TD5 : Synchronisation de threads avec des sémaphores

1 Précédence

Définition : graphe de précédence Dans un programme concurrent, l'activité de chaque thread est composée de phases dites de calcul (où on exécute des instructions ordinaires) et de synchronisations (où on invoque des méthodes de sémaphores). Une phase de calcul est supposée indépendante et purement séquentielle, c'est à dire qu'elle s'exécute sans interaction avec les autres threads. On dit qu'il y a une *contrainte de précédence* entre deux calculs C1 et C2 lorsque le calcul C1 *doit être terminé avant le début* du calcul C2.
On peut représenter visuellement ces contraintes par un graphe orienté :

— chaque calcul C est représenté par un sommet [C]

— une contrainte de précédence entre C1 et C2 est représentée par une arête [C1] → [C2]

Exercice 1 Soit le programme ci-dessous, composé de deux threads concurrents. Complétez ce programme pour qu'il respecte le graphe de précédence [C1] → [C2]. Il s'agit d'une part de décider la valeur initiale du sémaphore `S` (cadre n° 1) et d'autre part d'invoquer judicieusement `P(S)` et/ou `V(S)` au cours de l'exécution (cadres n° 2 et n° 3).

Remarque Dans ce TD, on vous donne des «programmes à trous» dans lesquels vous devez ajouter des synchros. Les points d'interrogation indiquent l'emplacement des trous. Par exemple pour le thread A ci-dessous, vous pouvez décider de rajouter un ou plusieurs appels à P() et/ou V() avant et/ou après le calcul C1. Même chose dans B, avant et/ou après C2. Attention, il n'est pas obligatoire de remplacer *chaque* point d'interrogation par quelque chose.

Conditions initiales

```
1 semaphore S = ?
```

Thread A

```
1 ?
2 C1;
3 ?
```

Thread B

```
1 ?
2 C2;
3 ?
```

Exercice 2 Soit le programme ci-dessous, composé de deux threads concurrents. Complétez ce programme pour qu'il respecte le graphe de précédence [C1] → [C2] → [C3]. Vous pouvez utiliser plusieurs sémaphores. Dans tous les cas, n'oubliez pas d'indiquer la valeur initiale de chaque sémaphore.

Conditions initiales

```
1 ?
```

Thread A

```
1 ?
2 C1;
3 ?
4 C3;
5 ?
```

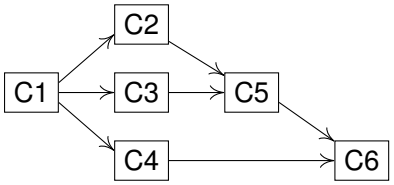
Thread B

```
1 ?
2 C2;
3 ?
```

Exercice 3 On s'intéresse à un programme composé de 6 threads concurrents T1 à T6, où chaque thread Ti exécute un unique calcul Ci. On veut que l'exécution de ce programme respecte les contraintes de précédences représentées ci-dessous, mais on ne veut pas non plus restreindre inutilement la concurrence. Autrement dit, C2, C3 et C4 doivent pouvoir s'exécuter simultanément.

1. Ajoutez à chaque thread les synchronisations nécessaires. Vous pouvez utiliser autant de sémaphores que vous voulez, mais n'oubliez pas de préciser leurs valeurs initiales.

2. Proposez une autre solution avec seulement trois sémaphores.



Exercice 4 Soit le programme ci-dessous, composé de deux threads concurrents. Complétez ce programme avec des sémaphores de telle sorte que l'exécution se déroule toujours dans l'ordre C1-C2-C1-C2-C1-C2...

Conditions initiales

```
1 ?
```

Thread A

```
1 ?
2 while(true) {
3   ?
4   C1;
5   ?
6 }
```

Thread B

```
1 ?
2 while(true) {
3   ?
4   C2;
5   ?
6 }
```

2 Interblocages

Définition : interblocage (en VO deadlock) Dans un programme concurrent, si tous les threads sont simultanément bloqués dans des appels `P()`, alors aucun d'entre eux n'est en mesure de faire un `V()` qui réveillerait les autres et donc le blocage est définitif.

Exercice 5 On s'intéresse au programme ci-dessous, dans lequel deux ressources critiques X et Y sont partagées par quatre threads concurrents A à D. Chaque ressource est «protégée» par un sémaphore binaire pour assurer l'exclusion mutuelle : un thread fait toujours `P(SR)` avant d'utiliser R, et fera un `V(SR)` ensuite pour «déverrouiller» l'accès à la ressource.

Conditions initiales

```
1 semaphore SX = 1
2 semaphore SY = 1
```

Thread A

```
1 while(true) {
2   P(SX);
3   A3; // A utilise X
4   V(SX);
5 }
```

Thread B

```
1 while(true) {
2   P(SY);
3   B3; // B utilise Y
4   V(SY);
5 }
```

Thread C

```
1 while(true) {
2   P(SX);
3   C3; // C utilise X
4   P(SY);
5   C5; // C utilise X et Y
6   V(SX);
7   V(SY);
8 }
```

Thread D

```
1 while(true) {
2   P(SY);
3   D3; // D utilise Y
4   P(SX);
5   D5; // D utilise X et Y
6   V(SX);
7   V(SY);
8 }
```

1. Donnez un scénario d'exécution menant à un interblocage. Exprimez votre réponse sous la forme d'une trace d'exécution de la forme (A1, B1, A2...) pour dire «A exécute sa ligne n° 1, puis B exécute sa ligne n° 1, puis A exécute sa ligne n° 2...»

2. Donnez un scénario dans lequel toutes les sections critiques (A3, B3, D5, etc) sont exécutées, sans pour autant provoquer d'interblocage.

3. (facultatif) Réécrivez les synchronisations de ce programme de façon à garantir l'absence d'interblocage. Vous devez bien sûr garantir l'exclusion mutuelle sur chaque ressource X et Y. Par contre, on ne veut pas limiter artificiellement le parallélisme : par exemple, A3 et B3 doivent pouvoir s'exécuter simultanément, de même que A3 et D3, ou encore C3 et D3, etc.

3 Synchronisation multi-processus

Exercice 6 On s'intéresse au programme ci-dessous. Dessinez le graphe de précédence imposé aux calculs C1, C2, C3, et C4. Parmi les différents schémas de synchronisation vus en cours, comment s'appelle celui-ci ?

Conditions initiales

```
1 semaphore X=0
2 semaphore Y=0
```

Thread A

```
1 C1;
2 V(Y);
3 P(X);
4 C2;
```

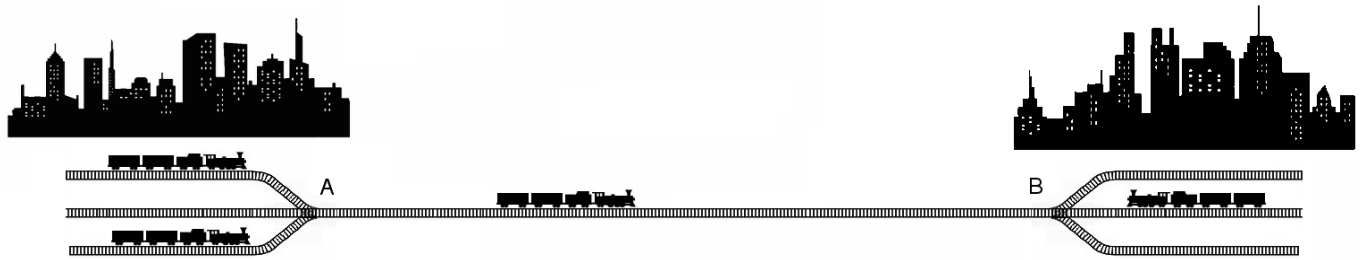
Thread B

```
1 C3;
2 V(X);
3 P(Y);
4 C4;
```

Exercice 7 Écrivez le code nécessaire pour imposer la même synchronisation à *N* threads, à l'aide de *N* sémaphores.

Exercice 8 La même synchronisation peut être obtenue avec seulement deux sémaphores (ainsi que une/des variables partagées si vous en avez besoin). Écrivez le code correspondant.

4 Modélisation concurrente



Deux villes A et B sont reliées par une ligne de chemin de fer à une seule voie. Plusieurs trains peuvent circuler dans le même sens, de A vers B, ou de B vers A, mais des trains circulant en sens opposés ne doivent pas occuper la voie en même temps.
On modélise cette situation sous la forme d'un programme concurrent, dans lequel une quantité arbitraire de threads exécutent chacun l'un ou l'autre des comportements ci-dessous :

Trains de A à B

```
1 ? demander le droit de passage A vers B
2 circuler de A vers B
3 ? sortir de la voie en B
```

Trains de B à A

```
1 ? demander le droit de passage B vers A
2 circuler de B vers A
3 ? sortir de la voie en A
```

Exercice 9 Modifier le code des threads pour que les trains respectent les règles de circulation sur la voie. Il s'agit d'implémenter, pour chaque type de train, la procédure de demande d'autorisation et celle de sortie de voie. Vous pouvez utiliser des sémaphores (pensez à l'initialisation) et des variables partagées (pensez à la synchronisation des accès).