantes. Argumentez cette règle.

Si ce n'est pas le cas, cela peut poser des problèmes de concurrence.

Et, mais lesquels ?

0,1 accès à une donnée partagée

b) Codez en assembleur la fonction **process** ci-dessous. Note : les « **int** » ont 32 bits.

```
typedef long (*oper_t) (int p1, int p2, int p3);

void process (int a1, int a2, int a3, oper_t oper) { oper(a1, a2-a3); }
```

```
process: push {lr, r4-r12}.   // → r0 ok
         sub r1, r1, r2 ✓     // a2 = a2 - a3 → r1
         blx r3 ✓
4        // rien de plus à faire ...
         pop {pc, r4-r12} ✓
```

c) Codez en assembleur la fonction **add** ci-dessous. Note : les « **int** » ont 32 bits

```
int add (int a1, int a2) { return (a1 + a2); }
```

```
add: push{lr}
     add r0, r0, r1   // r0 = r0+r1;
     pop{pc}; ✓
```

Systèmes Embarqués 1 & 2: Travail écrit no 3.

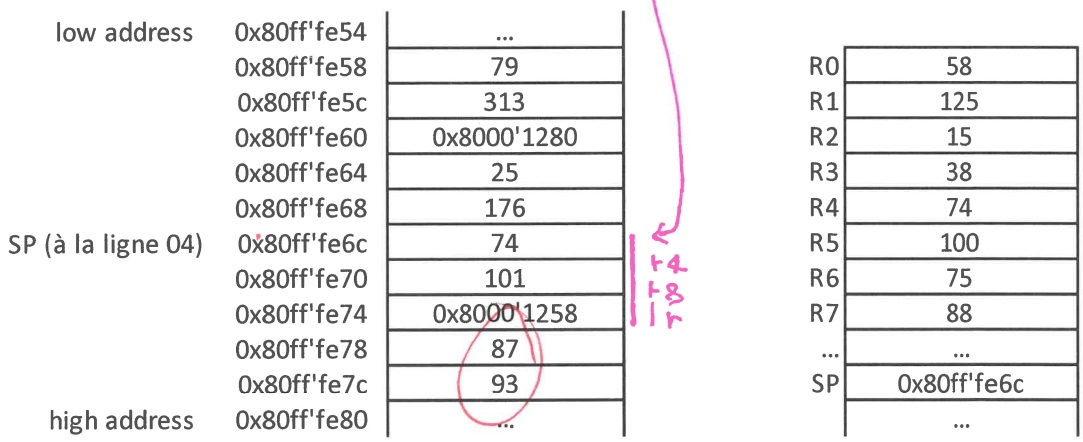
d) Lors d'une session de debugging, les développeurs ont rencontré des difficultés avec l'exécution de la fonction **fnct2** ci-dessous.

```
int fnct2 (int a1, int a2, int a3, int a4, int a5, int a6)
{
      return a1 + a3 + fnct3(a3, a4*a5, a6);
}
```

Afin de trouver l'erreur, ils ont désassemblé la fonction **fnct2** à l'aide du debugger GDB, dont voici une partie du code désassemblé :

```
01    fnct2:
02          nop
03          push    {r4,r8,lr}
04          add     r8, r0, r2
05          …
```

A l'aide de ce même débugger, ils ont « dumpé » une partie de la pile et des registres du processeur. Le graphique ci-dessous représente l'état du processeur (registres et pile sur 32 bits) lorsqu'ils arrêtent l'exécution de la fonction à la ligne n° 04 du code désassemblé ci-dessus.



Ils souhaitent maintenant connaître la valeur de chacun des arguments de la fonction lors de son appel. Pouvez-vous indiquer la valeur des paramètres a1 à a6 :

a1 : r0 → 58

a2 : r1 → 125

a3 : r2 → ~~15~~

a4 : r3 → 38

a5 : sp+4 → 101

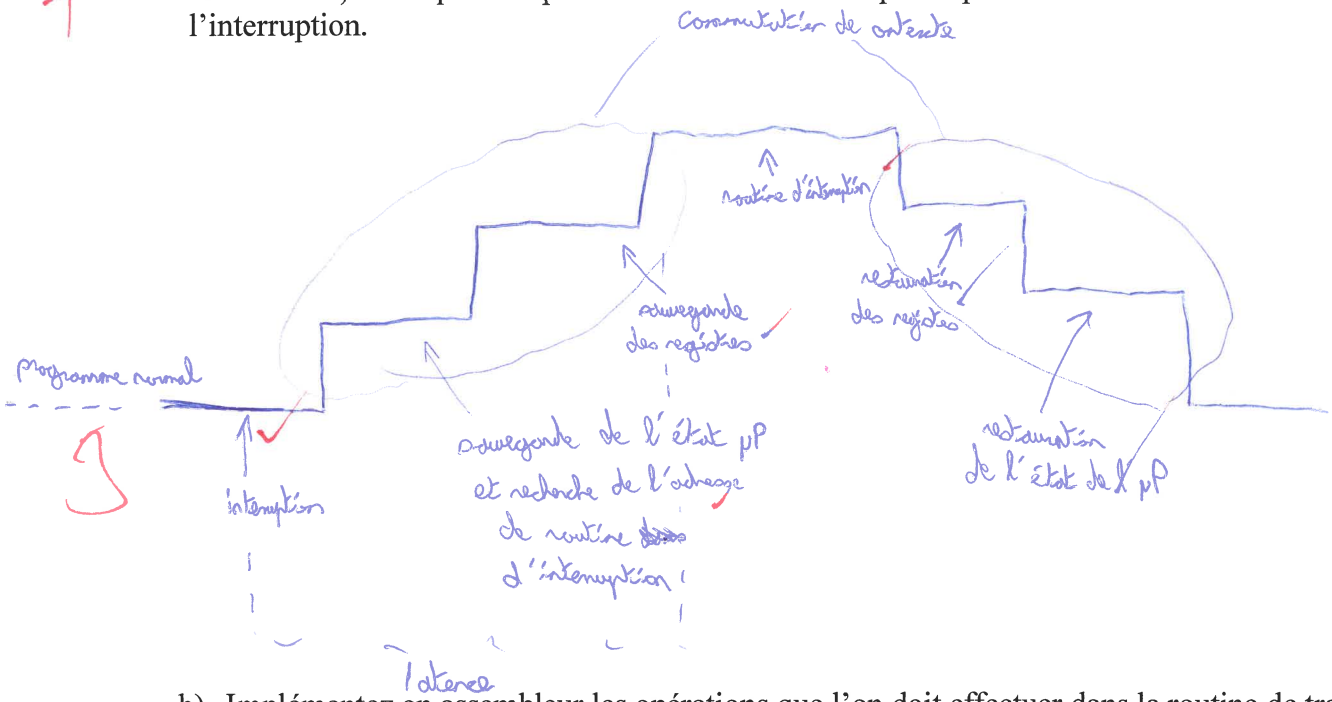
a6 : sp+8 → 0x8000'1258

2.1

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 2 (Interruptions)

a) Expliquez à l'aide d'un graphique la commutation de contexte d'interruption (sauvegarde et restauration) ainsi que les opérations effectuées à chaque étape du traitement dès la levée de l'interruption.



b) Implémentez en assembleur les opérations que l'on doit effectuer dans la routine de traitement d'interruption de 1er niveau en assembleur (irq_handler). Cette routine appellera de manière indirecte la routine de traitement de 1er niveau en C, via le pointeur de fonction stocker dans la variable « flih ».

irq_handler:

```
push {r4-r12, lr}

ldr r4, =flih
blx r4

pop {r4-r12, pc}
```

Systèmes Embarqués 1 & 2: Travail écrit no 3.

c) Expliquez comment le µP ARM am3358 de TI (µP ARM Cortex-A8) va pouvoir trouver et appeler la routine de traitement d'interruption irq_handler ci-dessus.

Le µP possède une table de vecteur d'interruption qui lie chaque type d'event (après l'avoir identifié en partie via l'INTC), à une adresse de routine de traitement d'interruption.

En parcourant la table avec le code de l'event, l'µP peut donc trouver et appeler la routine de traitement.

d) Expliquez la raison pour laquelle les µP ARM disposent de plusieurs modes d'opération.

Certains périphériques / fonctions ne doivent pas être disponibles pour chaque type d'utilisateur, ce qui nécessite des modes d'opérations variés.

De plus le traitement des interruptions peut fonctionner de différentes manières suivant le mode utilisé.

(je n'ai aucune idée de ce que j'écris) ← au moins c'est vrai pour lui ☺

- sécurité
- pas de pile ⇒ registres

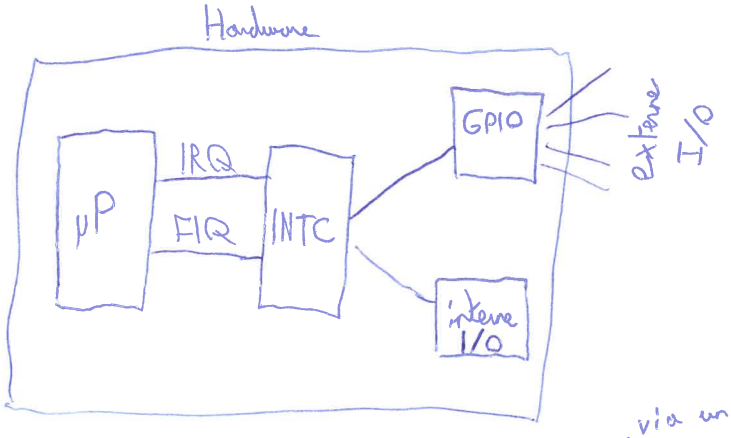
pas de pile

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 3 (Interruptions)

10

a) Les µP ARM ne disposent que de 2 lignes d'interruptions au niveau du core, la ligne IRQ et la ligne FIQ. Décrivez à l'aide d'une figure la solution mise en œuvre pour le µP AM3358 pour connecter une centaine de périphériques internes et 128 périphériques externes au système d'interruption du µP ARM Cortex A8.



5

INTC possède une ligne dédiée pour chaque I/O interne et GPIO, via un controlleur, il peut donc facilement transmettre l'interruption au µP. Si l'interruption vient d'un des GPIO, celui-ci doit fournir le code d'interruption.

b) Indiquez, pour les 3 composants principaux de la question précédente, la méthode utilisée pour identifier la source d'interruption (vectorisé, priorité ou scrutation) et argumentez votre choix.

µP: priorité ✓ il possède 2 lignes, une prioritaire (FIQ) et une "normale" (IRQ). Il n'a pas besoin de connaître directement l'origine de l'interruption car cela est traité au niveau de l'INTC

3

INTC: vectorisé ✓ à l'aide d'un controlleur et de ligne dédiée vers chaque GPIO/interne I/O, il peut identifier efficacement le type d'interruption.

GPIO: scrutation: ✓ GPIO doit interroger chaque périphérique externe les uns après les autres pour définir l'origine de l'interruption et fournir son code à l'INTC

c) Expliquez ce qu'est la « gigue d'interruption » et indiquez la raison principale à son origine.

gigue d'interruption → variation dans le temps entre l'interruption et le traitement de la routine (=latence variable)

2

cette dernière peut être très différente suivant le temps effectué pour terminer l'instruction en cours, mais surtout suivant le temps que les composants restent dans l'état où ils ne peuvent pas transmettre d'interruption. Le temps de commutation de contexte est variable également, mais dans une moindre mesure.

Systèmes Embarqués 1 & 2: Travail écrit no 3.

10

Problème n° 4 (Interruptions)

a, b et c sont des variables de type `int32_t` et nous utilisons un système « multithreads » **préemptif**. Considérez le code ci-dessous :

```
void f() {
    // ...
    a = 10;
    b = 20;
    c = a + b;
    b = b + 1;
}
```

a) Si a, b et c sont des variables **locales**, quelles sont les valeurs possibles pour a, b et c à la fin de ce code ? Expliquez votre réponse.

~~... affectation simple~~

a = 10 , b = 21 , c = 30

2 → les variables sont locales, les autres threads n'ont donc aucun effet sur les opérations.

b) Quelles sont les valeurs possibles pour a, b et c si a, b et c sont des variables **globales** et que **d'autres fonctions ont également accès à ces variables** ? Expliquez votre réponse.

~~...~~

a, b et c peuvent avoir des valeurs difficilement prévisibles et chaque tentative risque de mener à des résultats différents.

2 En prenant comme simple exemple une fonction g() { a = 20; } a pourra alors prendre les valeurs 10 ou 20 suivants si f ou g a été appelé en dernier par l'un des threads.

Cet effet s'aggrave encore plus pour les autres opérations.

Systèmes Embarqués 1 & 2: Travail écrit no 3.

c) Quelles sont les valeurs possibles pour a, b et c si a, b et c sont des variables **globales**, mais que **seule la fonction f modifie ces variables et que f est appelée n fois** (n > 1). Expliquez votre réponse.

a = 10 : n'est touché que par une affectation simple

b = [21; 20+n] : ~~si on imagine 2 threads effectuer~~ b peut être mis à jour par un autre thread alors que le thread courant s'est juste de récupérer son ancienne valeur et l'incrémenté. Après 2 appels, on pourrait donc avoir la valeur 21 ou 22. Après n appel, c'est encore plus imprévisible

2

c = [30; $30+n^{n-1}$] : ~~si~~ un thread effectue l'affectation a+b, il est possible qu'un autre thread ait déjà fait la ligne b = b+1.

d) Dans les 3 situations ci-dessus, s'il y a plusieurs valeurs possibles pour une variable, expliquez comment remédier à cela et faire en sorte qu'il n'y a qu'une seule valeur possible.

Il faut rendre les opérations ~~non~~-atomiques sur ces variables mutuellement exclusive, section critique ← variable partagées

2 On s'aide en général pour cela avec des sémaphores. ✓

e) Expliquez la raison pour laquelle le déclenchement des interruptions n'est pas optimal pour protéger des sections critiques et citez un mécanisme approprié pour ce type d'opération.

1.5 Le déclenchement d'interruption n'est pas instantané (sujet à une latence) et peut être problématique si elle se déroule au milieu d'une opération.

⇒ jitter.

- bloque pendant trop longtemps
- produit trop de jitter

Systèmes Embarqués 1 & 2: Travail écrit no 3.

Problème n° 5 (Système d'exploitation - noyau)

a) Implémentez un type en C, pour représenter les états d'un thread pour un mini-noyau coopératif. Donnez une petite description de l'état.

```
enum state { READY, RUNNING, BLOCKED, TERMINATE }
```

↳ le thread est prêt et dispo pour prendre la main

→ le thread est actuellement actif

→ le thread est en attente (d'un input, essentiellement)

→ le thread est terminée.

b) Implémentez en assembleur la méthode transfert exécutant la commutation de contexte entre 2 tâches (threads)

```
void transfer (struct context* former_task, struct context* new_task);
```

(r0 : former_task, r1 : new_task)

```
transfer:  push{lr, r0-r12}
           mrs  r4, cpsr
           str  r4, [r0, #15*4]
           ldr  r2, [r1, #15*4]
           msr  cpsr, r2

           pop {pc, r0-r12}
```

c) Décrivez en quelques lignes les différences entre thread et processus

Un thread possède une mémoire et des données partagées avec d'autres threads, ce qui n'est pas le cas d'un processus.

Un processus est plus "lourd", on considère généralement qu'1 processus est composé de plusieurs threads.

processus espace mémoire protégé

- espace partagé pour thread
- espace protégé pour processus

Systèmes Embarqués 1 & 2: Travail écrit no 3.

d) Définissez la structure C minimale représentant un sémaphore, composant indispensable d'un OS

```
struct sema { struct tcb* next;
              uint_32 count; }
```

2

on ajoute également parfois un pointeur vers le prochain semaphore, "struct sema* next_s", mais ce n'est pas indispensable.

e) Implémentez en C la méthode « sema_signal() » permettant de libérer le sémaphore

→ on considère une structure sema, appelée "sema", déjà définie.

```
void sema_signal() {
    sema.count ++;
    if (sema.next != 0) {
        sema.next.state = READY; // le prochain TCB peut passer
        sema.next = sema.next.next; }
    yield();
}
```

2