Algorithmes de Peterson et Lamport

Thibaut Antoine (thibaut.antoine@ens-rennes.fr)

Phrase d'introduction

On présente des algorithmes pour implémenter une section critique entre plusieurs threads : d'abord l'algorithme de Peterson avec sa correction pour deux threads, puis sa généralisation pour n threads : l'algorithme de Lamport.

1 Contexte [TOR]

Implémentation d'une section critique (SC) par deux fonctions, `lock()` et `unlock()` :

1. Exclusion mutuelle
2. Progrès (thread bloqué hors SC $\not\Rightarrow$ autres threads bloqués)
3. Attente bornée

On rappelle rapidement le contexte dans lequel on se place : on souhaite implémenter une section critique entre plusieurs threads. On doit donc donner l'implémentation de deux fonctions, `lock()` et `unlock()` qui entourent la section critique de sorte à assurer trois propriétés : l'exclusion mutuelle (pas deux threads en SC), le progrès (un thread bloqué en dehors de sa SC ne peut pas bloquer les autres processus – il n'y a pas d'interblocage), et l'attente bornée (une borne sur le temps d'attente avant entrée en SC).

2 Pour deux threads [TOR]

2.1 Approches naïves

On présente deux approches naïves et fausses, qui permettent d'illustrer à la fois deux idées qu'on va reprendre dans l'algorithme de Peterson, et certaines manières de ne pas respecter les propriétés 2 et 3.

On se donne P_0, P_1 deux threads qu'on veut synchroniser.

```
// Ex. Mutuelle
want = { false ; false };

void lock(int i) {
   want[i] = true;
   while(want[1 - i]);
}

void unlock(int i) {
    want[i] = false;
}
```

(1) ✓, (2) ×, (3) ✓
Les deux threads arrivent en même temps dans le `while`.

```
// Alternance
turn = 0;

void lock(int i) {
    while(turn != i);
}

void unlock(int i) {
    turn = 1 - i;
}
```

(1) ✓, (2) ✓, (3) ×
L'un des threads termine.

Le premier code se concentre sur l'implémentation de l'exclusion mutuelle avec un tableau partagé `want` : quand P_i souhaite entrer en section critique, il l'indique en mettant `want[i]` à vrai, puis il attend que l'autre processus soit sorti de sa section critique. Cependant il ne respecte pas la propriété 2, puisqu'il y a possibilité d'interblocage dans le cas où les deux threads entrent en attente au même moment.

Le second code tente de régler ce problème en forçant P_0 et P_1 à alterner au sein de la section critique, à l'aide de la variable partagée `turn` : d'abord P_0 entre et sort de la SC, puis P_1 fait de même, etc. Ainsi il n'y a pas d'interblocage, mais il n'y a pas attente bornée (3) : si P_1 termine, alors P_0 passe une dernière fois en section critique et attendra indéfiniment ensuite.

2.2 Algorithme de Peterson

```
want = { false ; false };
turn = 0;

void unlock(int i) {
    want[i] = false;
}
```

```
void lock(int i) {
    want[i] = true; // L1
    turn = 1 - i; // L2
    while(want[1 - i] &&
          turn == 1 - i); // L3
}
```

On réutilise ici les idées des deux codes précédents : on garde le tableau partagé `want` qui indique la volonté d'entrer en SC, mais avant d'entrer on laisse aussi à l'autre thread la possibilité d'entrer en SC s'il a envie. Et ainsi on vérifie les propriétés 1, 2, 3 (on le montre juste après).

3 Correction [BAR]

On s'attache ici à montrer la correction de l'algorithme de Peterson, sous l'hypothèse que la section critique ne modifie pas les variables partagées `want` et `turn`.

Théorème. *Si la SC ne modifie ni* `want` *ni* `turn`*, alors Peterson vérifie (1), (2) et (3).*

Démonstration.

1. Supposons P_0 et P_1 sont en SC. Alors `want[0] = want[1] = true`. Donc `turn = 0` et `turn = 1` : absurde.
2. Supposons un interblocage. Alors `want[0] = want[1] = true`, donc `turn = 1` et `turn = 0` : absurde.
3. Supposons P_0 bloqué avant la SC. P_1 est en SC. Alors `want[0] = want[1] = true` et `turn = 1`.
   - Si P_1 termine après son passage en SC : appel à `unlock(1)` donc `want[1] = false` et P_0 entre en SC.
   - Sinon :
     - Soit P_0 est ordonnancé avant que P_1 passe par `L1` : ✓
     - Soit P_1 exécute `L1` et `L2`. Alors `turn = 0` et P_0 peut entrer en SC.

□

Clarifications sur la preuve :

1. Dans le premier cas, on commence par supposer que les deux threads sont en section critique en même temps (i.e. l'algorithme de Peterson n'assurerait pas d'exclusion mutuelle). Alors P_0 et P_1 sont passés par `L1`, donc `want[i] = true` pour chaque i. Or si les deux sont en section critique, cela veut dire aussi que la condition de `L3` pour chaque thread est fausse. Par ce qu'on vient de dire, la seule possibilité pour cela est donc `turn = 0` dans le cas de P_0, et `turn = 1` dans le cas de P_1 : c'est absurde, une variable ne peux pas valloir deux choses différentes en même temps.
2. La preuve de l'absence d'interblocage est similaire à la précédente, à

ceci près que cette fois-ci la condition dans `L3` est vraie pour les deux threads, et cela amène à `turn = 1` pour P_0 et `turn = 0` pour P_1.
On peut noter que la supposition que P_0 et P_1 sont bloqués en `L3` est générale, puisque c'est la seule ligne bloquante du programme.

3. Pour prouver l'attente bornée, on suppose que P_0 est bloqué en `L3`, et donc que P_1 est en plein milieu de sa section critique. On montre alors qu'après un seul passage de P_1 en section critique, P_0 sera débloqué.
Pour cela on commence par étudier le cas où P_1 s'arrête après être sorti de la section critique ; il est clair qu'après ça, P_0 peut entrer en section critique. Dans l'autre situation, en fonction de l'ordonnancement de P_0 par rapport à P_1, on retombe soit dans le cas précédent, soit dans un cas où c'est au tour de P_0 d'entrer en section critique et donc il le fait.

4 Plusieurs threads [BAR]

On présente maintenant l'algorithme de la boulangerie de Lamport, qui est une généralisation de l'algorithme de Peterson pour plus de deux threads.

Algorithme de Lamport :

```
void lock(int i) {
    // Acquérir une priorité
    Acq[i] = true;
    n[i] = max(n[j]) + 1;
    Acq[i] = false;

    // Attente SC
    foreach(j) {
        while(Acq[j]);
        while(n[j] != 0 && (n[j], j) <= (n[i], i));
    }
}

void unlock(int i) {
    n[i] = 0;
}
```

L'idée est de fusionner `want` et `turn` en un seul tableau `n`, qui contient pour chaque thread i voulant entrer en section critique une priorité (si i ne veut pas entrer en section critique alors `n[i] = 0`). L'algorithme fonctionne en deux parties : la première consiste à acquérir un numéro plus élevé que tous ceux qui existent alors (on ne double aucun thread), et la seconde à attendre que tous les threads de priorité plus élevée (i.e. de numéro inférieur) soient sortis de section critique.

Organisation du tableau

Titre	a. Approche naïve	3. Correction	
1. Contexte			4. >2 threads
2. Pour 2 threads	b. Peterson		

Remarques

— On a mentionné le concept d'ordonnancement sans vraiment le définir : on peut soit en parler dans le plan soit laisser la porte ouverte à des questions.

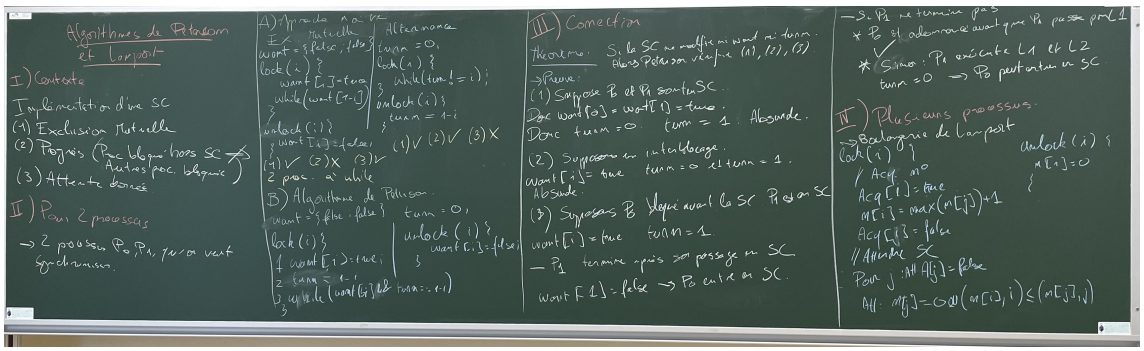
— L'algorithme de Peterson n'est pas utilisé en pratique, pour plusieurs raisons : il ne permet de synchroniser que deux processus, il utilise de l'attente active, et les optimisations des compilateurs modernes peuvent le rendre incorrect. Par exemple en voyant l'allocation de `1 - i` à `turn` puis la condition `turn == 1 - i` juste après, un compilateur pourrait supprimer l'allocation et remplacer la condition par `true`.

— Il est légitime après cela de se demander à quoi sert cet algorithme et pourquoi le présenter. On pourrait répondre que l'intérêt est pédagogique : on comprend comment mettre en place des solutions de synchronisation et les difficultés que cela présente.

— Le développement utilise un mélange de [**BAR**] et de [**TOR**] : le Barra donne une preuve très compréhensible de Peterson et donne l'algorithme de Lamport (c'est un des seuls livres à le faire ; il en donne la preuve aussi, ne

pas hésiter à regarder pour anticiper les questions du jury), et la tortue donne les versions fausses de Peterson ainsi que les propriétés d'une section critique.
— La colonne 1 ne contient pas beaucoup de texte, il est possible de la réduire un peu en taille.
— Il faut faire bien attention aux mots employés pendant le développement : on parle ici bien de *threads* et pas de *processus* au sens général, puisqu'on utilise directement des variables partagées. Tout fonctionne pareil avec des processus qui écriraient dans un pipe, mais c'est plus simple vu avec des threads.

Exemple de tableau



Références

[TOR] *" La Tortue MPI " Informatique - MP2I/MPI - CPGE 1re et 2e années - Cours et exercices corrigés*, Balabonski, Conchon, Filliâtre, Nguyen, Sartre

[BAR] *Informatique MP2I et MPI - CPGE 1re et 2e années*, Vincent Barra