École Normale Supérieure — Systèmes et réseaux
Année 2012-2013

TP6 Cubicle

Le but de ce TP est de modéliser et de prouver des programmes concurrents en utilisant Cubicle [2] : `http://cubicle.lri.fr`.

1 Algorithme de Peterson

Nous partons de l'implémentation de l'algrithme de Peterson suivante où `Turn`, `Want` et `Other` sont des variables globales :

```
void init () {
  Turn = P1;
  Want[P1] = False; Want[P2] = False;
  Other[P1] = P2; Other[P2] = P1;
}

void process (int i) {
  while (True) {
  L1:
    Want[i] = True;
  L2:
    Turn = Other[i];
  L3:
    while ( Want[Other[i]] && !(Turn = i) ) { ; }
  L4_crit:
    // section critique
  L5:
    Want[i] = False;
  L6:
    // section restante
  }
}
```

La modélisation de cet algorithme en Cubicle est donnée dans le fichier `peterson.cub` disponible dans l'archive `http://www.lri.fr/~mandel/systeme-ens/tp-06.tgz`.

Question 1. En utilisant Cubicle,[1] prouver que tel qu'il est modélisé, l'algorithme de Peterson est sûr (il ne peut pas y avoir deux processus simultanément en section critique) et que l'arrêt d'un processus en dehors d'une section critique n'empêche pas l'autre d'y entrer.

1. Pour utiliser Cubicle sur les machines de la salle Info 4, vous devez ajouter à la variable d'environnement `PATH` le chemin `~mandel/local/bin`.

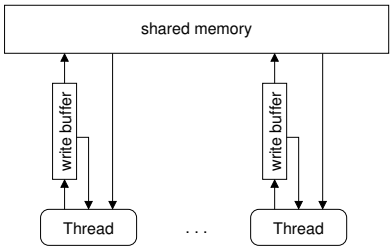
Question 2. Modifier l'algorithme pour introduire des problèmes de sûreté et d'accès à la section critique (*cf.* Solution 2 et Solution 3 du cours). Donner des traces qui exhibent ces problèmes.

Question 3. Prouver que tel qu'il est modélisé, l'algorithme de Peterson n'a pas de deadlock.

Dans le fichier `peterson.c`, l'algorithme de Peterson a été utilisé pour garantir l'exclusion mutuelle de deux threads qui incrémentent 1 000 000 de fois chacune une variable globale `cpt`.

Question 4. Tester le programme `peterson.c` sur votre machine et vérifier qu'il affiche « `cpt = 2000000` ».

Les machines multi-coeurs se comportent comme si les écritures en mémoire se faisaient à travers un buffer. Ainsi, une affectation revient à écrire une valeur dans un buffer et cette valeur est copiée de façon asynchrone en mémoire centrale.



Sur ces machines, la sémantique par entrelacements des threads n'est pas respectée. Par exemple, le programme suivant peut terminer avec `r1 = 0` et `r2 = 0` si initialement `x = 0` et `y = 0` :

Thread 0	Thread 1
`x := 1;`	`y := 1;`
`r1 := y;`	`r2 := x;`

Question 5. Modifier le fichier `peterson.cub` pour modéliser l'exécution du programme sur une machine multi-coeur. Donner une trace qui conduit à un problème de sûreté.

Il existe une instruction `fence` dont l'effet correspond à vider le buffer local au thread qui l'exécute.

Question 6. Déterminer où utiliser cette instruction pour rendre le code de l'algorithme Peterson de nouveau correct.

Question 7. En utilisant la macro suivante, corriger le fichier de `peterson.c`.

```
#define fence()  __asm__ __volatile__ ("mfence":::"memory")
```

2 Cohérence de cache : le protocole MESI

Afin d'assurer la cohérence des données entre les caches, des protocoles, dit de cohérence de cache, sont mis en oeuvre. Dans cette partie, nous allons modéliser le protocole MESI [3] qui garantit un modèle de *cohérence forte* : toute lecture d'une donnée reflétera toute modification antérieure à la lecture sur n'importe quelle copie de la donnée.

Dans ce protocole, un cache peut être dans quatre états : `Modified`, `Exclusive`, `Shared` ou `Invalid`.

L'état `Modified`. Les données qui sont actuellement de la cache sont « sales » : elles ont été modifiées par rapport aux données qui sont en mémoire centrale.

L'état `Exclusive`. Les données qui sont dans le cache sont « propres » : elles correspondent aux données présentes en mémoire centrale.

L'état `Shared`. Indique que les données qui sont dans le cache sont aussi potentiellement présentes dans d'autres caches.

L'état `Invalid`. Indique que les données qui sont dans le cache ne sont pas valides.

Nous décrivons maintenant les changements d'état des caches en cas de requêtes de lectures et d'écritures. Nous reprenons la description du protocole telle qu'elle est donnée dans [1].

- Si un processeur effectue une requête de lecture alors que son cache est dans l'état `Invalid`, il y a un *read miss*. L'état du cache passe dans l'état `Shared` et simultanément, les caches qui étaient dans l'état `Exclusive` ou dans l'état `Modified` passent dans l'état `Shared`.
- Si un processeur effectue une lecture alors que son cache est dans un autre état, il y a un *read hit* et il reste dans son état.
- Si un processeur effectue une requête d'écriture alors que son cache est dans l'état `Invalid`, il y a un *write miss*. L'état du cache passe dans l'état `Exclusive` et tous les autres passent dans l'état `Invalid`.
- Enfin, si un processeur effectue une requête d'écriture alors que son cache n'est pas invalide, il y a trois cas possibles :
  - si son cache est dans l'état `Exclusive`, il passe en `Modified` ;
  - si son cache est dans l'état `Shared`, il passe en `Exclusive` et tous les autres passent dans l'état invalide ;
  - si son cache est dans l'état `Modified`, il reste dans cet état.

Dans le tableau ci-dessous, les cases marquées d'une croix marquent les états qui ne sont pas permis pour tout couple de caches.

	M	E	S	I
M	X	X	X	
E	X	X	X	
S	X	X		
I				

Question 8. Prouver en Cubicle qu'aucun de ces états n'est accessible à partir d'une configuration initiale où tous les caches sont dans l'état `Invalid`.

Références

[1] Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *Proc. TACAS '07-13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[2] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *Computer Aided Verification - 24rd International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012. Proceedings*, Lecture Notes in Computer Science. Springer, 2012.

[3] Jim Handy. *The cache memory book*. Academic Press, 1993.