TP14 - Mutex

L'objet de ce TP est l'implémentation d'un *mutex* puis la présentation de l'algorithme de Peterson. Ce dernier étant limité à deux fils (*threads*), la fin du document présente une solution qui dépasse cette limite : l'*algorithme de la boulangerie de Lamport.*

Préliminaires

On rappelle qu'un programme est *concurrent* s'il contient plusieurs *fils* (ou *threads*), s'exécutant chacun séquentiellement mais dont les exécutions peuvent être entrelacées ou simultanées. La *programmation parallèle* exploite la concurrence et la présence de plusieurs fils d'exécution (processeurs) pour accélérer un calcul (en un sens très large de calcul).

Question 1. On donne le programme OCaml suivant (fichier joint doc1.zip).

```
let n = 10_000_000
let p = 5
let nb_fils = 3

let f index =
  Printf.printf "Le fil %d a démarré\n" index;
  for i = 1 to n * p do
    if i mod n = 0
    then Printf.printf "Le fil %d a atteint %d.\n" index i
  done

let main () =
  Printf.printf "Initialisation des threads\n";
  let fils = Array.init nb_fils (fun i -> Thread.create f i) in
    Printf.printf "Démarrage des threads\n";
  for i = 0 to nb_fils - 1 do
    Thread.join fils.(i)
  done;
  Printf.printf "Fin des threads\n"

let () = main ()
```

Compiler et exécuter ce code plusieurs fois. Quelle(s) propriété(s) met-il en évidence ?

Question 2. On considère à présent le programme C suivant (fichier joint doc2.zip).

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;

void *increment(void *arg){
  int n = *(int*)arg;
  for (int i = 1; i <= n; i++) {
    counter++;
  }
  return NULL;
}

int main(void){
  int N = 1000000;
  pthread_t t1, t2;
  pthread_create(&t1, NULL, increment, &N);
  pthread_create(&t2, NULL, increment, &N);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Counter = %d\n", counter);
  return 0;
}
```

Compiler et exécuter ce code plusieurs fois. Quelle(s) propriété(s) met-il en évidence ?

Problème de l'exclusion mutuelle

Les observations faites dans la partie précédente sont liées à l'existence d'une *course critique*[1]. On dit qu'il y a une *course critique* dans un programme dès que :

- au moins deux fils peuvent accéder simultanément à un emplacement mémoire ;
- au moins l'un de ces accès est une écriture.

De manière générale, il faut considérer une course critique comme une erreur. Un bloc de code durant l'exécution duquel un fil doit posséder un accès *exclusif* à une ressource partagée est appelé *section critique*. Garantir l'exclusivité de cet accès, et éventuellement d'autres propriétés, est le problème dit de l'*exclusion mutuelle*. Pour gérer ce problème, une solution consiste à utiliser un *mutex*[2], primitive de synchronisation qui offre au minimum l'interface suivante (où `m` est un *mutex*) :

- une fonction `CreateLock()` qui renvoie un nouveau *mutex* initialement libre ;
- une fonction `Lock(m)`, qui ne renvoie rien ;
- une fonction `UnLock(m)`, qui ne renvoie rien non plus.

Sa sémantique est la suivante :

- à tout moment, le verrou est tenu par zéro ou un fil ;
- un fil ne possédant pas le verrou peut appeler la fonction `Lock` ;
- cet appel termine au bout d'un temps indéterminé, et possiblement infini ;
- quand l'appel termine, le fil ayant effectué l'appel possède le verrou ;
- si un fil possède le verrou, il en garde la possession jusqu'à ce qu'il appelle `UnLock`.

Un *verrou* satisfait :

- l'**exclusion mutuelle** s'il est toujours tenu par au plus un fil ;
- l'**absence d'interblocage** (*deadlock-freedom*) si, lorsqu'au moins un fil tente d'acquérir ou de relâcher le verrou, au moins un fil finit par y parvenir ;
- l'**absence de famine** (*starvation-freedom*) si, lorsqu'un certain fil tente d'acquérir ou de relâcher le verrou, ce fil finit par y parvenir.

Remarques

- On suppose implicitement qu'un fil qui a acquis le verrou tente de le relâcher après un temps fini : s'il y a une boucle infini dans la section critique, les autres fils seront bien sûr bloqués mais ce n'est pas considéré comme un interblocage dû au verrou.
- De manière évidente, l'absence de famine implique l'absence d'interblocage.
- La propriété d'exclusion mutuelle est un exemple de propriété de *safety* : elle affirme que quelque chose ne peut jamais se produire.
- Les deux autres propriétés sont, elles, des propriétés de *liveness* : elles affirment que quelque chose finira forcément par se produire.
- Un verrou qui ne satisfait pas l'exclusion mutuelle n'est pas un verrou, et un verrou qui ne satisfait pas l'interblocage n'est pas un verrou utilisable. L'absence de famine, en revanche, peut ne pas être satisfaite dans certaines implémentations pourtant utilisées - et inversement, des implémentations peuvent offrir des garanties d'équité plus fortes (par exemple, que l'acquisition du verrou se fait dans l'ordre des demandes).

Question 3. On donne le programme OCaml suivant (fichier joint `doc3.zip`).

```
let counter = ref 0
let lock = Mutex.create ()

let multiple_increment n =
  for i = 0 to n - 1 do
    Mutex.lock lock;
    (* start of critical section *)
    counter := !counter + 1;
    (* end of critical section *)
```

1. *Race condition* en anglais.
2. Pour *MUTual EXclusion* en anglais. On parle également de *verrou*.

```
    Mutex.unlock lock
  done

let main () =
  let n = 1_000_000 in
  let t0 = Thread.create multiple_increment n in
  let t1 = Thread.create multiple_increment n in
  Thread.join t0;
  Thread.join t1;
  Printf.printf "counter = %d\n" !counter

let () = main ()
```

Compiler et exécuter ce code plusieurs fois. Commenter.

Question 4. On définit le type `atomic_counter` suivant et la fonction d'initialisation `new_counter`. Définir les fonctions `increase`, `decrease`, `get` et `set` qui pourraient constituer l'interface de ce type de données.

```
type atomic_counter = {
  lock : Mutex.t;
  mutable value : int;
}
```

```
let new_counter initial_value = {
  lock = Mutex.create ();
  value = initial_value
}
```

Question 5. On souhaite à présent définir une fonction `swap : counter -> counter -> unit` permettant d'échanger la valeur de deux compteurs de manière atomique. Pour chacune des versions proposées ci-dessous, déterminer si elle remplit correctement sa fonction.

```
let swap1 c c' =
  let v_c = get c in
  set c (get c');
  set c' v_c
```

```
let swap2 c c' =
  let v_c = get c in
  let v_c' = get c' in
  set c v_c';
  set c' v_c
```

```
let swap3 c c' =
  Mutex.lock c.lock;
  let v_c = get c in
  set c (get c');
  set c' v_c;
  Mutex.unlock c.lock
```

```
let swap4 c c' =
  Mutex.lock c.lock;
  Mutex.lock c'.lock;
  let tmp = c.value in
  c.value <- c'.value;
  c'.value <- tmp;
  Mutex.unlock c'.lock;
  Mutex.unlock c.lock
```

Question 6. La fonction `Mutex.try_lock` prend en entrée un `Mutex.t` et a le comportement suivant :

- elle tente d'acquérir le verrou ;
- si elle réussit, elle renvoie true et le fil qui a fait l'appel tient désormais le verrou ;
- si elle échoue, elle renvoie false (sans bloquer).

En utilisant cette fonction, proposer une version correcte de `swap`.

Question 7. En pratique, cette dernière version fonctionnerait très mal, voire pas du tout. Pourquoi ?

Algorithme de Peterson

Essayons de concevoir un *mutex* avec comme seule brique de base des registres atomiques booléens. On se limite au cas de deux fils d'exécution, l'interface souhaitée étant de la forme suivante :

- *CreateLock()*, qui renvoie un nouveau verrou libre ;
- *Lock(m,t)*, où *m* est le verrou et *t* est l'identifiant du fil, qui vaut nécessairement *0* ou *1* ;
- *UnLock(m,t)*, de même.

Ajoutons une remarque valable de manière générale pour les *mutex* : si un fil appelle *Lock* alors qu'il tient déjà le verrou, ou *UnLock* alors qu'il ne le tient pas, le comportement est non défini. On suppose que chacun des fils utilise correctement le *mutex*, c'est-à-dire qu'il entoure sa section critique d'un appel à *Lock* et d'un appel à *UnLock*. On envisage ci-dessous cinq algoritmes dont on montre que seul le dernier est correct. Il correspond à l'*algorithme de Peterson*.

Question 8. Dans un premier algorithme, on utilise un tableau *flag* de deux booléens : *flag[i]* est vrai si et seulement si le fil *i* possède le verrou. Ce dernier est un type enregistrement avec un seul champ *flag*. Montrer que cette tentative ne respecte pas la propriété d'exclusion mutuelle.

Algorithme 1 : Premier *mutex* incorrect

```
1 fonction CreateLock()
2     renvoyer {flag = [false, false]}
3 fonction Lock(m,t)
4     other ← 1 − t
5     tant que m.flag[other] faire
6         rien // attente active
7 fonction UnLock(m,t)
8     m.flag[t] ← false
```

Question 9. Un deuxième algorithme utilise toujours un tableau de deux booléens dont la signification change : il n'indique pas si le fil possède le verrou, mais si le fil souhaite l'acquérir.

Algorithme 2 : Deuxième *mutex* incorrect

```
1 fonction CreateLock()
2     renvoyer {want = [false, false]}
3 fonction Lock(m,t)
4     other ← 1 − t
5     m.want[t] ← true
6     tant que m.want[other] faire
7         rien // attente active
8 fonction UnLock(m,t)
9     m.want[t] ← false
```

Cette version respecte la condition d'exclusion mutuelle. Pour le prouver, on peut procéder par l'absurde, et considérer la dernière exécution de *Lock* par chacun des fils avant d'arriver dans la situation où les deux détiennent le verrou (et où l'exclusion mutuelle est donc violée). On note $write_i(x = v)$ pour l'action, par le fil i, d'écrire la valeur v dans x, et $read_i(x == v)$ pour l'action de lire la valeur v depuis x. Comme on a supposé les lectures et écritures atomiques, chacune de ces actions s'exécute instantanément (ou plutôt, on peut faire comme si elle s'exécutait instantanément). On note $A \to B$ pour indiquer que l'instant correspondant à l'action A précède nécessairement celui associé à l'action C : cela revient essentiellement à dire que B est accessible depuis A dans un graphe de dépendance. On a, à la lecture du code :

$$write_0(\text{want}[0] = \text{true}) \to read_0(want[1] == \text{false}) \to SC_0$$
$$write_1(\text{want}[1] = \text{true}) \to read_1(want[0] == \text{false}) \to SC_1$$

où SC_i indique que le fil T_i entre dans sa section critique (*i.e.* acquiert le verrou).

Une fois que $want[1]$ est mis à true, il reste à true jusqu'à ce que T_1 libère le verrou. Comme les sections critiques se chevauchent (il existe un instant où les deux fils possèdent le verrou), T_0 doit avoir lu $want[1]$ avant que T_1 ne l'ait fixé à true : on peut donc ajouter un arc $read_0(want[1] == \text{false}) \to write_1(want[1] = \text{true})$. De même, on a : $read_1(want[0] == \text{false}) \to write_0(want[0] = \text{true})$. En combinant, on obtient :

$$write_0(want[0] = \text{true}) \to read_0(want[1] == \text{false}) \to write_1(want[1] = \text{true}) \to read_1(want[0] == \text{false}) \to write_0(want[0] = \text{true})$$

Mais la relation précède nécessairement est un ordre partiel (le graphe est acyclique) : c'est absurde.
Montrer que cette version ne satisfait pas la condition d'absence d'interblocage.

Question 10. Un troisième algorithme utilise une variable *turn* qui indique quel fil possède la priorité. De manière arbitraire, le fil 0 a la priorité au départ. Montrer que cette version satisfait l'exclusion mutuelle mais pas l'absence d'interblocage.

Algorithme 3 : Troisième *mutex* incorrect

```
1 fonction CreateLock()
2     renvoyer {turn = 0}
3 fonction Lock(m,t)
4     other ← 1 − t
5     tant que m.turn = other faire
6         rien // attente active
7 fonction UnLock(m,t)
8     m.turn ← 1 − t
```

Question 11. Dans un quatrième algorithme, on combine les deux idées : deux booléens indiquant si chaque fil souhaite acquérir le verrou, et une variable indiquant qui a la priorité. Quel est le problème avec cette version ?

Algorithme 4 : Quatrième *mutex* incorrect

```
1  fonction CreateLock()
2      renvoyer {turn = 0; want = [false, false]}
3  fonction Lock(m,t)
4      m.want[t] ← true
5      m.turn ← t
6      tant que m.turn = other et m.want[other] faire
7          rien // attente active
8  fonction UnLock(m,t)
9      m.turn ← 1 − t
10     m.want[t] ← false
```

Une solution pertinente

L'algorithme de Peterson reprend la dernier algorithme, sauf que chaque fil commence par céder la priorité.

Algorithme 5 : Algorithme de Peterson

```
1 fonction CreateLock()
2     renvoyer {turn = 0; want = [false, false]}
3 fonction Lock(m,t)
4     m.want[t] ← true
5     m.turn ← 1 − t
6     tant que m.turn = other et m.want[other] faire
7         rien // attente active
8 fonction UnLock(m,t)
9     m.want[t] ← false
```

L'*algorithme de Peterson* est correct.

- Il satisfait la condition d'exclusion mutuelle.
- Il garantit l'absence d'interblocage.
- Il garantit l'absence de famine.

Démonstration

Exclusion mutuelle. On raisonne par l'absurde et l'on considère la dernière exécution de *Lock* par chacun des fils avant la violation de l'exclusion mutuelle. À la lecture du code, on a :

$$write_0(want[0] = \text{true}) \rightarrow write_0(turn = 1) \rightarrow read_0(turn) \rightarrow read_0(want[1]) \tag{1}$$
$$write_1(want[1] = \text{true}) \rightarrow write_1(turn = 0) \rightarrow read_1(turn) \rightarrow read_1(want[0]) \tag{2}$$

L'un des deux fils, disons T_0, est le dernier à avoir écrit dans la variable turn. On a alors :

$$write_1(turn = 0) \rightarrow write_0(turn = 1) \tag{3}$$

La lecture de *turn* par le fil T_0 a donc nécessairement renvoyé la valeur 1, et comme T_0 a quand même acquis le verrou, cela signifie (à la lecture du code) que $read_0(want[1])$ a renvoyé false. On a donc :

$$write_0(turn = 1) \rightarrow read_0(want[1] == \text{false}) \tag{4}$$

En combinant 2, 3 et 4, on obtient :

$$write_1(want[1] = \text{true}) \rightarrow write_1(turn = 0) \rightarrow write_0(turn = 1) \rightarrow read_0(want[1] == \text{false}) \tag{5}$$

C'est absurde, puisqu'il n'y a aucune écriture sur want[1] entre le moment où T_1 écrit true et celui où T_0 lit false.

Absence de famine. Par l'absurde, supposons qu'un appel à *Lock* ne termine jamais pour le fil T_0 (le problème est symétrique pour T_1). T_0 est donc en train d'exécuter le while et d'attendre que *turn* devienne égal à 0 ou que want[1] devienne égal à false. Notons que *want*[0] vaut true puisque T_0 l'a fixé à cette valeur avant de rentrer dans le *while* et que T_1 ne modifie jamais cette case. Regardons ce que T_1 peut faire pendant ce temps.

- Si T_1 commence une exécution de *Lock*, il fixe turn à 0. Cette valeur restera à 0 jusqu'à ce que T_0 la modifie, donc la prochaine fois que T_0 testera la condition du *while*, il pourra en sortir et acquérir le verrou, ce qui contredit notre hypothèse. Donc T_1 ne peut pas commencer d'exécution de Lock si T_0 est bloqué indéfiniment.
- Si T_1 termine une exécution de *Lock* et n'en commence donc pas une nouvelle d'après le point précédent, alors il fixe *want*[1] à false et ne le modifie plus. Donc T_0 va pouvoir sortir du *while*, ce qui est absurde.
- Si T_1 n'a jamais exécuté *Lock*, on conclut de même.
- Donc T_1 est nécessairement bloqué à l'intérieur d'une exécution de *Lock*, c'est-à-dire dans la boucle l. Mais cela signifie que *turn* vaut 0, ce qui est absurde.

Remarques

Limitation à deux fils. Bien évidemment, un verrou ne fonctionnant que pour deux fils ne présente qu'un intérêt limité. La partie suivante présente l'algorithme dit de *la boulangerie de Lamport*, qui permet de s'affranchir de cette limite ; les autres problèmes signalés ici persistent, en revanche.

Attente active. Dans l'algorithme de Peterson, un fil bloqué en attendant d'acquérir le verrou n'est pas *bloqué* au sens du système d'exploitation : il exécute en permanence des instructions (plus précisément, il est dans une boucle extrêmement serrée de quelques instructions processeur). Un verrou qui utilise cette technique est couramment appelé un spinlock. Il est tout à fait raisonnable de faire un peu d'attente active, mais le faire de manière non bornée comme ici est une très mauvaise idée. Le cas le plus absurde est celui où le fil qui tient le verrou ne peut pas s'exécuter car l'autre fil est en train de tourner en rond pour essayer d'acquérir ce verrou…

Pas de communication avec le système. Il y a (presque) systématiquement plus de fils actifs sur une machine que de processeurs (en tenant compte de tous les programmes qui sont en train de s'exécuter). Répartir le temps processeur entre ces différents fils est le travail de l'ordonnanceur (*scheduler*), qui est l'un des composants majeurs du noyau du système d'exploitation. Dans le cas d'un *mutex*, il est fortement souhaitable de communiquer avec ce *scheduler*, pour qu'il évite par exemple de mettre un fil en sommeil alors qu'il tient un *mutex* qu'un autre fil tente d'acquérir. Écrire un bon *mutex* sans interagir avec le noyau est essentiellement impossible.

Ça marche en théorie, mais…. Un point qui est quand même important : si vous traduisez directement le pseudo-code de l'algorithme de Peterson en C, vous vous rendrez assez vite compte que cela ne fonctionne pas ! Le problème le plus courant sera un interblocage, mais un non respect de l'exclusion mutuelle est également possible. C'est quelque peu gênant, vu que nous avons prouvé la correction de cet algorithme…La preuve est pourtant correcte ; ce sont les hypothèsesfaites qui ne sont pas vérifiées en pratique, du moins si l'on ne prend aucune précaution.

Il existe des instructions conçues pour ce genre de choses. L'intérêt de l'algorithme de Peterson est qu'il nécessite seulement des registres atomiques : des variables sur lesquelles les écritures et les lectures prennent effet de manière ponctuelle, et sur lesquelles il y a une notion d'ordre. Le point précédent montre que cette hypothèse n'est pas forcément

vérifiée avec un processeur et un compilateur moderne - mais d'un autre côté, ces processeurs modernes offrent tous des opérations atomiques plus riches qu'une simple lecture ou écriture. Ne pas en tirer parti quand on implémente un *mutex* est regrettable !

Don't try this at home. De manière générale, écrire ses propres primitives de synchronisation (*mutex*, mais aussi *sémaphores*, *condition variables*, etc) est une très mauvaise idée, encore nettement plus mauvaise que de ré-écrire sa propre fonction de tri plutôt que d'utiliser celle fournie par la bibliothèque standard de votre langage préféré. Cela n'empêche bien sûr pas qu'il puisse être formateur d'étudier quelques manières un peu naïves de le faire.

Algorithme de la boulangerie de Lamport

Étendre l'algorithme de Peterson au cas où n fils doivent partager l'accès à une ressource n'est pas évident. Le problème principal est que si l'on imagine par exemple un tableau de n booléens pour remplacer *want*, on ne peut pas raisonnablement espérer lire l'intégralité du tableau en une seule opération atomique : on va devoir faire une boucle, et entre le moment où l'on a constaté que la case d'indice zéro valait false et celui où l'on est arrivé à la fin du tableau, le contenu de la case zéro a tout à fait pu changer...

L'*algorithme de la boulangerie de Lamport* résout ce problème. On suppose pour simplifier que les fils qui vont utiliser le verrou sont numérotés $0, \dots, n-1$, et que le numéro du fil est passé comme argument à *Lock* et *UnLock*.

- Il y a deux tableaux de taille n : *want* qui contient des booléens et *ticket* qui contient des entiers (positifs) ;
- *want*$[i]$ vaudra true quand le fil i veut acquérir le verrou, ou le possède (comme dans l'algorithme de Peterson) ;
- *ticket*$[i]$ est une sorte de *numéro d'ordre* : si i et j sont tels que *ticket*$[i] <$ *ticket*$[j]$, alors le fil i est prioritaire sur le fil j.
- Quand un fil veut acquérir le verrou, il passe d'abord dans une section, que nous appellerons *vestibule*, pendant laquelle il ne peut être bloqué (mais peut bien sûr être interrompu). Dans cette section, il exécute les actions suivantes :
  - il lève son drapeau (il écrit true dans *want*$[i]$) ;
  - il lit toutes les valeurs des *ticket*$[j]$, dans un ordre quelconque, et en détermine le maximum m ; il prend alors le premier « ticket disponible » en écrivant $m+1$ dans *ticket*$[i]$.
- Le fil arrive ensuite dans la *section d'attente*. L'idée est alors que le fil détermine si, parmi tous les fils voulant rentrer en section critique (ou y étant déjà), c'est lui qui est «le plus prioritaire», et qu'il fasse une attente active jusqu'à ce que ce soit le cas. Pour cela, il lit les valeurs de *ticket*$[j]$ pour tous les j tels que *want*$[j]$ vaille true : si son ticket est strictement plus petit que toutes ces valeurs, alors il prend le verrou.
- Il y a cependant un problème : il est tout à fait possible que plusieurs fils possèdent le même ticket. En effet, un fil i peut parcourir le tableau ticket et trouver m comme maximum. En même temps, ou en tout cas avant que i ait eu le temps d'écrire $m+1$ dans *ticket*$[i]$, un autre fil j parcourt le tableau, et trouve également m comme maximum ; les deux fils choisissent ensuite le ticket $m+1$. Ce problème n'est pas immédiat à résoudre :
  - si un fil ne peut rentrer que si son ticket est strictement plus petit que ceux des autres fils en attente, on aura un interblocage en cas d'égalité ;
  - si d'un autre côté on autorise un fil à rentrer quand son ticket est inférieur ou égal à tous les autres, on n'assure plus l'exclusion mutuelle.

Une solution est d'utiliser l'identifiant du fil pour départager les *ex æquo* : on compare les couples $(ticket[i], i)$ lexicographiquement. Il est alors impossible que deux fils différents aient la même priorité. On obtient l'algorithme suivant :

Algorithme 6 : Algorithme de la boulangerie de Lamport

```
1  fonction CreateLock(n)
2      want ← [false, ..., false]
3      ticket ← [0, ..., 0]
4      renvoyer {want, ticket}
5  fonction Lock(m,i)
6      m.want[i] ← true
7      ticket[i] ← 1 + max(ticket)
8      tant que il existe j tel que want[j] = true et (ticket[j], j) <_lex (ticket[i], i) faire
9          rien
10 fonction UnLock(m,i)
11     m.want[i] ← false
```

Propriété 1 - Pour tout fil i, le numéro de ticket ticket $[i]$ est croissant au cours du temps.

Démonstration. Le ticket n'est jamais remis à zéro, donc au passage suivant dans le vestibule ticket[i] fait partie des valeurs lues par le fil i pour déterminer son ticket, qui augmentera donc au moins de un.

Propriété 2 - L'algorithme de la boulangerie est sans interblocage.

Démonstration. Supposons qu'il y ait interblocage : au moins un fil cherche à acquérir le verrou, et aucun n'y parvient en temps fini. Au bout d'un certain temps, tous les fils cherchant à acquérir le verrou se retrouvent en section d'attente : il y a alors parmi eux un unique fil T_i tel que le couple (ticket $[i], i$) soit minimal : ce fil n'attend pas, et obtient donc le verrou.

Propriété 3 - Si T sort du vestibule avant que $T'n'y$ rentre, alors T' obtiendra un ticket strictement supérieur à celui de T.

Démonstration. Comme T est sorti du vestibule, on est sûr qu'il a écrit son nouveau numéro de ticket dans le tableau. T' lira forcément ce numéro lors de son calcul de maximum (puisqu'il n'a pas encore commencé la lecture), et prendra donc un ticket strictement plus grand.

Propriété 4 - L'algorithme de la boulangerie est « premier arrivé, premier servi» : si T sort du vestibule avant que T' n'y entre, alors T obtiendra le verrou avant T'.

Démonstration. C'est une conséquence immédiate de la propriété précédente : T' aura un ticket strictement plus grand que celui de T, et ne pourra donc pas sortir de la section d'attente tant que T y est.
Remarquons que cette propriété garantit l'absence de famine : un fil T obtiendra le verrou après avoir attendu, au pire, tous les fils en attente et tous ceux dans le vestibule.

Propriété 5 - L'algorithme de la boulangerie satisfait l'exclusion mutuelle.

Démonstration. Par l'absurde, supposons que T_i et T_j sont tous les deux dans la section critique (possèdent tous les deux le verrou) et que $(\mathit{ticket}[i], i) <_{\text{lex}} (\mathit{ticket}[j], j)$) (sans perte de généralité). Pour sortir de sa section d'attente, T_j doit avoir lu que $(\mathit{ticket}[j], j) <_{\text{lex}} (\mathit{ticket}[i], i)$ ou que $\mathit{want}[i] = \text{false}$. Le premier cas est impossible :

- si T_j ne peut avoir lu la nouvelle valeur de ticket $[i]$, puisqu'on a supposé $(\mathit{ticket}[i], i) <_{\text{lex}} (\mathit{ticket}[j], j)$;
- si T_j a lu l'ancienne valeur x de $\mathit{ticket}[i]$, la propriété 1 permet de conclure.

Donc T_j a lu $\mathit{want}[i] = \text{false}$ dans la section d'attente. Cela signifie que T_j était déjà en section d'attente quand T_i a commencé à lire les tickets, donc T_i a lu la valeur actuelle de ticket $[j]$ et choisi $\mathit{ticket}[i] \geqslant 1 + \mathit{ticket}[i]$, ce qui contredit notre hypothèse.

Remarque. Tout comme l'algorithme de Peterson, en pratique, cet algorithme n'est pas utilisé. Toutefois, l'un de ses défauts est partagé par des algorithmes qui sont réellement utilisés : les tickets ne font que croître, et comme les entiers sont de taille fixée, ils finiront par déborder (et soit revenir à zéro, soit passer en négatif, suivant le type d'entier). Pour des entiers 32 bits, cela peut être un vrai souci.