TD 1 - LIF12 système d'exploitation

Passage d'informations sur les pipes

23 mars 2017

I Algorithme de Dijkstra

L'article donnant le premier algorithme de résolution du mutex est donné page 8. Il date de 1965 et a été conçu avant les sémaphores. 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.I.1) - Que sont I, K ?

Q.I.2) - Comment sont initialisés les tableaux B et C ?

Q.I.3) - Montrer que deux processus ne peuvent entrer en section critique en même temps.

Q.I.4) - Montrez qu'un processus peut entrer en section critique lorsque c'est possible, c-à-d lorsqu'elle est libre.

Q.I.5) - Que vient-on de montrer avec ces 2 propriétés ?

Q.I.6) - Donner un inconvénient à l'algorithme.

- I est le numéro du proc exécutant l'algorithme. K est le numéro du proc voulant entre en Section Critique.
- B et C doivent être dans le même état après la section critique qu'avant, donc on peut déduire qu'ils sont init à true au départ.
- Remarque préalable : On peut noter que 2 proc peuvent être rendus en Lib4 car l'un peut avoir exécuté la séquence d'instruction où il affecte I à K puis le test $(k! = i)$ alors qu'un autre a effectué le test $(B[K])$ et n'exécute sa prochaine instruction, c-à-d $K = I$ qu'après que le premier soit rendu à Lib4.
  Il faut donc départager les processus qui ont demandé et sont passés à la ligne Lib4 ensemble. On voit qu'ils affectent C[I] à false. Dans la boucle suivante, on attend que tous les C[J], dans l'ordre donné par la boucle, soient un à un à true. C'est cette boucle (l'ordre induit !) qui permet d'assurer l'exclusion mutuelle.
- On procède par un raisonnement par l'absurde :
  Supposons qu'il y ait 2 proc i et j en SC, alors lors du passage dans la boucle on a :
  - pour i, tous les C[j] valent true pour $J < i$
  - pour j, tous les C[J] valent true pour $J < j$

  Pourtant si i arrive à Lib4, il commence par affecter falser à C[i], et c'est la même chose pour j qui affecte false à C[j]. Comme on a $i < j$ ou $j < i$, on obtient une incohérence puisque C[i] et C[j] valent false.
  Donc 2 procs ne peuvent être en SC au même instant.
- Si personne ne demande à entrer en SC, alors tous les C[J] sont à true losqu'un processus fait sa demande. Le processus l'obtient donc sans attendre.
- La vivacité et la sûreté, donc que l'algo permet d'assurer une exclusion mutuelle.
- L'attente active ! C'est la raison des sémaphores et autres dispositifs hardware.

II Gestion d'ordre avec des sémaphores

On considère un système où s'exécutent trois processus (légers ou lourd) 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 sont en exclusion mutuelle.

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.II.1) - Les ordres d'exécutions suivant sont-ils possibles sinon, quelles parties posent problème :

1(a) - $dA_1 fA_1 dX_1 dB_1 dU_1 fX_1 fU_1 fB_1 dA_2 dX_2 dV_1 fV_1 fX_2 fA_2 dU_2 dB_2 fU_2 fB_2 dA_3 fA_3 dB_3 fB_3$

Non,

- $\ldots dA_2 dX_2 dV_1 fV_1 fX_2 fA_2 \ldots$, X_2 commence avant la fin de A_2 ;
- $\ldots dU_2 dB_2 fU_2 fB_2 \ldots$ les tâches U et B se recouvrent.

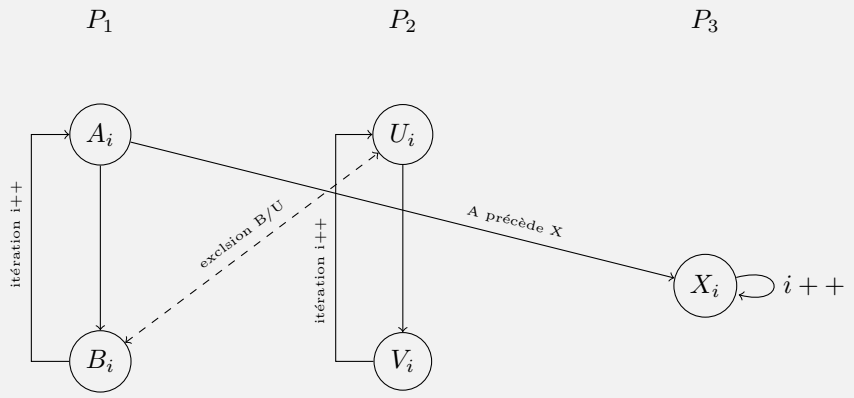
1(b) - $dA_1 fA_1 dX_1 dB_1 fB_1 dA_2 dU_1 fA_2 fX_1 fU_1 dX_2 dV_1 fV_1 fX_2 dU_2 fU_2 dB_2 fB_2 dA_3 fA_3 dB_3 fB_3$

oui, on vérifie chaque règle une a une en supprimant les de la liste les tâches qui ne sont pas concernées

- précédence dans P1 (A puis B en boucle) : $dA_1 dX_1 dB_1 fdB_1 dA_2 fA_2 dB_2 fB_2 dA_3 fA_3 dB_3 fB_3$ ... bon ;
- précédence dans P2 (U puis V en boucle) : $dU_1 fU_1 dV_1 fV_1 dU_2 fU_2 \ldots$ bon ;
- précédence dans P (X en boucle) : $dX_1 fX_1 dX_2 fX_2 \ldots$ bon ;
- précédence A puis X : $dA_1 fA_1 dX_1 dA_2 fA_2 fX_1 dX_2 fX_2 dA_3 fA_3$ ... bon ;
- exclusion mutuelle B et U : $dB_1 fdB_1 dU_1 fU_1 dU_2 fU_2 dB_2 fB_2 dB_3 fB_3$ ... bon.

Aucun problème détecté.

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



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

Un sémaphode S_{BU} initialisé à 1 (un mutex), les tâches B et U doivent *commencer* par $P(S_{BU})$ (P= tester= retirer 1 jeton) et *terminer* par $V(S_{BU})$ (V = ajouter 1 jeton).

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

Un sémaphore $S_{A->X}$ initialisé à 0, la tâche A doit *terminer* par $V(S_{A->x})$ (c'est à dire ajouter un jeton au stock) la tâche X doit *commencer* par $P(S_{A->X})$ (c'est à dire prendre un jeton).

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

Non, en tout cas pas avec cette algo. supposons que ce soit le cas. Quel serait la valeur initials du sémaphore ? Initialisé ≥ 1, rien n'empêche X de démarrer immédiatement ce qui est interdit. Initialisé à 0, U, B et X B seraient bloquées dès le départ. Seul A peut s'éxécuter. Après sa fin, X a la possibilité de s'éxécuter et dans ce cas, la tâche bloque la section critique de B et U. Comme elle ne peut pas s'écécuter 2 fois, elle ne doit pas réouvrir le passage (pas de V à la fin) et seul A peut le faire. Mais A se déroule après B qui est bloqué $\Rightarrow$ deadlock.

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 envoie 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 ?

1 Initialisation
2 Libération mémoire
3 Ajout d'un élément
4 Retrait d'un élément
5 (opt) Lecture de la taille de la file
6 (opt) Est-elle pleine
7 (opt) Est-elle vide
8 (opt) Essai du retrait (faire un retrait mais sans bloquer si la liste est vide)
9 (opt) Essai d'ajout (essayer un ajout sans bloquer si cela est impossible)

1 et 2 ne doivent pas être protégés car ils ne doivent pas être fait par plusieurs threads. 5,6 et 7 ne doivent pas être protégés car même si l'information est fausse cela ne pose pas de problème (de toute façon l'information qu'ils fournissent ne peut pas rester valide indéfiniment) 3,4 8 et 9 doivent être exécutés de manière unitaire car sinon, il pourrait y avoir des problèmes de perte de données ou de remplissage...

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).

```
début
    Mutex M ;
    Element Table[TAILLE] ;
    Entier Debut = 0 ;
    Entier NbElem = 0 ;
fin
```

Algorithme 1 : Structure de données

La table permet de stocker les données, les deux entiers servent à savoir où lire une donnée (Debut) et où enregistrer une nouvelle donnée (Debut+NbElem mod TAILLE). M servira a assurer la protection.

Pour l'ajout :

```
Données : Element e
Résultat :
début
    Lock(M)
    si NbElem == TAILLE alors
     !!!!La liste est pleine!!!
    Table[(Debut+NbElem) modulo TAILLE] = e ;
    NbElem ← NbElem+1 ;
    Unlock(M)
fin
```

Algorithme 2 : Ajout d'un élément

Pour le retrait :

```
Données :
Résultat : Element e
début
    Lock(M)
    si NbElem == 0 alors
     !!!!La liste est vide!!!
    res ← Table[Debut] ;
    NbElem ← NbElem-1 ;
    Debut ← Debut+1 ;
    Unlock(M)
    retourner res
fin
```

Algorithme 3 : Retrait d'un élément

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

```
début
    Mutex M ;
    Cond C_vide ;
    Cond C_plein ;
    Element Table[TAILLE] ;
    Entier Debut = 0 ;
    Entier NbElem = 0 ;
fin
```

Algorithme 4 : Structure de données

C_vide sert à assurer l'attente passive lorsque la pile est vide, C_plein idem pour le cas plein.

```
Données : Element e
Résultat :
début
    Lock(M)
    tant que NbElem == TAILLE faire
        attend(C_plein,M) ;
    Table[(Debut+NbElem) modulo TAILLE] = e ;
    NbElem ← NbElem+1 ;
    signal(C_vide) ;
    Unlock(M)
fin
```

Algorithme 5 : Ajout d'un élément

attend(C_*,M) signifie qu'on libère M et qu'on attend qu'un autre fasse le signal correspondant. Le fait d'utiliser tantque dans ce cas n'est pas très utile, mais au pire c'est comme un if, sinon, cela évite souvent d'oublier un cas.

```
Données :
Résultat : Element e
début
    Lock(M)
    tant que NbElem == 0 faire
        attend(C_vide, M) ;
    res ← Table[Debut] ;
    NbElem ← NbElem-1 ;
    Debut ← Debut+1 ;
    signal(C_plein) ;
    Unlock(M)
    retourner res
fin
```

Algorithme 6 : Retrait d'un élément

Q.III.5) - Donner l'implémentation de ces fonctions avec la librairie pthread. N'oubliez pas les fonctions d'initialisation et de libération de ressources.

A faire en TP

IV Ordonancement

Nous allons utiliser l'implantation POSIX 1003b sur Linux. Il existe 100 niveaux de priorité :

- L'ordonnancement est préemptif.
- Le niveau 0 est réservé à `SCHED_OTHER`, et les niveaux de priorité 1 à 99 aux politiques `SCHED_FIFO` et `SCHED_RR`.
- Les tâches de priorité 99 sont les tâches de plus forte priorité.

— Les tâches `SCHED_FIFO` ne sont interrompues que par des tâches de plus forte priorité.
— Les tâches `SCHED_RR` ne sont interrompues que par des tâches de plus forte priorité et elle partagent le processeur avec les tâches de priorité égale selon un **quantum de 2 unités** de temps.
— Lorsqu'une nouvelle tâche arrive elle est placée en tête de la liste des tâches de priorité identique.
— `SCHED_OTHER` est dédié à l'ordonnanceur temps partagé.
— Pour simplifier, nous supposerons que la politique `SCHED_OTHER` choisit systématiquement la tâche qui a passé le moins de temps dans le processeur (la décision se fait **avec le quantum de 1**).
— Une tâche interrompue avant la fin de son quantum obtient un quantum complet la prochaine fois qu'elle est ordonnancée.

Soit le jeu de tâches apériodiques suivant :

Tâche	Date d'arrivée	Priorité	Durée	Politique
o1	0	0	10	SCHED_OTHER
fifo1	0	5	4	SCHED_FIFO
fifo2	1	5	2	SCHED_FIFO
rr1	3	6	4	SCHED_RR
rr2	4	6	5	SCHED_RR
rr3	5	10	5	SCHED_RR

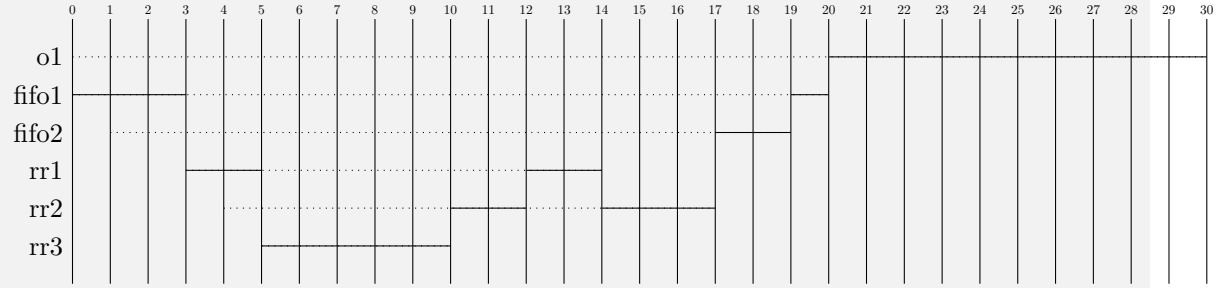
Q.IV.1) - On suppose qu'une fois arrivées, les tâches sont toujours prêtes. Dessinez de l'instant 0 à l'instant 30, l'ordonnancement généré par l'ordonnanceur.

Q.IV.2) - Donnez le temps de réponse de chaque tâche.

Q.IV.3) - On complique l'ordonnancement en ajoutant une règle qui est présente par sécurité pour empêcher les tâches temps réelles de monopoliser le CPU (et donc de bloquer l'ordinateur). Une proportion fixe des unités de temps est réservée aux tâches en temps partagé : $\frac{1}{10}$. Pour cela :
— **Toutes les 10 unités de temps**, en commençant par le temps 0, vous devez réserver un temps pour une tâche `SCHED_OTHER`.
— Vous devez la placer **au plus tôt** sans tenir compte de la priorité, dès qu'une tâche `SCHED_OTHER` est disponible, mais sans interrompre une tâche `SCHED_FIFO` ou `SCHED_RR` déjà commencée.
— Vous devez donc attendre la première interruption naturelle (fin de tâche, fin de quantum d'une tâche `SCHED_RR`, apparition d'une tâche plus prioritaire).
— S'il est impossible de placer ce temps réservé dans l'intervalle de 10 unités de temps, ce dernier est annulé.

Refaites l'ordonnancement et recalculez le temps de réaction des tâches en suivant cette règle.

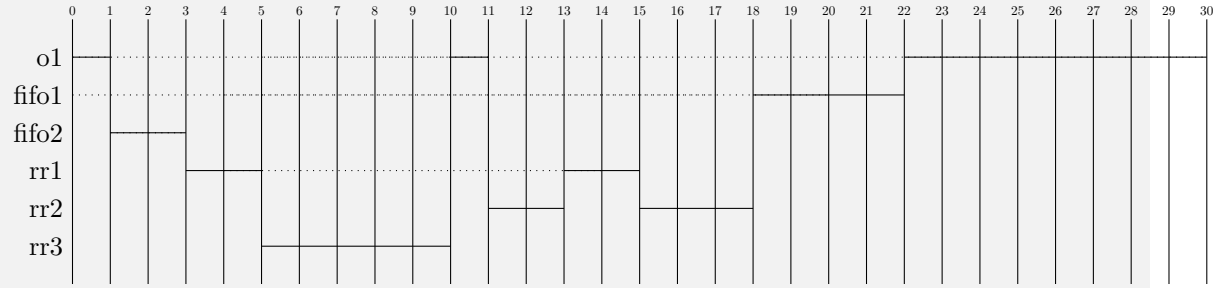
Question IV.1 :



Temps de réponse :

o1 : 30-0 = 30	
fifo1 : 20-0 = 20	fifo2 : 19-1 = 18
rr1 : 14-3 = 11	rr2 : 17-4 = 13
rr3 : 10-5 = 5	

Question IV.3 :



Temps de réponse :

o1 : 30-0 = 30	
fifo1 : 22-0 = 22	fifo2 : 3-1 = 2
rr1 : 15-3 = 12	rr2 : 18-4 = 14
rr3 : 10-5 = 5	

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.

FIGURE 1 – Article Dijkstra