Université Paris Diderot
M1 IAP
Théorie et pratique de la concurrence

Exam

Lundi 14 mai 2018–8h30 à 10h30

Aucun document n'est autorisé. Les exercices sont indépendants.

Quand on demande d'écrire du code, on peut écrire du "pseudo-code" suffisamment précis, il n'est pas nécessairement d'écrire du code java compilable.

Exercice 1.— On rappelle l'interface des sémaphores décrite dans le cours contenant les méthodes:

```
public void acquire();
public void release();
```

et le constructeur `Semaphore(int permits)`.

On rappelle l'interface des barrières décrite dans le cours;

```
public interface Barrier {
  /**
   * Bloque jusqu'à ce que tous les threads soient devant la barrière
   */
  public void await();
}
```

et un constructeur qui définit le nombre n de threads qui doivent être attendus avant de franchir la barrière. On rappelle qu'une thread reste en attente devant une barrière jusqu'à ce que toutes les n threads soient devant la barrière. Les barrières définies ici sont réutilisables: après un franchissement de la barrière, le franchissement suivant est soumis à la même condition de présence des n threads devant la barrière.

Sommes préfixes. Il s'agit de calculer à partir d'une séquence de nombre $a_0, .., a_{n-1}$ les n sommes (partielles) b_i $(0 \leq i \leq n-1)$

$$b_i = \Sigma_{j=0}^{i} a_j$$

On suppose dans la suite que n est une puissance de 2. Les valeurs des a_i sont dans un tableau d'entiers $a[]$, a_i étant la valeur de $a[i]$ initialement.

1. On considère le code séquentiel suivant:

   ```
   b[0]=a[0];
   for(int i=1; i<n; i++)
       b[i]=a[i]+b[i-1];
   ```

   Vérifier qu'à la fin de l'exécution de ce code, le tableau `b` contient bien les sommes partielles. Quel est l'ordre de grandeur de son temps d'exécution (on suppose que toutes les opérations de bases, lecture/écriture dans le tableau prennent un temps constant).

2. On va maintenant utiliser n threads pour le calcul. Pour cela, la i-ème thread calculera b_i de la manière suivante: la i-ème thread "attendra" que la $i-1$-ème thread ait terminé son calcul, elle le récupèrera, y ajoutera a_i pour calculer b_i et elle "préviendra" la thread suivante $i+1$ que le calcul est fait (la thread 0 commence sans attendre et la thread $n-1$ ne prévient aucune autre thread).

(a) En utilisant des sémaphores (on pourra en utiliser $n-1$), écrire le code (constructeur et méthode run) de ces threads (vous préciserez comment se fait l'accès aux éléments partagés).

(b) On suppose que les opérations de lecture/écriture d'un élément d'un tableau et l'addition prennent un temps constant; comme chaque thread exécute un nombre constant de telles opérations, on supposera que le temps de calcul de chaque thread est une constante C. On suppose aussi que si *sem* est un sémaphore et *sem.release*() est la dernière instruction d'une thread, le temps entre l'execution de ce *sem.release*() et l'activation d'une thread en attente sur *sem.acquire*() est négligeable. On suppose aussi qu'une thread exécute son code immédiatement (sauf si elle est en attente).

En supposant que le programme débute à l'instant t_0, à quel instant débute la i-ème thread? A quel instant le calcul est-il fini? En déduire le temps nécessaire pour calculer chacune des sommes préfixes?

3. Pour calculer ces sommes préfixes on peut aussi procéder en "rondes" de calcul. Chaque ronde r a ici une valeur qui est une puissance de 2. Vérifier que le code séquentiel suivant permet de calculer les sommes préfixes (le résultat étant, à la fin des calculs, dans le tableau a).

```
for(int r=1; r<a.length; r *=2)
    for(int j=a.length-1; j>=r; j--)
        a[j] += a[j-r];
```

(a) Combien de rondes (en fonction de la taille de a) sont exécutées par l'algorithme. Quel est l'ordre de grandeur du nombre d'opération faites à chaque ronde? En déduire quel est l'ordre de grandeur du temps que met l'algorithme à terminer?

(b) On veut maintenant calculer en "parallèle" chaque ronde. Pour cela pour chaque i entre 0 et $n-1$ on associe un thread qui va calculer la nouvelle valeur de $a[i]$ à la fin de chacune des rondes. Proposer un algorithme en utilisant des barrières de synchronisation pour faire ce calcul en parallèle.

(c) Sous les mêmes hypothèses que dans la question précédente et en supposant que si *Bar* est une barrière de synchronisation le temps entre l'arrivée de la dernière thread sur *Bar.await*() et l'activation de toutes les threads en attente sur *Bar.await*() est aussi une constante (on suppose qu'une thread qui ne se met pas elle-même en attente s'exécute), quel est le temps nécessaire pour calculer toutes les sommes préfixes?

Exercice 2.— On considère l'algorithme MP pour deux processus P_1 et P_2, décrit à la figure 1. Cet algorithme se base sur deux variables propres: `wantp1` pour P_1 et `wantp2` pour P_2.

```
int wantp1 := 0      // variables partagées
int wantp2 := 0
```

```
-- Processus P1
loop forever :
p1: section NC
p2: wantp1 := (wantp2== -1) ? -1 : 1
p3: await (wantp1 != wantp2)
p4: section critique
p5: wantp1 := 0
```

```
-- Processus P2
loop forever :
q1: section NC
q2: wantp2 := (wantp1== -1) ? 1 : -1
q3: await (wantp1 != wantp2)
q4: section critique
q5: wantp2 := 0
```

Figure 1: Algorithme MP

Comme en cours, on suppose que chaque instruction du code est atomique. En particulier les instructions p2 et q2. Ainsi pour l'instruction p2, on teste si la valeur de `wantp2` vaut -1, et selon le résultat du test, on affecte -1 ou 1 à `wantp1`, et tout cela se fait en un seul pas.

1. Dessiner une partie du diagramme d'états (une dizaine d'états suffit).

2. On s'intéresse à l'exécution de l'instruction p2: pour quelle valeur de `wantp2`, l'affectation p2 donne une valeur à `wantp1` qui permet, juste après (*ie* sans autre modification des variables du programme) à P_1 de franchir le `await` de l'instruction p3 et d'arriver en SC ?

   Même question pour les valeurs de `wantp1` qui permet à P_2 d'enchainer les instructions q2 et q3 et d'arriver en SC ?

3. Montrer l'invariant suivant: lorsque P_1 (resp. P_2) est en SC, alors `wantp1` (resp. `wantp2`) vaut 1 ou -1.

4. Montrer que l'algorithme MP assure l'exclusion mutuelle: il n'est pas possible d'avoir les deux processus en SC au même moment.
   Idée: on pourra montrer qu'il n'est pas possible pour P_1 (ou P_2) de rejoindre l'autre processus en section critique.

5. Montrer l'absence de famine: pour toute exécution équitable et en supposant que toute section critique termine, alors si P_1 (ou P_2) commence à exécuter son préprotocole, il arrivera en section critique.
   Idée: on pourra montrer qu'il n'est pas possible pour P_1 (ou P_2) de rester bloqué pour toujours dans le préprotocole.

6. On suppose à présent que les instructions p2 et q2 ne sont pas atomiques, c'est à dire que p2 a la forme suivante (et idem pour q2):

```
p2: if  (wantp2== -1) then
p2':          wantp1 :=  -1
p2'': else wantp1 := 1
```

   Montrer que l'exclusion mutuelle n'est plus garantie en donnant un scénario d'erreur.

7. Comment s'énonce en LT les propriétés suivantes pour l'algorithme MP:

   (a) L'exclusion mutuelle (absence de conflit en SC).

   (b) L'absence de famine.

8. Donner les codes Prism du processus P_1 lorsque l'instruction p2 est supposée atomique et lorsqu'elle est décomposée comme à la question 6.