TD 1 - LIFPCA Programmation Concurrente et Administration Système

—

Concurrence, mutexes logiciels et matériels

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

Printemps 2024

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 :

```
A. aktuelles = get_account(1867A)
B. neue = aktuelles - 1000
C. 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 ?

Ça n'a aucun rapport réel avec qui commence la transaction, puisqu'elles ne sont pas atomiques (la partie « En supposant ... en premier » de la question est volontairement piégeuse). Des variables locales + exécutions parallèles entremêlées donnent des résultats différents, par exemple :

- 1 ; 2 ; 3 ; A ; B ; C
  ⇒ compte inchangé
- 1 ; A ; 2 ; B ; 3 ; C
  ⇒ compte – 1000
- 1 ; A ; B ; 2 ; C ; 3
  ⇒ compte + 1000

On a une condition de compétition (*race condition*).
→ Comment garantir le même montant ?
⇒ en forçant l'ordre d'exécution.
→ Comment forcer l'ordre d'exécution ?
⇒ en rendant les opérations atomiques.
Cette opération est une **section critique**. Elle doit s'exécuter en **exclusion mutuelle**.

Attention ! Pour résoudre un problème de mutex, il faut trouver une solution vérifiant les propriétés de **sûreté** (il n'y a qu'un processus en section critique à un instant donné) et de vivacité (un processus souhaitant entrer en section critique le pourra dans un temps fini). Ces propriétés garantissent qu'il n'y a pas de problèmes d'interblocage ou de famine.

II Producteur consommateur

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 « consommateurs » 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. Vous avez déjà manipulé une forme de producteur-consommateur entre processus : le « pipe » unix (`cmd1 | cmd2`, où `cmd1` est producteur, et `cmd2` consommateur).

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.

Bien rappeler le principe du moniteur de Hoare (on dira parfois juste « moniteur », dans ce contexte c'est sous-entendu « moniteur de Hoare »).

Rappeler aussi le principe du producteur consommateur : une FIFO dans laquelle des threads peuvent envoyer des données, et d'autres les lire (avec blocage quand nécessaire).

Le producteur-consommateur est un schema de synchronisation et communication. Ici on l'implémente avec un moniteur, mais on aurait pu faire autrement (par exemple, c'est au delà des objectifs du cours mais on aurait pu avoir une structure de données lock-free).

Inversement, le principe du moniteur de Hoare peut être appliqué ailleurs qu'à un producteur-consommateur, ce n'est qu'un exemple.

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

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

Q.II.3) - Donnez l'algorithme des fonctions qui permettent d'assurer l'ajout et le retrait d'un élément (l'élément 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.

On écrit du pseudo-code, parce que si on écrivait du C++ directement on donnerait la solution du TP et les étudiants pourraient recopier sans comprendre.

```
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

Les Lock/Unlock sont nécessaires :

- Sans réfléchir, parce qu'on a dit qu'on faisait un moniteur, donc on verrouille en entrée et on déverrouille en sortie (sauf exceptions, cf. question précédente).
- En réfléchissant, on fait NbElem ← NbElem+1 qui est l'exemple bateau de code qui ne marche pas sans exclusion mutuelle.

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) modulo TAILLE ;
    Unlock(M)
    retourner res
fin
```

Algorithme 3 : Retrait d'un élément

Pour l'instant on ne dit pas vraiment ce qu'il se passe en cas de liste vide ou pleine, c'est l'objet de l'exercice suivant.

Q.II.4) - Modifiez ces fonctions de manière à assurer l'attente (passive) 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 en théorie, mais au pire c'est comme un « if », sinon, cela évite souvent d'oublier un cas. En pratique, la plupart des implémentations autorisent les « spurious wakeups » et imposent de re-tester la condition au retour de l'attente (donc de faire un « tantque » et pas un simple « if »).

```
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) modulo TAILLE ;
  signal(C_plein) ;
  Unlock(M)
  retourner res
fin
```

Algorithme 6 : Retrait d'un élément

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

A faire en TP

III 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.

Amener les étudiants à dire qu'elles 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.III.1) - Les ordres d'exécutions suivant sont-ils possibles ? Si non, quelles parties posent problème :

1(a) - $dA_1 f A_1 dX_1 dB_1 dU_1 f X_1 f U_1 f B_1 dA_2 dX_2 dV_1 f V_1 f X_2 f A_2 dU_2 dB_2 f U_2 f B_2 dA_3 f A_3 dB_3 f B_3$

Non,

— $\ldots dA_2 dX_2 dV_1 f V_1 f X_2 f A_2 \ldots$, X_2 commence avant la fin de A_2 ;

— $\ldots dU_2 dB_2 f U_2 f B_2 \ldots$ les tâches U et B se recouvrent, contradictoire avec l'exclusion mutuelle ;

— $\ldots dB_1 dU_1 \ldots f U_1 f B_1 \ldots$ la tâche U s'exécute à l'intérieur de l'intervalle d'exécution de la tâche B, également contradictoire avec l'exclusion mutuelle ;

Pour mieux visualiser la situation, on peut tracer un chronogramme.

1(b) - $dA_1 f A_1 dX_1 dB_1 f B_1 dA_2 dU_1 f A_2 f X_1 f U_1 dX_2 dV_1 f V_1 f X_2 dU_2 f U_2 dB_2 f B_2 dA_3 f A_3 dB_3 f B_3$

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

— précédence dans P1 (A puis B en boucle) : $dA_1 dX_1 dB_1 f dB_1 dA_2 f A_2 dB_2 f B_2 dA_3 f A_3 dB_3 f B_3 \ldots$ bon ;

— précédence dans P2 (U puis V en boucle) : $dU_1 f U_1 dV_1 f V_1 dU_2 f U_2 \ldots$ bon ;

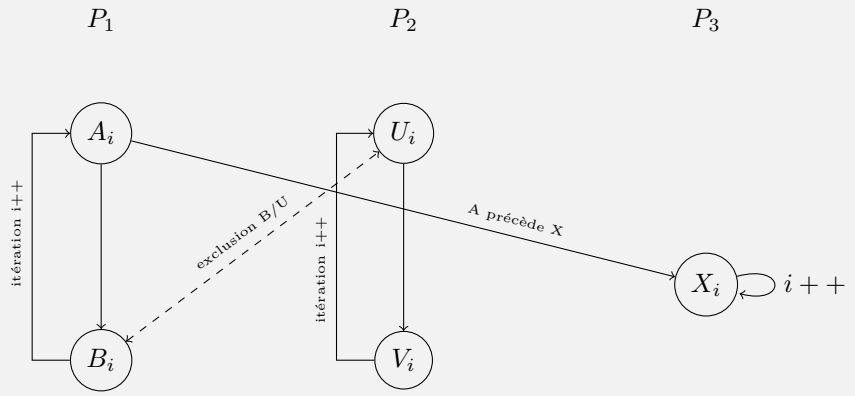
— précédence dans P3 (X en boucle) : $dX_1 f X_1 dX_2 f X_2 \ldots$ bon ;

— précédence A puis X : $dA_1 f A_1 dX_1 dA_2 f A_2 f X_1 dX_2 f X_2 dA_3 f A_3 \ldots$ bon ;

— exclusion mutuelle B et U : $dB_1 f dB_1 dU_1 f U_1 dU_2 f U_2 dB_2 f B_2 dB_3 f B_3 \ldots$ bon.

Aucun problème détecté.

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



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

Non, en tout cas pas avec cet algo. Supposons que ce soit le cas. Quelle serait la valeur initiale 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.