TD 1 - LIF12 système d'exploitation

Passage d'informations sur les pipes

22 mars 2017

I Algorithme de Dijkstra

L'article donnant le premier algorithme de résolution du mutex est donné page **??**. 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.

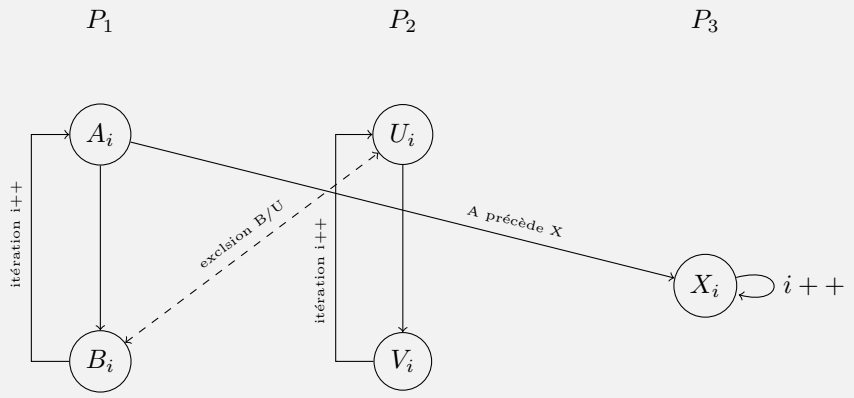
1(b) - $dA_1 fA_1 dX_1 dB_1 fB_1 dA_2 dU_1 fA_2 fX_1 fU_1 dX_2 dV_1 fV_1 fX_2 dU_2 fU_2 dB_2 fB_2 dA_3 fA_3 dB_3 fB_3$

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