Corrigé de l'examen de systèmes opératoires
Module Synchronisation des processus

2ième Année Informatique et Mathématiques Appliquées

17 Novembre 1999

1 Les sémaphores

Questions-Réponses

1. Donner la définition d'un sémaphore privé. Pourquoi de tels sémaphores sont plus simples à implanter ?

   Réponse Un sémaphore privé est un sémaphore pour lequel un unique processus, que l'on peut considérer comme propriétaire du sémaphore, exécute la primitive P. Il ne peut donc exister qu'un seul processus au plus bloqué sur ce sémaphore lors d'une opération P. Il n'est donc pas nécessaire de gérer une file d'attente des processus bloqués, ce qui simplifie l'implantation de l'objet sémaphore.
2. Par quel mécanisme simple peut-on assurer l'exclusion mutuelle entre les opérations P et V sur un système monoprocesseur ?

   Réponse Puisqu'il n'existe qu'un seul processeur, une primitive P ou V s'exécutera bien en exclusion mutuelle si elle n'est pas interrompue en cours d'exécution. Par conséquent, il suffit d'assurer l'exécution de telles primitives en mode ininterruptible. Les primitives seront parenthésées par, en entrée, passer en mode ininterruptible en sauvegardant le mode courant d'exécution de l'appelant et, en sortie, restaurer le mode d'exécution de l'appelant.
3. Une section critique utilisée par plusieurs processus est protégée de façon classique par un sémaphore d'exclusion mutuelle

   ```
   P(Mutex)
   /* section critique */
   V(Mutex)
   ```

   Un des processus provoque un déroutement durant la section critique et s'arrête. Quelle anomalie provoque-t-il ainsi sur la suite de l'exécution des autres processus ?

   Réponse Le processus dérouté n'ayant pas exécuté la sortie de la section critique par l'appel $V(Mutex)$, la section critique est considérée comme toujours occupée par un processus et par conséquent plus aucun autre processus ne pourra entrer en section critique.

2 Les moniteurs de Hoare

On considère un système composé de processus accédant à des ressources critiques de différentes classes. On suppose qu'il existe P classes de ressources critiques et qu'un tableau $Libre[0..P-1]$ précise initialement le nombre de ressources disponibles de chaque classe.

Le contrôle de l'allocation de ces ressources aux processus est réalisé à l'aide du moniteur de Hoare suivant :

```
monitor  Allocateur {

   int Libre[P] ;      /* Compteurs de ressources par classe */
   condition Assez ;   /* Attente de ressources suffisantes  */

   /* fonction interne contrôlant si l'allocation est possible */
   boolean insuffisant(int dem[P]) {
      int i = 0;
      while (i < P) { if (Libre[i] < dem[i]) return true ; i++ ; } ;
      return false ;
      }

void Allouer (int Dem[P]) {
   while (insuffisant(Dem)) { wait(Assez) ; }
   for (i=0 ; i < P ; i++) Libre[i] = Libre[i] - Dem[i] ;
   signal(Assez) ;
   }

void Libérer (int Res[P]) {
   for (i=0 ; i < P ; i++) Libre[i] = Libre[i] + Res[i] ;
   signal(Assez)
   }

/* bloc d'initialisation */
for (i=0 ; i < P ; i++) Libre[i] = V ;
}
```

Hypothèses On suppose que les processus respectent les règles de comportement suivantes :
- ils ne demandent pas plus de ressources d'une classe donnée qu'il n'en existe dans cette classe ;
- après avoir obtenu des ressources par une opération *Allouer*, ils libèrent TOUTES ces ressources par l'appel à l'opération *Libérer* adéquate avant de redemander éventuellement des ressources.

Questions-Réponses

4. Justifiez la présence de l'appel de l'opération *signal* dans la procédure *Allouer* ;

   Réponse L'appel de l'opération *signal* permet un réveil "en chaîne" des processus bloqués. En effet, compte tenu de l'approche adoptée, on ne sait pas quel processus doit être réveillé ou combien de processus peuvent être servis. Il faut donc les réveiller tous.

5. La solution proposée risque-t-elle de conduire à une situation d'interblocage ? (Justifiez votre réponse)

   Réponse Le système ne risque pas l'interblocage pour deux raisons :
   - la première tient au comportement des processus : ils ne demandent jamais de ressources s'ils en ont déjà ;
   - la deuxième tient à la stratégie d'allocation de la solution, en l'occurrence, le tout ou rien.

   Ces propriétés garantissent que la condition suivante, nécessaire au risque d'interblocage, est toujours fausse : un processus possède des ressources critiques et les garde alors qu'une nouvelle demande ne peut être satisfaite.

6. La solution proposée risque-t-elle de conduire à une situation de famine ? (Justifiez votre réponse)

Réponse La solution comporte un risque de famine puisque les processus réveillés entrent en compétition pour acquérir leurs ressources dans un ordre totalement arbitraire.

On associe maintenant une variable condition à chaque classe de ressources. On a donc un tableau $Assez[P]$. Les procédures *Allouer* et *Libérer* deviennent :

```
void Allouer (int Dem[P]) {
   int i = 0 ;
   while (i < P ) {
      if (Dem[i] > 0 ) {
         while (Dem[i] > Libre[i]) wait(Assez[i]) ;
         Libre[i] = Libre[i] - Dem[i] ;
         signal(Assez[i]) ;
         } ;
      i++ ;
      }
   }

void Libérer (int Res[P]) {
   for (i=0 ; i < P ; i++) { Libre[i] = Libre[i] + Res[i] ; signal(Assez[i]) ; }
   }
```

Questions-Réponses

7. Expliquez pourquoi cette solution ne présente pas de risque d'interblocage en précisant notamment quelle condition nécessaire à une situation d'interbocage reste fausse dans cette solution ;

   Réponse Dans cette solution, un processus va acquérir des ressources peu à peu dans un ordre prédéfini de la classe 0 à la classe $(P-1)$. La stratégie appliquée est donc celle des ressources ordonnées de Havender. Aucun cycle ne pourra se produire dans le graphe d'allocation et il n'y aura donc pas de risque d'interblocage.

8. Cette solution présente-t-elle un risque de famine ? (Justifiez votre réponse)

   Réponse Cette solution reste tout autant que la précédente sujette au risque de famine. En effet, un processus bloqué en attente de ressources de la classe i pourra infiniment souvent se faire "voler" les ressources libres dont il a besoin, par des processus en attente de la même classe de ressources, le réveil en chaîne donnant finalement priorité aux processus derniers arrivés pour tenter leur chance.

9. Les processus peuvent-ils avoir un comportement moins restrictif sans qu'il existe de risque d'interblocage : plus précisément, peuvent-ils acquérir des ressources nouvelles par une opération *Allouer* sans avoir forcément libéré les ressources préalablement obtenues ?

   Réponse Ce comportement est parfaitement acceptable sans risque d'interblocage dans la mesure où la condition maintenue fausse est l'occurrence d'un cycle dans le graphe d'allocation. Attention, ceci suppose que les processus demandent leurs ressources **dans l'ordre** 0 à $P-1$ (un processus qui possède des ressources de classe i ne demandera que des ressources de classes supérieure à i) et qu'ils demandent les ressources d'une classe en une **seule fois** (ils n'acquièrent pas des ressources d'une classe donnée petit à petit).

3 Tâches Ada et synchronisation par rendez-vous (2 points)

On considère la procédure *Test* suivante :

```
procedure Test() is
      task T is  entry Faire() ; end T;
      task body T is
        begin
           loop accept Faire() ; end loop ;
        end T ;
   begin
   end Test ;
```

Questions-Réponses

10. Expliquez pourquoi, si cette procédure est appelée, son exécution ne se terminera pas ;

    Réponse La tâche T ne se terminant pas, la procédure qui a activé cette tâche ne peut donc se terminer ;

11. Que faudrait-il modifier dans le corps de la tâche T pour que la terminaison de l'exécution de la procédure $Test$ soit possible ?

    Réponse Il suffit d'introduire une instruction *select* incluant la clause *terminate* en remplacement de l'acceptation inconditionnelle. Le corps de la tâche T devient donc :

```
task body T is
  begin
     loop
        select
           accept Faire() ;
        or
           terminate ;
        end select ;
     end loop ;
  end T ;
```

4 Transactions : Contrôle de concurrence (3 points)

On considère deux transactions $T1$ et $T2$ accédant à des variables partagées A, B :

```
transaction T1 {
  A = B + 1 ;
  A = A + 1 ;
}
```

```
transaction T2 {
   B = A * 2 ;
   B = B + 2 ;
}
```

Questions-Réponses

12. En supposant un état initial vérifiant $A = 1 \wedge B = 2$ précisez le ou les états qui peuvent être considérés comme corrects après l'exécution de $T1 \mid\mid T2$?

    Réponse Les seuls résultat acceptables sont ceux obtenus par l'exécution séquentielle $T1; T2$ ou $T2; T1$, soit donc :

$$(A = 4 \wedge B = 10) \vee (A = 6 \wedge B = 4)$$

13. Exhiber une exécution entrelacée de $T1 \mid\mid T2$ conduisant à un état erroné ;

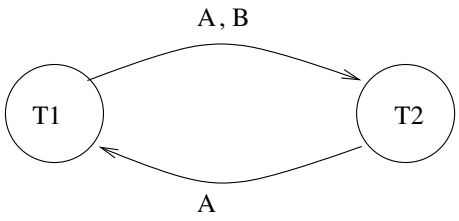

FIG. 1 – Graphe de sérialisation

Réponse Il suffit d'entrelacer la première instruction de $T2$ entre les deux instructions de $T1$, on obtient alors comme résultat :

$$(A = 4 \wedge B = 8)$$

La transaction $T2$ a lu une valeur intermédiaire de A.

14. Dessiner le graphe de sérialisation de $T1$ et $T2$ associé à l'exécution entrelacée décrite dans la question précédente.

Réponse L'ordre d'exécution choisi est donc :

$$\begin{array}{lll} T1 & : & A = B + 1 \\ T2 & : & B = A * 2 \\ T1 & : & A = A + 1 \\ T2 & : & B = B + 2 \end{array}$$

D'où le graphe de sérialisation :

Rappel : Un arc orienté de T_i à T_j étiqueté par X existe dans ce graphe ssi la transaction T_j lit une variable précédemmemt lue ou écrite par T_i) ;