TD – Outils d'exclusion mutuelle à base d'attente active

Cours Applications Concurrentes, *Polytech-INFO4*

F. Boyer, Université Grenoble Alpes, sept 2022

Nécessité de l'exclusion mutuelle

Lorsque des flôts d'exécution (processus ou threads[1]) partagent des ressources (données, terminal, ..), se pose le problème du maintien de la cohérence de ces ressources.
Intuitivement, c'est un peu comme si vous aviez des utilisateurs qui écrivent en parallèle dans un même fichier. Pour garantir la cohérence d'un fichier partagé, les utilisateurs doivent respecter certaines contraintes (par exemple, ne pas écrire au même endroit en même temps). Avec les threads, c'est un peu pareil, la manipulation de ressources partagées doit respecter certaines règles.

Considérerons le cas d'une imprimante avec un buffer de dépôt des fichiers à imprimer. On comprend facilement que les threads doivent déposer les fichiers à imprimer dans des cases différentes du buffer.
Considérons également le cas classique des comptes bancaire. Il faut bien sûr garantir que les exécutions concurrentes des opérations de crédit et débit n'entravent pas la cohérence du compte manipulé.

Les sections suivantes donnent quelques scénarios illustrant le problème.

1. Exécution indéterministe

L'exécution concurrente de threads accédant à des données partagées peut engendrer des résultats indéterministes, en raison des commutations qui peuvent avoir lieu à tout moment.
Soient A et B deux variables entières partagées entre deux thteads (initialement, A vaut 1 et B vaut 2).

thread T1 :	thread T2 :
a1) x = B	a2) y = A
b1) A = x	b2) B = y

Question 1. Trouver au moins 3 résultats possibles et différents pour (A,B), en élaborant trois scénarios d'entrelacements différents de T1 et T2.

2. Incohérence de données

Soit *cpt* une variable partagée qui compte le nombre de threads entrant dans une certaine

[1] Dans la suite on utilisera seulement le terme *thread* par simplicité.

section de code (fonction *incr)*. Les threads T1 et T2 exécutent la fonction *incr*() de manière concurrente.

int cpt = 0 ; fonction incr : cpt = cpt + 1 ; …	thread T1 : … incr() ; …	thread T2 : … incr() ; …

Question 2. Quelle sont les valeurs possibles de la variable *cpt* aprés l'exécution de T1 et de T2 ?

Remarque. Se rappeler que l'incrémentation d'une variable fait intervenir plusieurs instructions de bas niveau. Par ailleurs l'accumulateur (accu) est un registre sauvegardé lors des commutations.

Question 2-bis. Programmez une application multi-threadée très simple en Java dans laquelle vous créez 3 threads concurrents qui effectuent chacun 500 incrémentations d'un compteur partagé (*for (int i=0 ; i<500 ; i++) cpt++ ;*). Vérifiez que la valeur du compteur à l'issue de l'exécution n'est pas toujours égale à 1500.

3. Exclusion mutuelle

Les scénarios précédents illustrent le besoin de disposer d'outils permettant de rendre **atomique** un bloc d'instructions, c'est à dire de garantir que le résultat de l'exécution du bloc d'instruction sera équivalent à une exécution sans commutation. Attention, nous ne sommes pas en train de dire qu'une exécution atomique interdit les commutations !

Dans le langage général, on dit d'un bloc d'instruction atomique qu'il est exécuté en exclusion mutuelle.

Bloc d'instruction exécuté en exclusion mutuelle

Assure que si un processus P commence à exécuter le bloc d'instructions, tout autre processus essayant d'exécuter le bloc d'instructions *sera mis en attente* tant que P n'aura pas terminé l'exécution du bloc.

Notion de section critique

Une section critique (SC) correspond à un bloc d'instruction systématiquement exécuté en exclusion mutuelle. Les langages de programmation existants fournissent la notion de section critique. Schématiquement, le programmeur dispose de directives qu'il peut placer dans son code, indiquant le début et la fin de la section critique.

```
…
begin-section
  I1 ;
```

```
 I2 ;`
 I3 ;
end-section
…
```

Considérons la section critique composée des instructions I_1.I_2, I_3. Deux processus P1 et P2 essaient d'exécuter la section critique. Voici un scénario possible, dans lequel P1 entre en SC (il exécute la première instruction I_1), puis il est commuté. P2 prend la main, tente d'entrer en SC, mais est mis en attente puisque la SC est déjà occupée par P1. P2 ne pourra entrer en SC que lorsque P1 en sera sorti.

T0 : P1 exécute I_1
T1 : P2 essaie d'exécuter I_1 -> il doit être mis en attente
..
Tk : P1 exécute I_3
T_{k+1} : P2 exécute I_1

La notion de section critique a été formalisée par Djikstra. Les propriétés à garantir sont les suivantes:

- Prop1 : l'exclusion mutuelle
- Prop2 : pas de famine (tout processus qui se présente passe dans un temps fini)
- Prop3 : si la section est libre, l'accés est immédiat
- Prop4 : pas de processus privilégié
- Prop5 : les propriétés précédentes doivent être garanties quelque soit lea politique de scheduling (avec/sans temps partagé, ...)

Nous allons maintenant essayer de mettre en œuvre des sections critiques, c'est à dire de proposer une mise en oeuvre pour les directives *begin-section* et *end-section*. Nous allons considérer différentes approches intuitives et voir qu'elles présentent toutes une limitation (au moins une des propriétés précédente n'étant pas garantie). Incrémentalement, nous allons cependant arriver à une solution qui fonctionne.

4. Section critique à base de masquage des IT

Une première option consiste à simplement interdire les commutations durant l'exécution d'une section critique, en masquant les interruptions. Nous obtenons ainsi immédiatement la propriété d'atomicité. Cependant cette solution n'est pas réellement utilisable (sauf dans des contextes spécifiques).

Question 4. Lister des raisons pour lesquelles cette solution n'est pas retenue

5. Section critique à base d'attente active par variable partagée

On utilise une variable partagée qui permet de savoir si une section critique est libre ou occupée. La directive begin-section est composée des instructions a) et b), la directive end-section est composée de l'instruction d) ci-dessous.

1er essai

```
int busy = false;
        a)  while (busy);
        b)  busy = true
        c)   SC
        d)   busy = false
```

Question 5.1. Cet algorithme est non valide. Mettre en évidence un scénario qui démontre son invalidité.

2éme essai : on gère qui prend la ressource

```
int occupant[2] = {false, false};
        while (occupant[1-i]);
        occupant[i] = true
         SC
         occupant[i] = false
.
```

Question 5.2. Cet algorithme est non valide. Mettre en évidence un scénario qui démontre son invalidité.

3éme essai : un processus indique qu'il cherche à entrer en SC

```
int occupant[2] = {false, false};

        occupant[i] = TRUE
        while (occupant[1-i]) ;
        SC
        occupant[i] = false
```

Question 5.3. Cet algorithme est non valide. Mettre en évidence un scénario qui démontre son invalidité.

4ème essai : on remet occupant à Faux si l'autre processus cherche à entrer

```
int occupant[2] = {false, false};
        a) occupant[i] = true;
        b) while (occupant[1-i])  {
            c)      occupant[i] = false
            d)      occupant[i] = true
        }
        e) SC
        f) occupant[i] = false
```

Question 5.4. Cet algorithme est-il valide ? Mettre en évidence un scénario qui démontre son invalidité.

5ème essai : on utilise une variable qui définit le processus élu

```
int chosen = ..;

        while (i != chosen);
         SC;
         chosen = 1-i;
```

Question 5.5. Cet algorithme est-il valide ? Mettre en évidence un scénario qui démontre son invalidité.

6. Une solution qui fonctionne pour 2 processus : algorithme de Peterson

Le problème suivant a été mis en évidence dans les exercices précédents :

- Si un processus effectue un test (est-ce que la section est libre ?) avant de faire une mise à jour (la section devient occupée), on se retrouve avec plusieurs processus en SC parce que les tests sont effectués de manière concurrente.
- Si un processus effectue une mise à jour avant de tester la disponibilité de la section, il y a un risque d'attente infinie.

Pour résoudre ce dilemme, le principe de la solution de Peterson est le suivant : on utilise une variable *last* qui indique l'identité du dernier processus ayant demandé à entrer en SC. Si les 2 processus ont exécuté *occupant[i] = true*, alors la variable *last* va les départager : seul le premier arrivé passera.

Question 6.1. Avec ces indications, essayez de produire la solution de Peterson pour 2 processus (*begin-section* se code sur 3 lignes, *end-section* sur 1 ligne).

Question 6.2. Donner des exemples de scénarios d'exécution, notamment des scénarios montrant qu'il n'y a pas de famine possible.

P0	**P1**

occupant[0] = true;
last = 0
while …….. passant
<SC>
occupant[0] = false;
 	
occupant[0] = true; …… P0 continue
last = 1
while …….bloquant pour P0
	occupant[1] = true;
	last = 1
	while …….. bloquant
	
	while passant sur occupant[0] = false ou sur last =0
	…

7. Solution pour N processus : algorithme de Leslie Lamport (algorithme du boulanger)
Le principe consiste à affecter un numéro d'ordre é chaque processus. L'algorithme suppose qu'il y a au maximum N processeurs.

```
int demandeur [N + 1];                  // 1 .. N
int numero [N + 1] = {0,..};            // généralisation de tour, les processus passent dans l'ordre de
leurs numéros croissants

a)  demandeur[i] = true;
b)  for (j = 1; j <= N; j++)
        if (numero[j] > numero[i])      // celui qui s'exécute s'attribue le plus grand numéro
            numero[i] = numero[j];      // 2 processus peuvent sortir avec le méme numéro
    numero[i]++
    demandeur[i] = false
    for (j=1; j<=N; j++)
        while (demandeur[j]);  // si j est demandeur, attendre qu'il s'attribue un numéro
        while (numero[j] != 0) && (numero[j] < numero [i]) || (numero[j] == numero[i] && j < i) ;

    SC

    numero[i] = 0;
```

Question 7.1. Illustrer par un scénario la nécessité de l'instruction while (demandeur[j]) ;), qui force le processus courant à attendre que les processus en cours de demande de numéros aient terminé leur attribution de numéro.

Question finale. Les solutions que nous avons vues dans ce TD pour la mise en œuvre des directives *begin-section* et *end-section* sont qualifiées de solutions à base d'attente active. Expliquez pourquoi (en quelques lignes).