TD 3 - ASR7 Programmation concurrente

25 avril 2017

I Lamport's bakery algorithm

En 1974, Leslie Lamport apporte une nouvelle solution au mutex (tiré de wikipédia) :

```
Initialiser toutes les cases de COMPTEUR à 0.
Initialiser toutes les cases de CHOIX à 0 (ou FAUX).

CHOIX[ID] = 1

MAX = 0
// Calcul de la valeur maximale des tickets des autres threads
POUR J = 0 à N - 1 FAIRE
  CJ = COMPTEUR[J]
  SI MAX < CJ ALORS
     MAX = CJ
  FIN SI
FIN POUR J
// Attribution du nouveau ticket
COMPTEUR [ID] = MAX + 1

// Fin de la phase d'attribution du ticket
CHOIX[ID] = 0

// Boucle sur tous les fils d'exécution
POUR J = 0 à N - 1 FAIRE
  // On attend que le fil considéré (J) ait fini de s'attribuer un ticket.
  TANT QUE CHOIX[J] == 1 FAIRE /* (1) */
     RIEN
  FIN TANT QUE
  // On attend que le fil courant (ID) devienne plus prioritaire que le fil considéré (J).
  TANT QUE COMPTEUR[J] <> 0 ET /* (2) */
     (COMPTEUR[J] < COMPTEUR[ID] OU /* (3) */
     (COMPTEUR[J] == COMPTEUR[ID] ET J < ID)) /* (4) */
  FAIRE
     RIEN
  FIN TANT QUE
FIN POUR J

Section Critique

// Sortie de section critique
COMPTEUR[ID] = 0
```

Q.I.1) - Montrez que l'algorithme garantit l'exclusion mutuelle de N fils d'exécution. Précisez sous quelles hypothèses.

(D'après wikipedia)
Les fils doivent posséder des identifiants (habituellement, un entier) comparables (c'est-à-dire éléments d'un ensemble munis d'une relation d'ordre, supposée totale).
L'algorithme requiert également une zone de mémoire partagée servant à stocker deux tableaux comportant N éléments :
Le premier est un tableau de booléens, appelé CHOIX. Le second est un tableau d'entiers naturels, appelé COMPTEUR.
La valeur particulière 0 indique, dans ce dernier tableau, qu'un fil ne souhaite pas entrer en section critique. Les autres valeurs représentent des numéros de ticket potentiels.
Il n'est pas nécessaire que les opérations de lecture et d'écriture sur les tableaux partagés soient réalisées de manière atomique. Pour faciliter la compréhension, le lecteur peut dans un premier temps supposer que ces opérations sont atomiques, puis se référer à la section Historique et Propriétés pour obtenir des précisions sur la validité de ce point.
L'algorithme ne présuppose rien sur la vitesse d'exécution relative des différents fils. Il garantit qu'un seul fil exécute la section critique à la fois. Il n'introduit pas d'interblocage ou de famine.
Première phase Cette portion de code vise à l'attribution d'un ticket. Le principe en est de regarder tous les numéros déjà attribués et de s'en attribuer un nouveau plus grand que les autres. Cependant, il est possible que plusieurs fils d'exécution exécutent le code de cette phase de manière concurrente. Ils peuvent alors réaliser le même calcul du maximum et s'attribuer le même ticket. Ce cas est pris en compte lors de la phase suivante.
En toute généralité, si on ne suppose pas que les lectures et écritures sont atomiques, il est même possible que des fils exécutant de manière concurrente la première phase obtiennent des valeurs de ticket différentes. Ces valeurs sont cependant nécessairement plus grandes que celles des tickets d'autres fils ayant atteint la deuxième phase avant que ne commence l'attribution des tickets pour les premiers.
Deuxième phase Cette phase correspond, dans l'analogie développée plus haut, à l'attente de son tour. Le principe est d'examiner tous les autres fils d'exécution et de tester s'ils ont un numéro de ticket inférieur, auquel cas ils doivent passer dans la section critique en premier.
La comparaison des tickets a lieu à la ligne indiquée par (3). Comme expliqué précédemment, il est possible que deux fils d'exécution s'attribuent le même ticket. Dans ce cas, le plus prioritaire est celui qui possède l'identifiant le plus petit (voir la ligne (4)). La priorité est donc évaluée selon l'ordre lexicographique sur les couples (COMPTEUR[J], J). Seuls les fils souhaitant réellement entrer en section critique sont pris en compte (voir la ligne (2)).
Un point essentiel au bon fonctionnement de l'algorithme, et vraisemblablement l'un des plus difficiles à comprendre, est l'utilisation du tableau CHOIX, afin d'éviter de prendre en compte par erreur une ancienne valeur de ticket pour les autres fils (ligne (1)). Supposons que 2 fils J et ID, avec J < ID, exécutent la boucle de la première phase de manière concurrente et qu'ils se voient attribuer le même numéro de ticket. Pour simplifier, nous considérerons que tous les autres fils sont hors de la section critique et ne cherchent pas à y entrer. Supposons également que, alors que le fil ID exécute la deuxième phase, le fil J n'a pas encore exécuté l'opération de mise à jour de son ticket dans la première phase ($COMPTEUR[J] = MAX + 1$ n'a pas été exécutée). Supposons enfin que l'on a retiré la boucle d'attente sur la valeur de CHOIX[J] (ligne (1)). Dans ce cas, le fil ID lit COMPTEUR[J] (ligne (2)), qui vaut 0, sans prendre en compte la valeur de CHOIX[J]. Cette valeur de 0 est censée signifier que le fil J est hors de la section critique et ne cherche pas à y entrer. Le fil ID en déduit qu'il est prioritaire par rapport à J et entre dans la section critique. Le fil J, poursuivant son exécution, met à jour COMPTEUR[J] et entre dans la deuxième phase. Il remarque alors qu'il a le même numéro de ticket que le fil ID et compare leurs identifiants. Comme J < ID, J est prioritaire. Il finit donc par entrer dans la section critique. Finalement, dans ce cas, les fils J et ID peuvent tous les deux entrer dans la section critique, ce que l'algorithme cherche justement à éviter.

Remarques :

— Aucune famine n'est possible : Le premier point résulte de l'obligation pour un fil d'obtenir un nouveau ticket afin de rentrer dans la section critique une nouvelle fois. Ce nouveau ticket a nécessairement un numéro d'ordre strictement supérieur à celui de l'ancien ticket, à moins que les autres fils ne soient tous hors de la section critique et ne cherchent pas à y entrer. Un fil qui cherche à entrer dans la section critique est donc certain d'y entrer au plus tard lorsque tous les autres fils ayant exécuté la première phase de manière concurrente vis-à-vis de lui sont passés dans la section critique une seule fois.

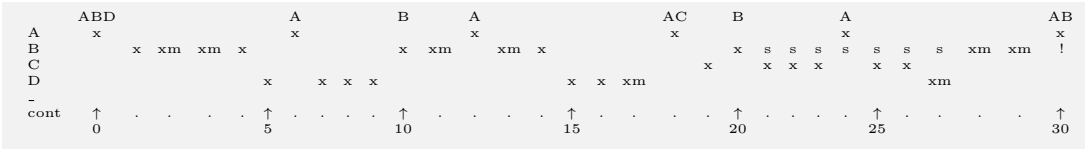
— Les opérations de lecture et d'écriture dans les tableaux partagés n'ont pas besoin d'être atomiques : Le second point est particulièrement remarquable et délicat à assimiler. Il résulte de plusieurs caractéristiques de l'algorithme. En premier lieu, ce dernier n'utilise pas d'écritures concurrentes aux mêmes adresses mémoire. Le fil ID, et lui seul, peut écrire dans CHOIX[ID] et COMPTEUR[ID]. En second lieu, les valeurs de CHOIX[ID] sont booléennes, ce qui implique que n'importe quel changement de valeur est visible au travers d'un seul bit, et la modification de ce dernier est nécessairement atomique. Autrement dit, les changements de CHOIX[ID] sont perçus de manière atomique. En troisième lieu, l'utilisation de CHOIX sérialise tout accès concurrent à COMPTEUR[J], à l'exception de ceux se produisant entre fils en train d'exécuter la première phase en même temps. Dans ce dernier cas, les fils peuvent obtenir des numéros de tickets différents, mais ils attendront tous mutuellement que leurs valeurs soient visibles dans la deuxième phase, et un seul fil finira par accéder à la section critique. Par ailleurs, si un fil K était déjà dans la deuxième phase avant que les premiers ne commencent l'exécution de la première phase, alors son COMPTEUR[K] a nécessairement été lu correctement pour le calcul du maximum et les nouveaux tickets sont nécessairement supérieurs à celui-ci, même si on ne peut prévoir leur ordre relatif. Une démonstration plus formelle est disponible dans l'article original de Leslie Lamport.

II Ordonancement avec des mutex

Nous utilisons un ordonnancement préemptif avec priorité [1]. Nous allons utiliser un jeux de tâches qui mélange des tâches périodiques et des tâches ponctuelles. De plus, deux tâches partagent un mutex.

Tâche	Date(s) d'arrivée(s)	Priorité	Durée	Remarque
A	0, 6, 12, 18, 24, 30	10	1	Périodique
B	0, 10, 20, 30	8	4	Périodique, à chaque itération, la tâche doit acquérir le mutex à la fin du temps 1 et la libérer à la fin du temps 3.
C	18	5	6	Ponctuelle
D	0	1	8	Ponctuelle, à la fin du temps 6, la tâche acquière le mutex et le conserve jusqu'à la fin du temps 8.

Q.II.1) - Faire l'ordonnancement de ces tâches sur 32 unités de temps.



Q.II.2) - Quel est le temps de réponse de chaque tâche ?

1. Plus la valeur de priorité est importance plus la tâche est prioritaire

— A : 1
— B : 12 (le pire est la seconde fois) Un échéance est loupée
— C : 9 (arrivé en 18 fini en 27)
— D : 28 (arrivée en 0 terminée en 28)

Q.II.3) - Que remarque-t'on aux temps 21 à 27 ?

Une inversion de priorité, B est bloquée par D qui est bloquée par C. Donc C qui est moins prioritaire que B et ne partage pas de mutex avec B passe avant B.

III Lecteur rédacteur avec tampon (CCF automne 2010)

On souhaite améliorer les performances d'une base de données. Ses tables doivent être accessibles en lecture et écriture. Pour le moment, tout est géré par des verrous de type *lecteur/rédacteur*[2]. Après différentes mesures, on s'est aperçu que l'essentiel des blocages vient des requêtes d'écriture. Les verrous ont été programmés de manière équitable, et les requêtes en écriture sont rares par rapport aux lectures, mais :

— lorsqu'une requête en écriture arrive, elle doit attendre que toutes les requêtes en lecture se terminent ce qui peut prendre du temps,
— pendant qu'elle attend, une requête en écriture bloque les nouvelles requêtes en lecture.
— si l'écriture est longue elle bloque toute autre requête pendant son exécution.

III.1 Proposition de solution

Pour y remédier, on utilisera un buffer. C'est-à-dire que la table sera dupliquée en mémoire.
— la première, la *table active* qui sert au lectures ;
— la seconde, la *table copie* qui est la seule à pouvoir être modifiée.

On utilisera à la fois des verrous de type *lecteur/rédacteur* et des verrous simples (*mutex*). Lorsqu'une requête a besoin de modifier la table :

— elle modifie la *copie* (pendant que les lectures se font sur la table *active*) ;
— elle échange les deux tables ;
— elle synchronise les modifications sur l'ancienne table *active* (pendant que les lectures utilisent l'ancienne *copie*).

Cela signifie qu'il faut faire 2 modifications au lieu d'une mais cela bloque moins les lectures, on espère donc gagner du temps.

Le principe est donc :

— Pour la lecture :

```
lecture(reqlect) {
  int table;

  // Trouver la table active par lecture d'une variable partagée
  // de gestion des tables;
  table = ChoixTableActive();

  // Accéder en lecture à la table active.
  res = LireTable(table, reqlect);

  return res;
}
```

— Pour l'écriture :

```
modifie(reqmodif) {
  // Trouver la table copie et la table active;
```

2. Plusieurs lecteurs peuvent lire ensemble, un seul rédacteur peut écrire à la fois et il interdit la lecture en même temps.

```
    tactive = ChoixTableActive();
    tcopie = ChoixTableCopie();

    // Appliquer la requête à la copie
    ModifieCopie(tcopie, reqmodif);

    // Inverser le rôle des 2 tables c'est à dire modifier
    // la variable partagée de gestion des tables;
    InverseTable();

    // Synchroniser les deux tables c'est à dire lecture de
    // tcopie (l'ancienne copie) et écriture de tactive (l'ancienne active).
    Synchronise(tactive, tcopie);
}
```

III.2 Travail à faire

Vous disposez :

- Des mutex simples et leurs primitives :
  - Mutex M pour déclarer le mutex ;
  - Lock (M) pour bloquer le passage ;
  - Unlock(M) pour débloquer le passage.
- Des mutex lecteur/rédacteur et leurs primitives :
  - MutexLR Lr pour déclarer le mutex ;
  - DebutLecture(Lr) pour annoncer le début de la lecture ;
  - FinLecture(Lr) pour annoncer la fin de la lecture ;
  - DebutEcriture(Lr) pour annoncer le début de l'écriture ;
  - FinEcriture(Lr) pour annoncer la fin de l'écriture.
- Des fonctions :
  - table = ChoixTableActive(); fonction qui renvoie le numéro de la table active ;
  - table = ChoixTableCopie(); fonction qui renvoie le numéro de la table de copie ;
  - LireTable(table, reqlect) pour appliquer la requête en lecture reqlect à la table table ;
  - Modifie(table, reqmodif) pour appliquer la modification de la requête reqmodif à la table table ;
  - InverseTable() pour inverser le rôle des tables ;
  - Synchronise(table1, table2) pour synchroniser la table table1 à partir de la table table2.

En utilisant ces primitives, vous devez :

Q.III.1) - Donner l'algorithme du lecteur : lecture(reqlect).

Q.III.2) - Donner l'algorithme du rédacteur : modifie(reqmodif).

Q.III.3) - En supposant que la modification et la synchronisation durent $10ms$ que le choix de la table active et l'inversion des tables durent $0.1ms$, que la lecture dure $20ms$. Donnez le temps nécessaire dans le pire des cas (avant et après la modification) pour :

3(a) - une requête en écriture ;

3(b) - une requête en lecture ;

3(c) - que remarque-t'on ?

```
Mutex m; // pour empêcher plusieurs modifs à la fois
MutexLR lrChoix; // pour les accès à la variable de choix
MutexLR lrTable[2]; // pour les accès à chaque tables

lecture(reqlect) {
  int table;

  DebutLecture(lrChoix);
  table = ChoixTableActive();
  FinLecture(lrChoix);

  DebutLecture(lrTable[table]);
  res = LireTable(table, reqlect);
  FinLecture(lrTable[table]);

  return res;
}

modifie(reqmodif) {
  Lock (M); //une seule écriture à la fois

  tactive = ChoixTableActive();
  tcopie = ChoixTableCopie();

  DebutEcriture(lrTable[tcopie]);
  ModifieCopie(tcopie, reqmodif) // ce thread est le seul à faire ça et à
pouvoir lire ou écrire cette table à ce moment
  FinEcriture(lrTable[tcopie]);

  DebutEcriture(lrChoix);
  InverseTable(); // ce thread est le seul à toucher à la variable partagée
  FinEcriture(lrChoix);

  DebutEcriture(lrTable[tactive]);
  Synchronise(tactive, tcopie) // idem car tactive n'est plus active.
  FinEcriture(lrTable[tactive]);

  Unlock(M); //fin de l'écriture
}
```

La dernière question est une question piège. En effet, il peut arriver qu'une requête en lecture effectue la lecture sur la copie car elle a demandé le choix avant l'inversion. Dans ce cas, elle bloque l'écriture du rédacteur ou peut être bloquée par lui.
De plus, si 2 requêtes dans ce cas arrivent au bon moment (avant le début de la fonction modifie et entre FinEcriture(lrChoix); et DebutEcriture(lrTable[tactive]);), l'écriture est bloquée 2 fois :

- requête d'écriture
  - avant : bloquée 20ms + exec 10ms = 30ms
  - après : bloquée avant l'écriture de la copie 20ms + modif 10ms + inversion 0.1ms + bloqué avant la synchro 20ms + synchro 10ms = 60,1ms exec sur la copie 10ms + inversion 0.1ms + bloqué sur l'écriture (par une requête qui vient de demander le choix) 20ms + synchro 10ms = 21ms
- requête de lecture
  - avant : bloqué par l'écriture 30ms + exec 20ms = 50ms
  - après : elle ne peut être bloquée que par soit l'inversion (20ms+0.1ms) soit par l'écriture sur la table non active (30ms+20ms)
- On remarque qu'on ne fait pas mieux pour la lecture et pire pour l'écriture. Mais c'est une valeur dans le pire des cas et les cas en question sont beaucoup moins probables.