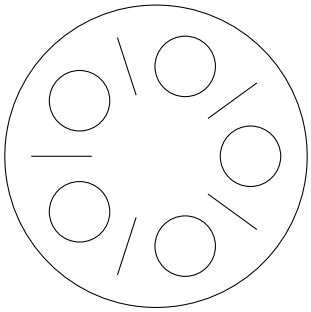
I Problème du dîner des philosophes

Le problème du dîner des philosophes est un problème théorique. Autour d'une table ronde se trouvent 5 philosophes. Malheureusement, lors de la préparation de la table, le serveur a installé une seule baguette de part et d'autre de chaque assiette. Pour pouvoir manger, chaque philosophe a besoin d'utiliser deux baguettes simultanément (les deux qui sont de part et d'autre de son assiette), et les repose au même endroit une fois qu'il a terminé de manger. On souhaite réfléchir à une solution où chaque philosophe peut manger, en respectant l'absence de famine :

- au départ, tous les philosophes *pensent* ;
- à tout moment, un philosophe en train de penser peut *avoir faim* ;
- après un temps fini à avoir faim, le philosophe doit pouvoir commencer à *manger* ;
- après avoir mangé pendant un certain temps, le philosophe est repu et peut recommencer à penser.



On s'intéresse à différentes solutions. On supposera qu'on dispose des primitives suivantes :

- `penser` : le philosophe pense ; quand un philosophe a fini de penser, il a nécessairement faim ;
- `manger` : le philosophe mange pendant un temps fini ; quand un philosophe a fini de manger, il se remet à penser.

Le schéma le plus général pour un philosophe est le suivant :

```
philosophe(i)

while true :
    penser()
    Prendre les baguettes.
    manger()
    Poser les baguettes.
```

1. Est-il possible que trois philosophes soient simultanément en train de manger ? Deux philosophes ?
2. Pour chacune des solutions suivantes, écrire une fonction `philosophe`(i) en pseudo-code qui décrit le comportement du i-ème philosophe, et indiquer quelles contraintes sont respectées parmi l'exclusion mutuelle et l'absence de famine.
   (a) Dans une première solution, on utilise un seul mutex m pour l'ensemble des philosophes. Le philosophe qui verrouille le mutex est le seul qui est autorisé à manger.
   (b) Dans une deuxième solution, on utilise un tableau de booléen D de taille 5 qui indique quelle baguette est disponible. Ainsi, $D[i]$ vaut vrai si la i-ème baguette est disponible. On suppose que le i-ème philosophe doit utiliser les baguette i et $i+1$ pour manger (avec la convention qu'on considère ces indices modulo 5). Enfin, le tableau est verrouillé en écriture par un mutex global (pour éviter que deux philosophes écrivent simultanément dedans). Lorsqu'un philosophe a faim, il doit d'abord vérifier que ses baguettes sont disponibles avant de les prendre, et les remettre comme disponible lorsqu'il a fini de manger.
   (c) Dans une troisième solution, chaque baguette est protégée par un mutex, dans un tableau M de taille 5. Lorsqu'un philosophe d'indice i veut manger, il commence par acquérir sa baguette de gauche (celle d'indice i), puis la conserve jusqu'à ce qu'il puisse acquérir celle de droite (d'indice $i+1$) et mange.
3. En s'inspirant de la troisième solution, proposer une solution correcte (on pourra traiter le philosophe d'indice 4 différemment).
4. En s'inspirant de la troisième solution, proposer une autre solution correcte avec un sémaphore initialisé à 4.

II Algorithme de Dekker

```
create()
want = [false, false]
turn = 0
```

```
lock(i)
want[i] = true
while want[1 - i] :
    if turn != i then
        want[i] = false
        while turn != i :
            wait
        want[i] = true
```

```
unlock(i)
turn = 1 - i
want[i] = false
```

Algorithme de Dekker

Montrer que l'algorithme de Dekker permet d'implémenter un mutex pour deux threads, c'est-à-dire qu'il respecte l'absence de famine et l'exclusion mutuelle.

III Barrière de synchronisation

Une barrière de synchronisation, ou rendez-vous est un objet de synchronisation permettant à des fils d'exécution de se synchroniser au cours de leur exécution (donc pas une jointure, qui permet de se synchroniser après l'exécution). On veut pouvoir réaliser les opérations suivantes :

- `creer(n)` prend en argument un entier et renvoie une barrière de synchronisation pour n fils d'exécution ;
- `rdv(b)` prend en argument une barrière de synchronisation et attend qu'un total de n fils aient fait un appel à cette fonction avant de continuer.

On implémente une barrière de synchronisation avec un compteur (le nombre de fils qui attendent), un mutex (qui protège le compteur) et un sémaphore. La fonction de création est la suivante :

```
creer(n)
return {compteur = n, m = Mutex(), s = Semaphore(0)}
```

1. Quel problème se pose avec chacune des solutions suivantes ?

(a)

```
rdv(b)
Verrouiller b.m.
b.compteur←b.compteur - 1.
Déverrouiller b.m.
if b.compteur = 0 :
    Post b.s.
Wait b.s.
```

(b)

```
rdv(b)
Verrouiller b.m.
b.compteur←b.compteur - 1.
if b.compteur = 0 :
    Post b.s.
Wait b.s.
Post b.s.
Déverrouiller b.m.
```

2. Adapter les solutions précédentes pour résoudre le problème.
3. Comment modifier la structure pour pouvoir réutiliser la barrière de synchronisation plusieurs fois (avec le même nombre de fils) ?