Feuille d'exercices n°10 - Concurrence (corrigé partiel)

Exercice 3 : Rendez-vous à l'aide de sémaphores

Dans cet exercice, on se penche sur des problèmes de rendez-vous entre plusieurs fils d'exécution. La solution proposée diffère selon le nombre de fils d'exécution qui doivent se coordonner et le nombre de fois qu'ils vont se donner rendez-vous.

1. Deux fils d'exécution se rencontrent une fois

On suppose qu'on souhaite synchroniser deux fils d'exécution P_1 et P_2 entre l'exécution de leurs premières instructions, rassemblées sous la notation A, et le reste de leurs instructions, rassemblées sous la notation B. Autrement dit on a la situation suivante.

Algo. du fil d'exécution P_1	Algo. du fil d'exécution P_2
1 A_1 ;	1 A_2 ;
2 RDV avec P_2 ;	2 RDV avec P_1 ;
3 B_1 ;	3 B_2 ;

Q. 1 Proposer une solution qui permette de synchroniser deux fils d'exécution une fois à l'aide de sémaphores. Comment cette solution se généralise à trois fils d'exécution ? à N fils d'exécution ?

Solution

Première solution Pour deux fils P_1 et P_2, on crée S_1 et S_2 deux sémaphores initialisés à 0, gérant respectivement la mise en attente de P_1 et P_2.

Algo. du fil d'exécution P_1	Algo. du fil d'exécution P_2
1 A_1 ;	1 A_2 ;
2 post S_2 ;	2 post S_1 ;
3 wait S_1 ;	3 wait S_2 ;
4 B_1 ;	4 B_2 ;

Pour 3 fils d'exécution on crée S_1, S_2, S_3 trois sémaphores initialisés à 0.

Algo. de P_1	Algo. de P_2	Algo. de P_3
1 A_1 ;	1 A_2 ;	1 A_3 ;
2 post S_2 ;	2 post S_1 ;	2 post S_1 ;
3 post S_3 ;	3 post S_3 ;	3 post S_2 ;
4 wait S_1 ;	4 wait S_2 ;	4 wait S_3 ;
5 wait S_1 ;	5 wait S_2 ;	5 wait S_3 ;
6 B_1 ;	6 B_2 ;	6 B_3 ;

Ça se généralise assez mal : il faut N sémaphores pour N fils d'exécution, et $(N-1)$ appels à

post, et $(N-1)$ appels à wait pour chaque fil d'exécution..

Deuxième solution On continue de travailler avec un sémaphore pour chaque fil, mais on distingue un fil qui sera le chef, disons P_1, il attend tous les autres puis les libère tous.

Algo. de P_1

```
1 A_1 ;
2 wait S_1 ;
3 wait S_1 ;
4 post S_2 ;
5 post S_3 ;
6 B_1 ;
```

Algo. de P_2

```
1 A_2 ;
2 post S_1 ;
3 wait S_2 ;
4 B_2 ;
```

Algo. de P_3

```
1 A_3 ;
2 post S_1 ;
3 wait S_3 ;
4 B_3 ;
```

Pour N fils on aura besoin de N sémaphores, le fil chef fera $N-1$ appels à wait et $N-1$ à post, mais tous les autres fils feront seulement un appel à wait et un à post.

Troisième solution On se rend compte que l'on peut mutualiser la salle d'attente pour tous les fils qui ne sont pas le chef. On utilise donc deux sémaphores : S_{chef} initialisé à $N-2$ et S_{autre} initialisé à 0.

Algo. de P_1

```
1 A_1 ;
2 pour k de 1 à N − 1 faire
3     wait S_chef ;
4 pour k de 1 à N − 1 faire
5     post S_autre ;
6 B_1 ;
```

Algo. de P_j avec $j \neq 1$

```
1 A_j ;
2 post S_chef ;
3 wait S_autre ;
4 B_j ;
```

Cette solution marche très bien pour une rencontre, mais si il doit y avoir des rendez-vous à chaque tour de boucle, rien ne garantit que les appels à post S_{autre} que fait le chef vont bien réveiller les bons fils, par exemple un fil très rapide pour être réveillé par le premier appel, faire rapidement ce qu'il a à faire da,ns B, et revenir faire post $_{\text{chef}}$, et pourrait donc repasser, prenant la place d'un fil qui devait venir au rendez-vous.

2. Plusieurs fils d'exécution se rencontrent une fois

Dans cette section on suppose qu'on a N fils d'exécution $(P_i)_{i\in[\![1,N]\!]}$ à synchroniser. Comme précédemment on note A_i (resp.B_i) les instructions que P_i doit réaliser avant (resp.après) le rendez-vous. Pour réaliser cette synchronisation, on crée un objet qu'on appellera une **barrière**. Cette barrière $\mathcal{B}$ est connue des N fils d'exécution, qui peuvent alors la solliciter pour passer à travers la fonction appel_barriere $(\mathcal{B})$, comme les piétons appellent le feu vert avant de traverser une route. Lorsqu'un fil d'exécution P_i appelle cette fonction, c'est qu'il est au rendez-vous : il a fini A_i et il attend les autres fils d'exécution pour passer la barrière et faire B_i après.

Q. 2 Proposer une implémentation en pseudo-code d'un tel objet barrière et de la fonction appel_barriere. On pourra utiliser verrous et sémaphores. On veillera aussi à expliciter comment un tel objet doit être initialisé dans une fonction cree_barriere.

Solution

On définit un objet de type barrière comme une structure ayant 4 champs :

- un champ nb_p qui enregistre le nombre de fils d'exécution participant au rendez-vous ;
- un sémaphore s qui met en attente les fils d'exécution et les réveillera après ;
- un entier nb_att qui indique combien de fils d'exécution sont en train d'attendre à la barrière, afin que le dernier arrivé puisse constater qu'il est le dernier et qu'il doit réveiller tous les autres ;
- un verrou v pour protéger l'entier nb_att.

On définit alors les fonctions appel_barriere et cree_barriere comme suit.

```
cree_barriere (N : int) → barrière
1 B ← objet de type barrière ;
2 B.nb_p ← N ;
3 B.nb_att ← 0 ;
4 B.s ← cree_sémaphore(0) ;
5 B.v ← cree_verrou ;
6 retourner B ;
```

```
appel_barriere (B : barrière)→ ()
1 lock(B.v) ;
2 B.nb_att ← B.nb_att + 1 ;
3 if B.nb_att = B.nb_p then
4     faire nb_p−1 fois : post(B.s) ;
5     B.nb_att ← 0 ;
6     unlock(B.v) ;
7 else
8     unlock(B.v) ;
9     wait(B.s) ;
```

3. Plusieurs fils d'exécution se rencontrent plusieurs fois

Dans la section précédente, on a proposé une solution pour que $N \in \mathbb{N}^*$ se rencontrent une fois. Cependant on peut aisément imaginer des algorithmes où une famille de fils d'exécution doit se réunir à chaque tour de boucle.

Q. 3 Que dire de la solution qui consiste à créer, avant de lancer les fils d'exécution, une barrière b comme proposé ci-avant, et de demander à chaque fil d'exécution d'attendre les autres à chaque tour en appelant à `attend_barriere b` ? Est-elle robuste ? Plus précisément, cette solution convient-elle si les fils d'exécution ne s'exécutent pas tous autant de fois ?

Solution

La solution précédente convient à condition que le nombre de fils d'exécution en jeu à chaque tour reste le même. En effet, lorsqu'on crée une barrière, on fixe le nombre de fils d'exécution qui doivent être arrivés à la barrière avant de débloquer tout le monde. Dès lors qu'un fil d'exécution est arrêté, il va paraître manquant, et les autres fils d'exécution vont inutilement l'attendre à la barrière. Pour pallier ce problème, il faudrait qu'en fin d'exécution avant de mourir, un fil d'exécution puisse signaler qu'on ne l'attende plus.

Q. 4 Proposer une amélioration de l'implémentation de barrière proposée ci-avant qui soit robuste à la disparition de certains fils d'exécution initialement rattachés à la barrière. On pourra supposer qu'il s'agit plus de départ volontaire que de disparition, et que les fils d'exécution se signalent avant de disparaître.

Solution

Comme suggéré à la question précédente on doit pouvoir modifier le nombre de fils d'exécution à attendre, ainsi le champ nb_p doit être mutable, et donc lui aussi protégé par un verrou. De plus pour que le test $\mathcal{B}$.nb_att $=$ $\mathcal{B}$.nb_p ait du sens, il ne faut pas qu'un fil d'exécution signale son départ juste entre ce test et l'action qui va avec, sinon le fil d'exécution qui a fait ce test et obtenu une réponse négative ne va pas libérer les autres car il ne se croit pas le dernier. En fait cela soulève le problème du cas où un fil d'exécution signale son départ alors que tous les autres se sont déjà endormis à côté de la barrière... Ainsi le fil d'exécution qui signale sont départ se doit de tester s'il est le dernier qu'on attendait et alors libérer tous les autres. Mais là encore il ne faudrait pas qu'un malencontreux entrelacements des instructions fasse que deux fils d'exécution (l'un arrivant à la barrière et l'autre voulant signaler de ne plus l'attendre) libèrent chacun les autres... On décide donc d'utiliser le même verrou.

Les procédures de création et d'appel restent les mêmes, mais on ajoute donc la procédure de départ suivante.

signale_départ ($\mathcal{B}$: barrière)$\rightarrow$ ()

```
1 lock(B.v) ;
2 B.nb_p ← B.nb_p − 1 ;
3 if B.nb_att = B.nb_p then
4 |   faire B.nb_att fois : post(B.s) ;
5 |   B.nb_att ← 0 ;
6 unlock(B.v) ;
```