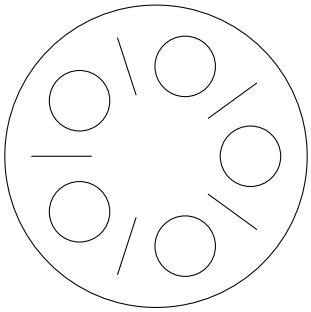
I Problème du dîner des philosophes

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.

Solution :

1. Il n'est pas possible que trois philosophes soient en train de manger simultanément, car cela nécessiterait 6 baguettes (et il n'y en a que 5). Il est possible que deux philosophes mangent simultanément tant qu'ils ne sont pas voisins.
2. (a) On propose :

```
philosophe(i)
while true :
    penser()
    Verrouiller m.
    manger()
    Déverrouiller m.
```

L'exclusion mutuelle est respectée (c'est le principe du mutex). Il en est de même pour l'absence d'interblocage (car le mutex est sans interblocage). Par contre il y a un risque de famine : il est possible que deux philosophes mangent tour à tour et bloquent les trois autres. De plus, il est dommage de ne limiter qu'à un seul philosophe qui mange étant donnée la remarque de la première question.

(b) Il y a ici en fait deux manières d'interpréter le processus : le test de disponibilité se fait-il en verrouillant le mutex ou non ?

- dans le premier cas :

```
philosophe(i)
while vrai :
    penser()
    while vrai :
        Verrouiller m.
        if D[i] ∧ D[i + 1] :
            D[i] ← faux.
            D[i + 1] ← faux.
            Déverrouiller m.
            Sortir de la boucle.
        Déverrouiller m.
    manger()
    D[i] ← vrai.
    D[i + 1] ← vrai.
```

Il y a bien exclusion mutuelle et absence d'interblocage, mais il y a un risque de famine, similaire à la situation précédente ;

- dans le deuxième cas :

```
philosophe(i)
while vrai :
    penser()
    while ¬(D[i] ∧ D[i + 1]) :
        Wait.
    Verrouiller m.
    D[i] ← faux.
    D[i + 1] ← faux.
    Déverrouiller m.
    manger()
    D[i] ← vrai.
    D[i + 1] ← vrai.
```

L'exclusion mutuelle n'est pas respectée si un philosophe constate que ses baguettes sont disponibles et que l'un des booléens soit modifié le temps de verrouiller le mutex.

(c) On propose :

```
philosophe(i)
while vrai :
    penser()
    Verrouiller M[i].
    Verrouiller M[i + 1].
    manger()
    Déverrouiller M[i].
    Déverrouiller M[i + 1].
```

Ici il y a bien exclusion mutuelle à nouveau grâce aux mutex par baguette, mais il y a un risque d'interblocage (donc de famine), par exemple si chaque philosophe s'empare de sa baguette de gauche.

3. L'idée ici est que lorsqu'un philosophe souhaite manger, il s'empare de sa baguette de gauche, puis celle de droite, sauf le dernier qui fait l'inverse.

```
philosophe(i)
while vrai :
    penser()
    if i < 4 :
        Verrouiller M[i].
        Verrouiller M[i + 1].
    else
        Verrouiller M[0].
        Verrouiller M[4].
    manger()
    Déverrouiller M[i].
    Déverrouiller M[i + 1].
```

Si on suppose qu'il y a interblocage, alors cela signifie que chaque philosophe possède une baguette et une seule (sinon l'un des philosophes pourrait s'emparer d'une deuxième baguette et manger). Or cela n'est pas possible, car alors soit tous les philosophes possèdent leur baguette de gauche ou de droite.

4. Pour éviter l'interblocage, on peut également limiter le nombre de philosophes qui ont le droit de s'emparer de leur première baguette à 4. Pour un argument similaire au précédent, cela donne bien une solution correcte.

```
philosophe(i)
while vrai :
    penser()
    wait s.
    Verrouiller M[i].
    Verrouiller M[i + 1].
    manger()
    Déverrouiller M[i].
    Déverrouiller M[i + 1].
    post s.
```

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.

> Solution :
>
> 1. Absence de famine : Supposons que le thread 0 soit dans la boucle `while want[1 - i]`. Considérons la prochaine action du thread 1 :
>   - Supposons que le thread 1 est dans la boucle `while want[1 - i]`. Si `turn` vaut 0 alors le thread 1 rentre dans le `if`, met `want[1]` à `false` reste dans `while turn != 1`. Comme `want[1 - i]` est faux, le thread 0 sort de la boucle et entre dans la section critique. Si `turn` vaut 1 alors c'est le thread 1 qui va entrer dans le section critique puis finalement appeller `unlock(1)` qui permet au thread 0 de rentrer dans la section critique.
>   - Si le thread 1 appelle `unlock(1)` alors il met `turn` à 0 et `want[1]` à `false`. Le thread 0 peut alors rentrer dans la section critique.
> 2. Exclusion mutuelle : Supposons que les deux thread soient dans la section critique. On peut supposer sans perte de généralité que le thread 0 est entré en premier. À cet instant, `want[0]` vaut donc `true` et le thread 1 ne peut pas entrer dans la section critique tant que le thread 0 n'a pas appelé `unlock` pour mettre `want[0]` à `false`.

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) ?

Solution :

1. (a) Dans cette première solution, il peut y avoir interblocage : comme seul le dernier fil à terminer libère le sémaphore, seul un fil pourra l'attendre avec succès (car le sémaphore est initialisé à 0).

   (b) Dans cette deuxième solution, il y a à nouveau interblocage, cette fois-ci dès le premier fil à terminer : puisque le compteur de la barrière ne vaudra pas 0, le compteur du sémaphore vaudra toujours 0 au moment d'attendre le sémaphore. Cette attente se fera donc indéfiniment, et le mutex ne sera jamais déverrouillé.

2. On propose :

```
Fonction Rendez_vous(B)
  Verrouiller B.m.
  B.compteur ← B.compteur − 1.
  if B.compteur = 0 :
    Post B.s.
  Déverrouiller B.m.
  Wait B.s.
  Post B.s.
```

   Ici, il n'était pas nécessaire de verrouiller le mutex lorsqu'on attend le sémaphore. Tant que le dernier fil n'a pas terminé, tous les fils qui ont terminé doivent attendre le sémaphore qui n'a jamais été libéré. Dès lors que le dernier fil termine, il libère le sémaphore, et chaque fil peut donc passer le tourniquet et laisser la place au suivant. À la fin du rendez-vous, le sémaphore aura un compteur qui vaudra 1, donc n'est pas réutilisable.

3. On peut s'en sortir en utilisant deux tourniquets. On fait d'abord passer les fils par un même tourniquet qui n'est débloqué que lorsque le dernier fil a terminé (et il bloque alors le deuxième tourniquet). On fait ensuite la même chose en inversant les rôles.

Fonction `Créer`(n)
    return $\{\text{total} = n, \text{compteur} = n, m = \texttt{Mutex()}, s_1 = \texttt{Semaphore(0)}, s_2 = \texttt{Semaphore(1)}\}$
Fonction `Rendez_vous`(B)
    Verrouiller $B.m$.
    B.compteur $\leftarrow B$.compteur $- 1$.
    if B.compteur $= 0$ **:**
        Wait $B.s_2$.
        Post $B.s_1$.
    Déverrouiller $B.m$.
    Wait $B.s_1$.
    Post $B.s_1$.

    Verrouiller $B.m$.
    B.compteur $\leftarrow B$.compteur $+ 1$.
    if B.compteur $= B$.total **:**
        Wait $B.s_1$.
        Post $B.s_2$.
    Déverrouiller $B.m$.
    Wait $B.s_2$.
    Post $B.s_2$.