Le « Dîner des Philosophes »

1 Présentation du problème

N philosophes discutent autour d'un repas. Il y a bien N assiettes pleines de spaghettis, mais seulement N fourchettes. Pour manger, un philosophe doit prendre la fourchette de gauche, puis la fourchette de droite.

Un philosophe va passer successivement, et en boucle, dans les états suivants :

- Il a faim, et tente de prendre les fourchettes dans l'ordre précisé
- Il mange
- Il pense

Les conditions initiales sont les suivantes :

- Avant de commencer, les philosophes sont tous en train de penser
- À l'instant $t = 0$, ils ont tous faim en même temps
- La fourchette à la gauche d'un philosophe aura le même ID que le philosophe lui-même

Ce problème classique de blocage entre tâches est légèrement modifié ici pour notre cours de temps-réel. On va poser les contraintes suivantes :

- Le temps passé dans les états « mange » et « pense » est aléatoire, uniformément réparti dans un intervalle qui vous est précisé dans le code source fourni
- Les fourchettes sont libres au départ
- Les philosophes ont tous faim au départ
- Pour les solutions 1 et 2 : lorsqu'il a fini de manger, un philosophe relâche immédiatement les 2 fourchettes (dans le même ordre de leur acquisition)

Il s'agira de temps-réel si l'on pose une contrainte dure, par exemple sur la durée maximale que peut passer un philosophe dans l'état « veut manger ».

1.1 Récupérer les codes sources de base de ce programme console, les compiler et lancer une exécution en Debug dans Eclipse.

1.2 Envoyer un signal `SIGTERM` par la commande `kill` depuis un terminal. Vérifier qu'on ait bien un affichage de fin de programme dans la console, expliquer dans les grandes lignes comment a fonctionné ce signal.

2 Sémaphores de simulation des fourchettes

2.1 Coder la création et la destruction des sémaphores représentant les fourchettes, puis tester.

2.2 Coder la création des threads des philosophes, puis leur destruction après la réception d'un signal `SIGTERM`.

2.3 Coder les attentes aléatoires (fonction `usleep`) de simulation des actions « manger » et « penser » dans la fonction `vieDuPhilosophe`.

2.4 Coder l'acquisition des fourchettes par les philosophes (dans leur fonction `vieDuPhilosophe`) et dans le même temps coder le relâchement des ressources fourchettes.

2.5 Constater en pratique la situation d'inter-blocage entre tâches. Pour forcer l'inter-blocage, on pourra mettre une pause de l'ordre de la dizaine de millisecondes entre l'acquisition de la fourchette gauche et l'acquisition de celle de droite.

2.6 Expliquer pourquoi l'apparition de l'inter-blocage est aléatoire (et de faible probabilité), et pourquoi ceci en fait un problème d'autant plus ennuyeux.

3 Première résolution du problème : scrutation par un ordonnanceur maître [approche synchrone]

Cette solution est la moins élégante, mais la plus simple à appréhender et à mettre en œuvre. La plus simple ne signifie cependant pas qu'il n'y a pas de piège, surtout en considérant le fait que l'on ne travaille pas sur un vrai système d'exploitation temps-réel.

L'idée générale est d'introduire notre propre ordonnanceur, qui va scruter de manière cyclique l'état de tous les philosophes, pour prendre une décision selon la situation actuelle (et éventuellement passée). La politique d'ordonnancement sera pour l'instant assez basique, telle que :

- Si on a un nombre pair N de philosophes, on autorise le 0, le 2, le 4, ..., le $N-2$ à manger, et on attend qu'ils aient fini. Puis on autorise le 1, le 3, ..., le $N-1$, et on attend qu'ils aient fini. Puis on autorise le 2, le 4, ..., le 0, et on attend, etc, etc...
- Si on a un nombre impair N de philosophes, on appliquera une politique similaire en faisant attention au piège dans ce cas
- Attention : pour que cela fonctionne, tous les philosophes du prochain groupe qui peut manger doivent commencer à manger au même moment

3.1 Prévoir des mécanismes de synchronisation entre tâches, qui permettront de contrôler les philosophes dans leur fonction `vieDuPhilosophe`, et ce depuis la boucle de scrutation de l'ordonnanceur.

3.2 Implémenter et tester la politique d'ordonnancement précisée.

4 Seconde résolution : ordonnanceur personnalisé invoqué par les philosophes eux-mêmes [approche asynchrone]

On va ici appliquer la même politique d'ordonnancement, mais sans avoir recours à un mécanisme de scrutation cyclique. Toute la logique d'ordonnancement, les appels à des objets POSIX, etc., devront être contenus dans la fonction `actualiserEtAfficherEtatsPhilosophes(int, char)`. Vous pourrez bien sûr séparer votre code d'ordonnancement dans d'autres fonctions appelées à partir de cette dernière.

L'idée est que l'ordonnancement soit fait de manière « transparente » pour le philosophe : un philosophe transmet à l'ordonnanceur ses changements d'état, et l'ordonnanceur peut alors décider ou non de le bloquer, selon la situation.

4.1 Coder cette seconde solution en respectant les contraintes.

4.2 Avantages et inconvénients par rapport à la solution précédente ?

4.3 Calculer le temps max durant lequel un philosophe est affamé. Peut-on alors parler de système temps-réel dur ? On suppose que les latences du S.E. sont négligeables par rapport aux temps d'attente.

5 Comparaison des résultats et conclusion

5.1 Intégrer dans les programmes des mesures de temps d'attente par philosophe (minimum, moyenne, médiane, maximum, ...)

5.2 Selon le critère précédent, et d'autres critères que vous définirez, comparer les méthodes de résolution du problème dit du « dîner des philosophes ».

5.3 Citer quelques exemples concrets (industriels, de la vie courante, ...) dans lesquels on peut appliquer les solutions programmées ici.

5.4 Expliquer pourquoi l'on ne rencontrera pas le problème d'inversion de priorité sur un système d'exploitation à usage général, lorsque l'on travaille avec des sémaphores ou des mutex.