Modélisation Transactionnelle des Systèmes sur Puces

Ensimag 3A, filière SEOC

Février 2020

Consignes :

- durée : 2h ;
- une feuille manuscrite recto-verso est le seul document autorisé ;
- le barème est donné à titre indicatif ;
- on attend des réponses courtes et pertinentes, inutile de tirer à la ligne.

1 Question de cours sur le langage C++

1.1 Héritage en C++

On considère le programme suivant :

```
#include <iostream>
using namespace std;

struct Root {
        int x;
        Root () { x = 0xbaaaaaad; }
};

struct X : Root {
        int y;
        X () { x = 0xbeef; y = 0xf00d; }
};

struct Y : Root {
        int y;
        Y () { x = 0xdead; y = 0xd00d; }
};

struct Z : X, Y {
        void w() {
```

```
                cout << "x␣=␣" << x << endl;
                cout << "y␣=␣" << y << endl;
        }
};

int main() {
        Z z;
        z.w();
}
```

Question 1 (2 points) *Le programme est-il compilable ? Si oui, qu'affiche-t-il ? Si non, proposez une correction.*

Il n'est pas compilable, il faut désambiguiser les accès à x et y dans Z, avec par ex. X : :x et Y : :y.

1.2 Templates en C++

On considère la classe template suivante, dans laquelle certains identificateurs ont été remplacés par les identifiants `X1` à `X4` :

```
template<typename T, size_t S>
struct bounded_array {
        X1 elements[X2];

        X3 get_elem(unsigned index) {
                assert(index < S);
                return elements[index];
        }

        X4 set_elem(unsigned index, T e) {
                assert(index < S);
                elements[index] = e;
        }
};
```

Question 2 (1 point) *Par quoi faut-il remplacer `X1`, `X2`, `X3` et `X4` pour que la déclaration soit proprement typée ?*

```
% 0.25 point chacun.
#define X1 T
#define X2 S
#define X3 T
#define X4 void
```

Question 3 (1 point) *Écrivez un programme principal qui instancie cette classe* `bounded_array` *en un tableau de 12 flottants, et qui utilise les méthodes* `get_elem()` *et* `set_elem()` *pour faire ce qu'il vous plaira.*

```
#include <iostream>
#include <assert.h>
using namespace std;

#define X1 T
#define X2 S
#define X3 T
#define X4 void

#include "array-def.h"

int main() {
        bounded_array<float, 42> t;
        t.set_elem(41, 14.0);
        cout << t.get_elem(41) << endl;
}
```

```
4
```

2 Questions de cours sur TLM et les SoCs

3 Modélisation du temps et ordonnancement en SystemC

On considère le programme suivant :

```
#include <systemc>
#include <iostream>

using namespace std;
using namespace sc_core;

SC_MODULE(A) {
        void f() {
                wait(1, SC_SEC);
                cout << "f1" << endl;
                wait(2, SC_SEC);
                cout << "f2" << endl;
                sleep(1);
                cout << "f3" << endl;
        }
        SC_CTOR(A) {
```

```
                SC_THREAD(f);
        }
};

SC_MODULE(B) {
        void g() {
                cout << "g0" << endl;
                sleep(2);
                cout << "g1" << endl;
                wait(4, SC_SEC);
                cout << "g2" << endl;
        }
        SC_CTOR(B) {
                SC_THREAD(g);
        }
};

int sc_main(int, char**)
{
        A a("a");
        B b1("b1");
        B b2("b2");

        sc_start();

        return 0;
}
```

On rappelle que la fonction `sleep(n)` est une fonction POSIX (pas une fonction SystemC) qui provoque une attente de `n` secondes du processus courant.

Le standard SystemC autorise plusieurs exécutions possibles de ce programme (i.e. l'ordonnanceur a la liberté de choisir entre ces exécutions). Certaines exécutions produisent les mêmes affichages.

Question 4 (2 points) *Combien y a-t-il d'exécution différentes (i.e., produisant des affichages différents) autorisées par SystemC ? Donnez cette ou ces traces d'exécution.*

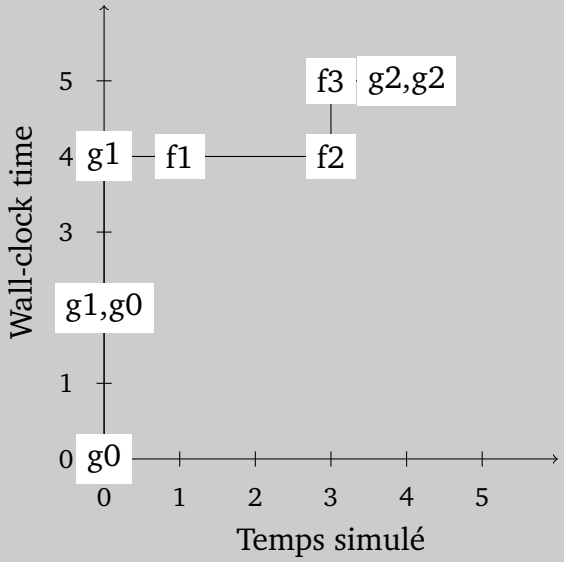
Au lancement de la simulation, b1.g et b2.g sont éligibles. a.f est aussi éligible mais ne fait rien d'autre qu'entrer dans un wait, donc a.f n'a aucun effet sur l'affichage.

b1.g et b2.g s'exécutent l'un après l'autre sans rendre la main jusqu'au wait. On arrive ensuite au temps simulé t=1, on exécute f1, puis t=3, on exécute f2, puis t=4 on exécute g4 deux fois. Les seuls choix possibles sont entre b1.g et b2.g qui font la même chose, il n'y a qu'un affichage possible.

```
g0
g1
g0
```

```
g1
f1
f2
f3
g2
g2
```

Question 5 (3 points) *Représentez une des exécutions sur un graphique à deux dimensions. On mettra le "wall-clock time" sur l'axe des ordonnées et le temps simulé sur les abscisses. Si on néglige le temps de calcul, combien de temps prendra une exécution en "wall-clock time"? Combien de temps en temps simulé ?*



Dans les 2 cas, 4 secondes de temps simulé pour une exécution de 5 secondes en wall-clock time.

4 Modélisation d'un "spin-lock engine" matériel

L'implantation classique d'un verrou d'exclusion mutuelle se fait grâce à des opérations atomiques sur la mémoire. Ceci permet d'avoir autant de verrous que l'on veut, au prix d'une complexité importante dans le matériel du sous-système mémoire. Il est possible de simplifier grandement le matériel en dédiant un composant à la gestion des spin-locks, avec la contrainte que le nombre de spin-locks est alors fortement contraint.

On se propose donc maintenant de modéliser un composant qui mette à disposition du programmeur d'OS embarqué un ensemble de 1024 spin-locks. Du point de vue logiciel, il n'y a pas d'instructions spéciales genre load-linked/store-conditional ou compare and swap, on peut vivre avec les instructions de lecture et d'écriture classiques du processeur.

Nous utiliserons les mêmes conventions que dans le TP2, mais on suppose pour les besoins de la cause que la plateforme a été étendue à plusieurs processeurs, sans quoi l'utilité de ce composant est plus que discutable.

Le but de cette partie est d'implémenter les fonctions suivantes (pour utilisation dans le logiciel embarqué) :

```
// Acquière le spin-lock numéro n (avec 0 <= n < 1024)
// Si le spin-lock est déjà verrouillé, boucle tant qu'il n'est pas relâché
void hw_spin_lock(int n);

// Relâche le spin-lock numéro n (avec 0 <= n < 1024)
void hw_spin_unlock(int n);

// Tente une acquisition du spin-lock numéro n, et renvoie 1 si le
// verrouillage réussit. En cas d'échec, abandonne immédiatement et
// renvoie 0.
int hw_spin_trylock(int n);
```

Les plus assidus d'entre vous auront reconnu à peu de choses près le nom des fonctions de la bibliothèque des threads posix : pthread_spin_lock, pthread_spin_unlock, pthread_spin_trylock : la sémantique des fonctions ci-dessus est la même.

Voici un exemple d'utilisation de ces fonctions :

```
hw_spin_lock(0);
// ... section critique
hw_spin_unlock(0);
```

Le composant est une cible sur le bus, et chaque "spin" ne peut contenir que 2 valeurs possibles, 0 ou 1. Trois opérations différentes peuvent être exécutées sur un spin-lock :

1. Écrire une valeur (0 ou 1)
2. Lire une valeur (0 ou 1)
3. Tenter de prendre le spin-lock :
   - si le registre visé contient la valeur 0, alors le registre prend la valeur 1 et l'opération renvoie 0, indiquant que le spin-lock a été acquis ;
   - si le registre contient la valeur 1, alors le registre prend la valeur 1 et l'opération renvoie 1. On voit que dans les deux cas on retourne la valeur présente dans le registre lors de l'accès, avant de mettre à 1 ce dernier, ce que l'on fait d'ailleur de manière systématique.

Les opérations 1 et 2 se font via les opérations read et write du bus : pour accéder au spin-lock n, on fait une lecture ou une écriture à l'offset $n \times 4$ dans le composant. L'opération 3 se fait via une opération read : lire à l'offset $(n + 1024) \times 4$ provoque une tentative de prise du spin-lock n. La valeur lue est le résultat de l'opération (0 si le spin-lock était libre et a été acquis, 1 sinon).

Écrire à un déplacement (offset) supérieur à 1024×4 ou lire à une adresse supérieure à $1024 \times 4 \times 2$ est interdit.

Initialement, tous les registres du composant contiennent la valeur 0, indiquant que tous les spin-locks sont libres.

Le composant est placé à l'adresse `0x40700000` dans la plateforme. La constante suivante est définie :

```
#define SPIN_ENGINE_BASEADDR 0x40700000
```

4.1 Logiciel embarqué

Question 6 (2 points) *Pour mémoire, l'API d'abstraction du matériel du TP2 (hal.h, vu en cours) contient en particulier les deux fonctions suivantes :* `int32_t hal_read32(a)`*, qui retourne la valeur lue sur le bus à l'adresse* `a`*,* `hal_write32(a, d)`*, qui écrit sur le bus à l'adresse* `a` *la valeur* `d`*,* `hal_wait_for_irq()`*, qui met le processeur en pause en attente d'une interruption, et* `hal_cpu_relax()`*, qui, pour la simulation native, rend la main au noyau de simulation afin de lui permettre de faire progresser les autres composants. En utilisant cette API, proposez une implantation des fonctions* `hw_spin_lock(int n)`*,* `hw_spin_unlock(int n)` *et* `hw_spin_trylock(int n)`*.*

```
int hw_spin_trylock(int n)
{
        return hal_read32(SPIN_ENGINE_BASEADDR + (n + 1024) * n);
}

void hw_spin_lock(int n)
{
        while (!hw_spin_trylock(n)) {
                cpu_relax();
        }
}

void hw_spin_unlock(int n)
{
        hal_write32(SPIN_ENGINE_BASEADDR + 4 * n, 0);
}
```

4.2 Modélisation du matériel

Un point clé pour le fonctionnement du composant est l'atomicité : si plusieurs modules tentent d'accéder au composant en même temps, les accès ne sont jamais entrelacés : ils sont dits « sérialisés ».

Question 7 (0.5 point) *Dans l'implantation matérielle, quel mécanisme permet de garantir cette atomicité ?*

L'arbitrage sur le bus qui fait que deux initiateurs ne peuvent accéder au même port cible en même temps.

Question 8 (0.5 point) *Dans le modèle SystemC/TLM, quel mécanisme permet de garantir cette atomicité ?*

La sémantique de co-routine de SystemC.

On propose le squelette de code suivant :

```
// Fichier spinengine.h
#define SPIN_ENGINE_NBSLOTS 1024
#define SPIN_ENGINE_ADDRSIZE (SPIN_ENGINE_NBSLOTS * sizeof(ensitlm::data_t))

class SPIN_ENGINE : public sc_core::sc_module {
public:
        ensitlm::target_socket<SPIN_ENGINE> target;

        tlm::tlm_response_status
                read(ensitlm::addr_t a, ensitlm::data_t& d);

        tlm::tlm_response_status
                write(ensitlm::addr_t a, ensitlm::data_t d);
};

// Fichier spinengine.cpp
tlm::tlm_response_status
SPIN_ENGINE::write(ensitlm::addr_t a, ensitlm::data_t d)
{
        if (a >= SPIN_ENGINE_ADDRSIZE) {
                return tlm::TLM_ADDRESS_ERROR_RESPONSE;
        }
        // ...
        return tlm::TLM_OK_RESPONSE;
}

tlm::tlm_response_status
SPIN_ENGINE::read(ensitlm::addr_t a, ensitlm::data_t& d)
{
        // ...
        return tlm::TLM_OK_RESPONSE;
}
```

Question 9 (0.5 point) *À quoi sert le test `if (a >= SPIN_ENGINE_ADDRSIZE)` en début de méthode `write`?*

À vérifier qu'on est bien dans la plage autorisée avant de faire une écriture.

Question 10 (1.5 points) *Faut-il ajouter quelque chose à la déclaration de classe dans `spinengine.h` pour implanter le comportement du composant dans `read` et `write`? Si oui, écrivez le code à ajouter.*

```
        SC_CTOR(SPIN_ENGINE) {
            memset(locks, 0, sizeof(SPIN_ENGINE::locks));
        }
private:
        bool locks[SPIN_ENGINE_NBSLOTS];
```

Question 11 (1.5 points) *Proposez une implantation pour la partie manquante de la méthode* `write`.

```
        locks[a / 4] = (d == 1); // Attention au /4.
```

Question 12 (3 points) *Proposez une implémentation pour la partie manquante de la méthode* `read`.

```
        if (a >= SPIN_ENGINE_ADDRSIZE) {
                a = a - SPIN_ENGINE_ADDRSIZE;
                if (a >= SPIN_ENGINE_ADDRSIZE) {
                        return tlm::TLM_ADDRESS_ERROR_RESPONSE;
                }
                // Test and set
                if (locks[a / 4]) {
                        // Failure, already set.
                        d = 1;
                } else {
                        locks[a / 4] = true;
                        d = 0;
                }
        } else {
                // Read
                d = locks[a / 4];
        }
```

Question 13 (1.5 points) *Le "top level" du système est implanté dans la fonction* `sc_main`*, qui n'est pas reproduite ici. Que faut-il y ajouter pour pouvoir utiliser notre composant* `spinengine`*?*

```
SPIN_ENGINE spinengine("spinengine");

bus.initiator(spinengine.target);

bus.map(spinengine.target, SPIN_ENGINE_BASEADDR, 1024*2*4);
```

4.3 Questions bonus

Les questions de cette partie sont des questions de compréhension de l'interface matériel/logiciel non liées à SystemC).

Notre implantation du spin en utilisant le composant que nous avons développé est assez inefficace dans le cas où un processus tente d'accéder à un spin déjà verrouillé et que l'attente est longue (i.e. si les sections critiques sont longues).

Question 14 (1 point) *Quels sont les problèmes qui amènent à des pertes de performances ?*

Attente active, donc utilisation du processeur, et surcharge du bus et du composant spinengine pendant toute la période d'attente. Les autres processus auraient pu utiliser le CPU, mais même les autres composants sont ralentis par l'occupation du bus.

Question 15 (1 point) *Quelle(s) solution(s) proposez-vous pour améliorer les performances dans ce cas (pour la plateforme du TP2 et/ou pour une plateforme plus complexe) ? Il n'est pas demandé de les implémenter.*

fall-back sur un autre mécanisme (e.g. appel système si on a un OS) après n tour de boucles.

exponential backoff côté logiciel : si on voit que le verrou n'est pas disponible, on attend un peu, en multipliant par 2 la période d'attente à chaque tour de boucle.

gestion des priorités entre process s'il y a du multi-tâche : on rend la main à chaque tour de boucle.

mécanisme à base d'interruption, ou déverrouiller un spin enverrait une interruption au CPU qui en a fait la demande.

18
22