TD 2 - LIFPCA Programmation Concurrente et Administration Système

—

Thread et concurrence

Sylvain Brandel, Yves Caniou, Guillaume Damiand, Meriem Ghali, Laurent Lefèvre, Thibaut Modrzyk, Grégoire Pichon, Alec Sadler, Florence Zara, Jerry Lacmou Zeutouo

Printemps 2024

I Retour sur le TD1

— Exercices non-faits pendant le TD1

II Le pont de Miralonde

Le pont de Miralonde est trop étroit pour que 2 voitures puissent se croiser. Vous devez mettre en place un système qui évitera tout incident. Le système se déclenchera automatiquement à l'arrivée d'une voiture à l'une des extrémités du pont. Il autorisera ou non le passage en fonction de la configuration sachant que :

— Si le pont est vide, la première voiture qui arrive peut passer.

— Si le pont contient une voiture qui va du nord au sud, seules les voitures circulant dans le même sens (donc arrivant au nord) peuvent passer.

— Inversement, si le pont contient une voiture qui va du sud au nord, seules les voitures arrivant au sud ont l'autorisation de passer.

Pour éviter tout problème, votre système dispose de barrières contrôlées par un ordinateur. Vous devez écrire le programme de contrôle qui tourne sur cet ordinateur en utilisant les outils classiques (mutex, variables de conditions, ...). À chaque fois qu'une voiture arrive à l'extrémité nord du pont, le système appelle automatiquement la fonction `EntreeNS()` puis lorsque cette voiture sort du pont, la fonction `SortieNS()` est appelée. Inversement, les fonctions `EntreeSN()` et `SortieSN()` servent à gérer les voitures qui circulent dans l'autre sens. Toutes ces fonctions peuvent être appelées en concurrence.

Nous supposerons que si la fonction `EntreeNS()` (ou `EntreeSN()`) se termine, le véhicule est autorisé à passer, alors que si elle bloque, le véhicule est aussi bloqué (par exemple, on peut imaginer un système qui détecte l'arrivée d'une voiture et appelle `EntreeNS()` quand la voiture arrive, lève la barrière d'entrée quand la fonction termine, et la redescend immédiatement après le passage de la voiture).

Q.II.1) - Donnez un algorithme des fonctions `EntreeNS()`, `EntreeSN()`, `SortieNS()` et `SortieSN()` en utilisant le principe du moniteur de Hoare.

Q.II.2) - Montrez la propriété de sûreté : il n'y a jamais à la fois une voiture venant du nord et une venant du sud sur le pont.

Q.II.3) - Montrez que cet algorithme n'a pas de problème de *deadlock* (inter-blocage).

Q.II.4) - L'algorithme pose-t-il un problème de famine ? Si oui, donnez un exemple puis proposez un nouvel algorithme.

III Pourquoi faire un programme multithread

On souhaite comparer l'efficacité d'un serveur de fichiers en mode mono ou multithread, même sur un ordinateur disposant d'un seul processeur monocoeur.

L'intérêt du multithread se trouve uniquement lors des accès disque car le thread qui demande l'accès à un fichier doit attendre pendant que les données sont lues sur le disque. Le serveur de fichiers dispose d'un cache en mémoire pour les fichiers les plus couramment lus. On suppose :

- la durée pour traiter une requête sans accès disque est de 15ms (récupérer la requête, chercher dans le cache, rendre le résultat) ;
- pour gérer le multithreading un surcoût de 5ms est nécessaire (changement de contexte et passage en mode noyau pour passer d'un thread à l'autre) ;
- si le fichier ne se trouve pas en cache il faut 75ms supplémentaires pour la lecture sur le disque ;
- en moyenne un fichier est disponible dans le cache dans 2/3 des cas.

Q.III.1) - Donner le nombre de requêtes traitées par seconde pour un serveur monothread

Q.III.2) - Donner le nombre de requêtes traitées par seconde pour un serveur multithread utilisant des threads noyaux.

Q.III.3) - Est-il intéressant d'utiliser des threads utilisateurs (green threads) ?

IV Algorithme de Dijkstra

Voici une tentative (incorrecte) d'algorithme d'exclusion mutuelle :

```
1   int locked = false;
2
3   void lock() {
4       while (locked) {
5           /* wait */
6       }
7       locked = true;
8   }
9
10  void unlock() {
11      locked = false;
12  }
```

Q.IV.1) - Montrer que cet algorithme ne garantit pas l'exclusion mutuelle.

Q.IV.2) - Si on vous fournit une fonction `bool test_and_set(bool* ptr)` qui permet atomiquement de lire la valeur à l'adresse ptr, d'y écrire la valeur `true` si `*ptr == false` et de renvoyer l'ancienne valeur de `*ptr`, pouvez-vous faire mieux (en pratique les processeurs modernes fournissent en général une instruction permettant de faire ceci) ?

L'article donnant le premier algorithme de résolution du mutex à n processus est donné page 4. Il date de 1965. Il suppose n processus sur N processeurs, et s'exécute en **mémoire partagée** : Il suppose que K est accessible en lecture par tous et peut être mis à jour par tous.

Q.IV.3) - Que sont I, K ?

Q.IV.4) - Comment sont initialisés les tableaux B et C ?

Q.IV.5) - Montrer que deux processus ne peuvent entrer en section critique en même temps.

Q.IV.6) - Montrez qu'un processus peut entrer en section critique lorsque c'est possible, c-à-d lorsqu'elle est libre.

Q.IV.7) - Que vient-on de montrer avec ces 2 propriétés ?

Q.IV.8) - Donner un inconvénient à l'algorithme.

Q.IV.9) - Pour ceux voulant aller plus loin, vous pouvez lire `http://jakob.engbloms.se/archives/65`. Le titre de l'article, « Dekker's Algorithm Does not Work, as Expected », devrait vous mettre la puce à l'oreil (sachant que l'algorithme de Dijkstra est une généralisation de Dekker) ;-).

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

The Problem

To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

The solution must satisfy the following requirements.

(a) The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"-"After you"-blocking is still possible, although improbable, are not to be regarded as valid solutions.

We beg the challenged reader to stop here for a while and have a try himself, for this seems the only way to get a feeling for the tricky consequences of the fact that each computer can only request one one-way message at a time. And only this will make the reader realize to what extent this problem is far from trivial.

The Solution

The common store consists of:

"**Boolean array** b, $c[1{:}N]$; **integer** k"

The integer k will satisfy $1 \leq k \leq N$, $b[i]$ and $c[i]$ will only be set by the ith computer; they will be inspected by the others. It is assumed that all computers are started well outside their critical sections with all Boolean arrays mentioned set to **true**; the starting value of k is immaterial.

The program for the ith computer ($1 \leq i \leq N$) is:

```
"integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
     go to Li1
     end
       else
Li4: begin c[i] := false;
       for j := 1 step 1 until N do
         if j ≠ i and not c[j] then go to Li1
     end;
     critical section;
     c[i] := true; b[i] := true;
     remainder of the cycle in which stopping is allowed;
     go to Li0"
```

The Proof

We start by observing that the solution is safe in the sense that no two computers can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the compound statement $Li4$ without jumping back to $Li1$, i.e., finding all other c's **true** after having set its own c to **false**.

The second part of the proof must show that no infinite "After you"-"After you"-blocking can occur; i.e., when none of the computers is in its critical section, of the computers looping (i.e., jumping back to $Li1$) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the kth computer is not among the looping ones, $b[k]$ will be **true** and the looping ones will all find $k \neq i$. As a result one or more of them will find in $Li3$ the Boolean $b[k]$ **true** and therefore one or more will decide to assign "$k := i$". After the first assignment "$k := i$", $b[k]$ becomes **false** and no new computers can decide again to assign a new value to k. When all decided assignments to k have been performed, k will point to one of the looping computers and will not change its value for the time being, i.e., until $b[k]$ becomes **true**, viz., until the kth computer has completed its critical section. As soon as the value of k does not change any more, the kth computer will wait (via the compound statement $Li4$) until all other c's are **true**, but this situation will certainly arise, if not already present, because all other looping ones are forced to set their c **true**, as they will find $k \neq i$. And this, the author believes, completes the proof.