TD 1 - ASR7 Programmation Concurrente
—
Concurrence, mutexes logiciels et matériels

Matthieu Moy, Fabien Rico, Adil Khalfa

Printemps 2018

I Gestion de comptes en banque, et concurrence

Deux agences d'une banque veulent mettre à jour le même compte bancaire. Pour cela, l'agence de Nancy effectue :

```
1. courant = get_account(1867A)
2. nouveau = courant + 1000
3. update_account(1867A, nouveau)
```

et l'agence de Karlsruhe :

```
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 1000
3. update_account(1867A, neue)
```

Q.I.1) - En supposant que l'agence de Nancy commence en premier, quel sera le montant à l'issue des transactions ?

II Algorithme de Dijkstra

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

```
int locked = false;

void lock() {
    while (locked) {
        /* wait */
    }
    locked = true;
}

void unlock() {
    locked = false;
}
```

Q.II.1) - Montrer que cet algorithme ne garantie pas l'exclusion mutuelle.

Q.II.2) - Si on vous fournit une fonction `test_and_set()` qui permet atomiquement de lire la valeur d'une variable et d'y écrire une autre valeur, 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.II.3) - Que sont I, K ?

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

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

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

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

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

Q.II.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) ;-).

III Producteur consomateur

Le problème du producteur consommateur est un problème classique de synchronisation en programmation multi-thread. Par exemple, le problème du producteur/consommateur présente un ensemble de threads « producteurs » qui dialogue avec un ensemble de threads « consomateurs » qui dialoguent grâce à une file de données partagées. On peut par exemple envisager un thread « Maître » qui reçoit les connexions des clients et qui fournit les sockets de discussions à des threads « Esclaves » qui traitent leurs demandes.

Pour éviter les problèmes d'accès concurrents à la liste de sockets, il faut protéger cette donnée. Le but de l'exercice est de programmer la liste sous la forme d'un moniteur.

Pour les questions suivantes, dans un premier temps, vous ne donnerez que les algorithmes.

Q.III.1) - Quelles fonctions doit implémenter le moniteur ? Quelles sont celles qui doivent être protégées ?

Q.III.2) - Donnez la description de la structure de données qui permet de stocker cette file.

Q.III.3) - Donnez l'algorithme des fonctions qui permettent d'assurer l'ajout et le retrait d'une tâche (la tâche sera un simple `int`). Dans un premier temps on pourra renvoyer une erreur quand on tente d'insérer dans une file pleine ou de retirer un élément d'une file vide.

Q.III.4) - Modifiez ces fonctions de manière à assurer l'attente (passive) en cas de file pleine ou vide.

Q.III.5) - Donner l'implémentation de ces fonctions avec la bibliothèque `thread C++`. N'oubliez pas les fonctions d'initialisation et de libération de ressources.

IV Gestion d'ordre avec des sémaphores

On considère un système où s'exécutent trois processus (légers ou lourds) P1, P2 et P3 qui ont les caractéristiques suivantes :

- P1 exécute en boucle les tâches A puis B ;
- P2 exécute en boucle les tâches U puis V ;
- P3 exécute en boucle la tâche X.

De plus, les contraintes suivantes doivent être respectées :

- La tâche A de P1 produit un élément nécessaire à la tâche X de P3. Cela signifie qu'une occurrence de X ne peut pas démarrer avant la fin d'une occurrence de A.
- Les tâches B et U ne peuvent s'exécuter en même temps.

On note dA_i et fA_i respectivement le début et la fin de la tâche A. On fait de même pour toutes les tâches. Répondez aux questions suivantes :

Q.IV.1) - Les ordres d'exécutions suivant sont-ils possibles ? Si non, quelles parties posent problème :

1(a) - $dA_1fA_1dX_1dB_1dU_1fX_1fU_1fB_1dA_2dX_2dV_1fV_1fX_2fA_2dU_2dB_2fU_2fB_2dA_3fA_3dB_3fB_3$

1(b) - $dA_1fA_1dX_1dB_1fB_1dA_2dU_1fA_2fX_1fU_1dX_2dV_1fV_1fX_2dU_2fU_2dB_2fB_2dA_3fA_3dB_3fB_3$

Q.IV.2) - Donnez le graphe de précédence et d'exclusion mutuelle.

Q.IV.3) - Gérez le problème entre P1 et P2 avec des sémaphores.

Q.IV.4) - Gérez le problème entre P1 et P3 avec des sémaphores.

Q.IV.5) - Peut-on utiliser la ou les mêmes sémaphores pour les questions IV.3 et IV.4 ?

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 $1 \le k \le N$, $b[i]$ 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 ($1 \le i \le N$) 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 $k \ne i$. 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 $k \ne i$. And this, the author believes, completes the proof.

Volume 8 / Number 9 / September, 1965 Communications of the ACM 569

Figure 1 – Article Dijkstra