Mini-projet #10 – Evitement d'inter-blocages

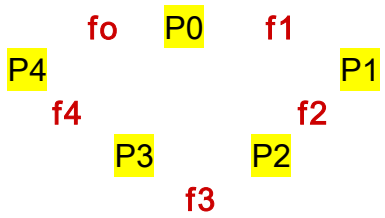
Cours Applications Concurrentes, *Polytech-INFO4*

F. Boyer, Université Grenoble Alpes, année 2022-2023

1. Introduction

Dans ce TD/TP nous allons considérer un problème de synchronisation classique (appelé *le problème des philosophes*), illustrant le problème des inter-blocages et différentes méthodes permettant de les prévenir.

Dans le problème des philosophes, N philosophes sont assis autour d'une table où sont disposées N fourchettes (figure ci-dessous, avec N = 5). Un philosophe passe son temps à manger pendant un temps indéterminé, puis à penser pendant un temps indéterminé. Pour manger, tout philosophe doit disposer des 2 fourchettes qui sont de part et d'autre de son assiette.



Dans la modélisation de ce problème, chaque philosophe est représenté par un thread, instance de la **classe *Philo***, et l'allocation des fourchettes est gérée par une instance de la **classe *Table***. Chaquephilosophe exécute une boucle infinie, dans laquelle il acquiert les fourchettes dont il a besoin, les utilise puis les relâche avant de penser.

```
thread Philo (int id, Table table) :

    while (true) {
      table.beginEat(id);
      Thread.sleep(eatTime);
       table.endEat(id);
      Thread.sleep(thinkTime); }
```

Ce qui nous intéresse est la programmation de la classe *Table*. Les invariants qui doivent être garantis sont les suivants :

- une fourchette ne peut être allouée qu'à un philosophe à la fois

- un philosophe ne peut s'allouer que les fourchettes qui sont immédiatement à sa droite et à sa gauche

Remarque : le problème des philosophes est représentatif des situations dans lesquelles plusieurs threads cherchent à s'allouer des ensembles de ressources différenciées (les demandes d'allocation de fourchettes concernent des fourchettes bien identifiées), mais se recouvrant les uns les autres (par exemple, la fourchette f0 peut être demandée par P4 et P0).

2. Solution de type moniteur

Nous allons tout d'abord considérer la mise en œuvre de la classe *Table* sous la forme d'un moniteur.

Exercice. Définissez le tableau de gardes-actions associé à la classe *Table* et déduisez-en la solution directe. Pour la définition du tableau de gardes-actions, on peut choisir de modéliser le système via l'état des fourchettes (libres/ allouées), ou bien via l'état des philosophes (mange/ne mange pas).

3. Solution directe – discussion

La solution directe comporte un risque de famine : il se peut que certains philosophes ne mangent jamais.

Pour éviter cela, on peut envisager d'évoluer la solution directe vers une solution FIFO. Cela évite la famine, mais limite le parallélisme et reste donc non optimal du point de vue de l'usage des ressources. En effet, dès qu'un philosophe ne peut pas obtenir les fourchettes dont il a besoin, il bloque les suivants alors que certains pourraient manger.. Par exemple avec le scénario suivant :

- P1 mange,
- P2 demande à manger (il est mis en attente car P1 utilise f1 & f2),
- P3 demande à manger (il pourrait manger mais il est mis en attente car il arrive après P2..).

On pourrait évoluer la solution directe vers une solution sans famine et favorisant le parallélisme, en analysant finement l'état du système et en gérant des priorités dans l'allocation des fourchettes selon qu'un philosophe a pu manger récemment ou pas. Cela reste compliqué à mettre en oeuvre.

Lorsque l'on considère un problème d'allocation de ressources dans lequel les ressources sont différenciées, il est parfois plus simple de se tourner vers des solutions à base de sémaphores.

3. Solution basée sur les sémaphores

Quand les ressources sont différenciées, il est classique d'utiliser un sémaphore par ressource, gérant l'allocation et la libération de la ressource en question.

Exercice. Proposez une implémentation de la classe *Table* dans laquelle on représente chaque fourchette par un sémaphore.

Evaluation : cette solution pose t'elle un problème d'interblocage ? si oui, identifiez un scénario menant à un interblocage.

3. Solutions sans inter-blocages basées sur les sémaphores

Différentes options peuvent être considérées pour éviter les interblocages :

- Réaliser une allocation globale
- Limiter le nombre de threads qui demandent des ressources
- Réaliser une allocation ordonnée des ressources

De manière générale, toutes ces options poursuivent un unique objectif : empêcher la formation d'un **cycle** dans le graphe des attentes des threads.

A) Allocation globale

Le principe est que chaque thread acquiert toutes les ressources dont il a besoin de manière simultanée. Autrement dit, si un thread a besoin durant son exécution de s'allouer les ressources R1, R2, .. Rk, alors il va demander une allocation atomique de cet ensemble de ressources :

> → *soit toutes les ressources demandées sont libres, et dans ce cas elles sont allouées au demandeur, soit certaines ressources parmi celles demandées ne sont pas libres, auquel cas aucune ressource n'est allouée au demandeur et celui-ci est mis en attente.*

Notons que la solution directe met en oeuvre une allocation globale. Comme on l'a vu, cette solution directe présente cependant un risque de famine.

Exercice. Voici une nouvelle version de la classe Table. Cette version met-elle en œuvre une allocation globale ? Que pensez-vous de son efficacité ?

```
public void Table(int N) {
      global = new Semaphore(1);
      fourch = new semaphore[N];
      for (int i = 0; i < N; i++)
       fourch= new Semaphore[1];
   }
   public void beginEat(int i) {
      global.P()
        fourch[i].P();
        fourch[i+1%N].P()
      global.V()
   }
```

B) Limitation du nombre de threads

Dans le cas du problème des philosophes, il faut que les N philosophes soient bloqués pour former un cycle. Pour éviter les interblocages, une option est donc de limiter à N-1 le nombre de philosophes pouvant se bloquer dans l'acquisition des fourchettes.

Exercice. En repartant de la solution donnée en 2 pour la classe *Table*, proposez quelques transformations pour évoluer vers une limitation du nombre de philosophes demandant à s'allouer des fourchettes.

C) Prise de ressource ordonnée

Si l'on dispose d'un ordre total sur les ressources (par exemple, via leur identifiant), l'idée de l'allocation ordonnée est de forcer les ressources à être demandées dans un ordre croissant (ou décroissant).

Exercice : appliquez ce principe à la solution donnée en 2 pour la classe *Table.*

Note finale. Certaines solutions que nous avons considérées dans ce TD (ordonnancement des allocations, allocation globale atomique) imposent de connaître à l'avance l'ensemble des ressources dont va avoir besoin un thread. Il n'est pas toujours possible d'avoir cette information, et c'est la raison pour laquelle on est parfois obligé de traiter les interblocages par des techniques d'évitement, de détection-guérison, ou par un kill-restart de l'application.