# TP3 : Le problème des philosophes

RS : Réseaux et Systèmes

Deuxième année

Cinq philosophes, réunis pour philosopher, ont au moment du repas un problème pratique à résoudre. En effet, le repas est composé de spaghettis qui, selon la coutume de ces philosophes, se mangent avec deux fourchettes. Or, la table n'est dressée qu'avec une seule fourchette par couvert. Les philosophes décident d'adopter le rituel suivant :

- Chaque philosophe prend une place à table.
- Chaque philosophe qui mange utilise la fourchette à sa droite et celle à sa gauche (pas celle d'en face).
- À tout instant, chaque philosophe est dans l'un des états suivants :
  - il mange avec deux fourchettes ;
  - il a faim, et attend la fourchette de droite, celle de gauche ou les deux ;
  - il pense, et n'utilise pas de fourchette.
- Initialement, tous les philosophes pensent.
- Un philosophe qui mange s'arrête en un temps borné.

L'objectif du TP est d'expérimenter diverses solutions à ce problème. Copiez les fichiers de `/home/depot/2A/RS/philosophes` dans un répertoire chez vous. `philosophes.c` et `philosophes.h` constituent un simulateur de philosophes. Pour pouvoir l'utiliser, **il n'est pas nécessaire de modifier les fichiers** `philosophes.[ch]`. À la place, il faut ajouter (dans un fichier séparé) trois fonctions manquantes spécifiant le comportement des philosophes :

- `initialize_data()` : Une fonction réalisant toutes les initialisations de globales dont vous avez besoin. Elle reçoit en argument le nombre de philosophes assis autour de la table.
- `pickup()` : Ce que fait un philosophe qui a faim pour prendre ses fourchettes. Elle reçoit en argument le numéro du philosophe qui l'appelle, ainsi que le compte de philosophes autour de la table.
- `putdown()` : Ce que fait un philosophe rassasié pour reposer ses fourchettes. Ses arguments sont similaires à `pickup`.

Une fois que le comportement des philosophes est spécifiés, il devient possible de compiler le simulateur : `gcc -pthread philosophes.c mon_philosophe.c -lm -o mon_philosophe`. Le programme généré attend deux arguments obligatoires plus un optionnel : le nombre de philosophes assis autour de la table, le nombre maximal qu'un philosophe passe à dormir et le temps de simulation souhaité (la simulation est infinie si on ne spécifie pas de durée maximale). `./mon_philosophe 5 1 20` simule 5 philosophes pouvant dormir au plus 1 seconde à chaque fois, et termine la simulation après 20 secondes. À intervalles réguliers, des lignes résument le temps que chaque philosophe a passé bloqué.

```
10.0 Total blocktime : 13 : 2.9 3.1 3.6 2.6 3.9 (2 CS violations)
```

Après 10 secondes, les philosophes ont attendu 13 secondes au total. Le premier a attendu 2.9 secondes, le second 3.1, et ainsi de suite. En plus de tenir ces chronométrages à jour, le simulateur vérifie qu'aucun philosophe ne mange en même temps que l'un de ses voisins, car cela voudrait dire que la fourchette placée entre les deux est utilisée par deux personnes en même temps. Dans l'exemple ci-dessus, un tel événement est arrivé deux fois (le simulateur indique deux violations de section critique).

L'objectif est d'obtenir une solution sans violation de section critique, sans interblocage, efficace (le temps total d'attente est minimal) et équitable (l'attente est bien répartie entre les processus). Pour que ces grandeurs soient représentatives, il convient de laisser tourner la simulation au moins cent fois plus que le temps de sommeil maximal.

Le répertoire des sources contient également une bibliothèque d'interposition permettant de détecter les interblocages dans un programme pthread sans devoir modifier ce dernier. Voir l'annexe à ce sujet à la fin de l'énoncé.

★ **Exercice 1:** Le fichier `philo_null.c` contient une implémentation vide du philosophe : chacun commence à manger sans vérifier si les couverts sont disponibles ou non. Compilez le programme, et exécutez le. Constatez que le nombre de violations de section critique augmente avec le temps.

```
gcc -pthread philosophes.c philo_null.c -lm -o philo_null
```

★ **Exercice 2:** Le fichier `philo_naive.c` contient l'implémentation naïve, consistant à prendre la fourchette à sa droite puis la fourchette à sa gauche en vérifiant leur disponibilité avec un mutex. Compilez le programme, et exécutez le. Contrairement à ce qu'on attend, ce programme n'entre pas en interblocage (si cela ne vous étonne pas, relisez le TD correspondant).

Ceci est dû au fait qu'il faut qu'un philosophe soit interrompu entre le moment où il prend sa fourchette de gauche et la ligne suivant, où il prend sa fourchette de droite pour provoquer un interblocage. Ajoutez un sleep(1) entre les deux lignes, et remarquez la différence.

★ **Exercice 3:** Écrivez une version `philo_ordo` évitant l'interblocage en ordonnant les requêtes. On peut par exemple verrouiller $min(id, (id+1)\%count)$ en premier et $max(id, (id+1)\%count)$ ensuite. Cela revient à dire que tous les philosophes sont droitiers, sauf le dernier qui est gaucher.

★ **Exercice 4:** Écrivez une version `philo_parite` dans laquelle les philosophes dont l'identifiant est pair sont droitiers (ils prennent leur fourchette droite en premier) tandis que les philosophes impaires sont gauchers.

▷ **Question 1:** L'interblocage est-il possible ? Pourquoi ?

**Réponse**

Non, car on ordonne encore les requêtes (les fourchettes pairs d'abord, et les impaires ensuite). Le fait que les pairs soient pas ordonnés entre eux n'est pas important car personne ne cherche à prendre plusieurs fourchettes paires. Donc, on respecte en fait tous les ordres où les pairs sont devant et les impaires derrières. Par exemple :

$$x \overset{?}{>} y = \begin{cases} \text{VRAI si x pair ET y impair} \\ \text{FAUX si x impair ET y pair} \\ \text{VRAI si x pair ET y pair ET } x > y \text{ (ordre des entiers)} \\ \text{VRAI si x impair ET y impair ET } x > y \text{ (ordre des entiers)} \end{cases}$$

**Fin réponse**

▷ **Question 2:** Comparez l'équité de cette version par rapport à la solution de l'exercice précédent.

**Réponse**

Je sais pas ce qu'on constate, c'est juste pour leur faire écrire et les faire réfléchir un peu :)

**Fin réponse**

★ **Exercice 5:** Écrivez une version `philo_global` évitant l'interblocage en réalisant des réservations globales (ne pas prendre la moindre fourchette si on n'est pas sûr de pouvoir avoir les deux). On introduit une variable d'état pour chaque philosophe c[i] qui vaut 0 si le philosophe $i$ pense, 1 si il veut manger et 2 si il mange. Le tableau `c` doit être partagé par tous les processus de façon à ce que chacun puisse tester l'état de ses voisins. Le tableau doit naturellement être protégé par mutex.

Un philosophe ayant faim consulte le tableau pour savoir s'il peut manger. Si oui, il modifie le tableau (pour bloquer ses voisins) ; si non, il s'endort sur une sémaphore privée pour que ses voisins le réveillent quand ils ont fini.

▷ **Question 1:** Isolez les fonctions pickup() et putdown() dans le code suivant, et traduisez les en C dans `philo_global`. **Ne modifiez pas philosophe.c**

```
fonction test(k):
   si c[k] = 1 et c[k+1] != 2 et c[k-1]!=2
   alors
     c[k]:=2
     V(sempriv[k]) /* on libere le philosophe de son attente */

fonction philosopher(i):
  penser()
  P(mutex); /* pour proteger le tableau c */
  c[i]:= 1; /* i a faim puisqu'il a fini de penser */
  test(i); /* il regarde si il peut manger; si oui, V(sempriv[i]) est fait */
  V(mutex); /* pour que les voisins puissent rendre les fourchettes s'ils les avaient*/
  P(sempriv[i]); /* Si le V est fait, ça continue et il mange.
                Sinon il attend qu'un voisin le réveille en libérant les fourchettes */
  manger()
  P(mutex); /* on va dire a tout le monde que l'on a fini de manger */
  c[i]:= 0; /* on pose les couverts */
  test(i-1); /* on regarde si les voisins peuvent manger maintenant que l'on */
  test(i+1); /* a posé les couverts et on fait leur V(sempriv[i]) si oui */
  V(mutex);
```

▷ **Question 2:** Modifiez l'algorithme pour utiliser une variable de condition à la place de la sémaphore.

★ **Exercice 6:** (solution du maître de table).

Nous allons maintenant appliquer la troisième méthode du cours pour éviter les interblocages en modifiant l'algorithme. Le principe est de ne pas prendre sa première fourchette s'il s'agit de la dernière posée sur la table. Cela s'implémente simplement avec une sémaphore initialisée à $count - 1$ et prise (une seule fois) par chaque philosophe avant de manger.

▷ **Question 1:** Implémentez cette approche dans `philo_maitre`.

★ **Exercice 7:** Tous les algorithmes précédents présentent un risque de famine. Si le hasard veut que les philosophes 0 et 2 mangent, puis 1 et 3, puis 0 et 2, et ainsi de suite, il est possible que 4 ne parvienne jamais à manger (si l'implémentation des mutex ne garantie pas que les threads obtiennent le verrou dans l'ordre où ils l'ont demandé – ce n'est pas imposé par le standard).

Pour corriger ce problème, il faut s'assurer nous même que les philosophes sont servis dans l'ordre. Quand l'un d'entre-eux appelle `pickup()`, s'il a déjà quelqu'un dans la queue, le philosophe s'y ajoute. Si la queue est vide, il tente de prendre ses fourchettes. S'il réussi, il mange et il est placé dans la queue sinon. Lorsque l'on pose les fourchettes, on regarde si cela libère le premier de la file.

▷ **Question 1:** Appliquez cette approche dans `philo_fifo`. Remarquez que cette approche amène des temps de blocage bien plus long puisque certains philosophes sont bloqués dans la file d'attente alors que leurs fourchettes sont libres.

★ **Exercice 8:** Saurez-vous proposer une solution garantie sans famine mais efficace ?

## Détection d'interblocages par bibliothèque d'interposition

Dans les sources fournis sur `/home/depot/2A/RS/`, vous trouverez un fichier `pthread_interposer.c`. Il s'agit du source d'un petit outil que vous pouvez utiliser pour détecter de façon relativement sûre les interblocages dans vos programmes pthreads.

Son principe général est d'intercepter tous les appels aux fonctions `pthread_mutex_lock` et consoeurs, et d'ajouter des vérifications pour voir si un cycle de dépendances se crée entre les threads en attente de verrous possédés par d'autres.

En ce qui concerne la détection de deadlock, on construit une structure partagée (donc elle même protégée par mutex) indiquant quelle ressource est possédée par quel thread, et quel thread est en attente de quelle ressource. Il ne reste plus qu'à la parcourir parfois pour chercher un cycle. Notons que ce n'est pas infaillible car l'interposeur change le comportement de l'application observée, puisqu'on ajoute des barrières de synchronisation pour ne pas avoir de compétition sur notre structure partagée. Votre code peut donc entrer en interblocage sans l'interposeur, et fonctionner très bien avec (ou le contraire).

En ce qui concerne l'interception elle-même, on compile le code comme une bibliothèque partagée

(`pthread_interposer.so`, voir l'entête du source sur comment faire), et on demande au chargeur dynamique d'utiliser préférentiellement notre code plutôt que le code système. Il suffit de définir une variable d'environnement `LD_PRELOAD` contenant le chemin des bibliothèques à interposer de la sorte.

Notre bibliothèque contient par exemple une fonction nommée `pthread_mutex_lock`. Donc, quand le programme exécuté avec le mécanisme d'interposition en place voudra appeler une fonction de ce nom, c'est notre version qui sera invoquée et non celle de la bibliothèque pthread du système (le contenu de LD_PRELOAD prend le pas sur LD_LIBRARY_PATH et sur les indications de chargement dynamique présentes dans l'exécutable).

Notre objectif n'est pas de réimplémenter complètement cette fonction, mais simplement d'ajouter du code avant et après l'appel à la vraie fonction. Il faut donc que notre version puisse appeler la version système après avoir vérifié qu'il n'y a pas de deadlock. Oui, mais comment appeler une fonction du même nom que la fonction courante, mais venant d'une autre bibliothèque ?

On utilise les fonctionnalités du chargeur dynamique linux pour récupérer un pointeur sur la fonction. Par exemple, le code suivant initialise le pointeur sur fonction `mutex_lock` pour qu'il pointe sur le prochain symbole "pthread_mutex_lock" défini dans les bibliothèques dynamiques actuellement chargées. Ensuite, `mutex_lock` est utilisable comme une fonction normale, et appelle la fonction du système...

```
1  mutex_lock = (int(*)(pthread_mutex_t*)) dlsym(RTLD_NEXT, "pthread_mutex_lock");
2  mutex_lock(&the_mutex);
```

```
1 $ LD_PRELOAD=./pthread_interposer.so ./philo_naive 3 1 20
2 [...]
3 DEADLOCK DETECTED:
4   - Thread 3 waits for lock 0, owned by thread 1.
5   - Thread 1 waits for lock 1, owned by thread 2.
6   - Thread 2 waits for lock 2, owned by thread 3.
```

Ce mécanisme d'interposition de bibliothèques partagées permet de réaliser des tours de passe-passe en système, et nous le mettons par exemple souvent en œuvre pour automatiser la correction de projets de programmation.

Il existe un mécanisme équivalent sous Mac OS X (avec une syntaxe complètement différente), mais Windows ne permet malheureusement pas de définir une interposition pour un processus donné. Si on donne un interposeur, tous les programmes du système l'utilisent, ce qui peut poser problème si l'interposeur est un peu ≪agressif≫.