

## I Calcul de somme

Écrire une fonction OCaml `somme` (`t : int array`) (`n : int`) permettant de calculer la somme des éléments de `t` en utilisant  $n$  threads.

On utilisera les fonctions suivantes :

---

```

Mutex.create : unit -> Mutex.t
Mutex.lock : Mutex.t -> unit
Mutex.unlock : Mutex.t -> unit
Thread.create : ('a -> 'b) -> 'a -> Thread.t
Thread.join : Thread.t -> unit

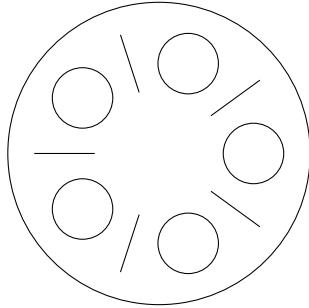
```

---

## II Problème du dîner des philosophes

$n$  philosophes sont assis autour d'une table ronde, avec une baguette entre chaque assiette. Pour manger, un philosophe doit utiliser les deux baguettes adjacentes, qu'il repose ensuite. On souhaite faire manger les philosophes en parallèle avec, si possible, les conditions suivantes :

- Exclusion mutuelle : la même baguette ne peut pas être utilisée par deux philosophes en même temps.
- Absence d'interblocage : il n'est pas possible que tous les philosophes restent bloqués.
- Parallélisme : si deux philosophes ne sont pas voisins, ils peuvent manger en même temps.

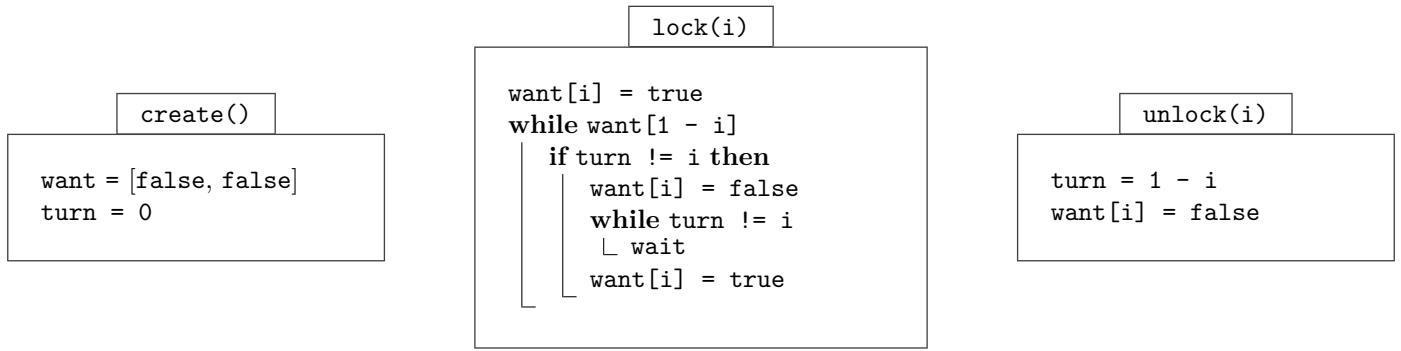


On suppose les philosophes numérotés de 0 à  $n - 1$ , et que le philosophe d'indice  $i$  doit utiliser les baguettes d'indice  $i$  (à sa gauche) et  $i + 1$  (à sa droite). Chaque philosophe est un thread pouvant appeler les fonctions `manger()`, `prendre(i)` et `poser(i)`.

Remarque : ce problème classique apparaît lorsque plusieurs processus d'un même système d'exploitation doivent accéder à des ressources partagées. Par exemple, deux processus souhaitent accéder à la fois à une imprimante et à un scanner risquent de se bloquer mutuellement.

1. Quel est le nombre maximum de philosophes qui peuvent manger simultanément ?
2. Pour chacune des solutions suivantes, écrire une fonction `p(i)` qui décrit le comportement du  $i$ -ème philosophe, et indiquer quelles contraintes sont respectées.
  - (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 baguettes  $i$  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).
  - (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 proposition, proposer une solution correcte.
4. En utilisant un sémaphore dans la troisième proposition, proposer une autre solution correcte.

### III Algorithme de Dekker



Algorithme de Dekker

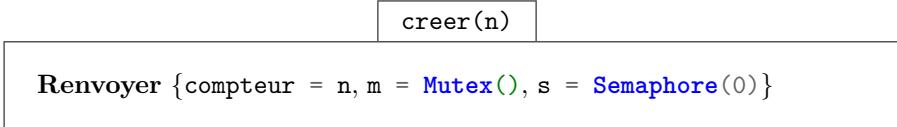
Montrer que l'algorithme de Dekker permet d'implémenter un mutex pour deux threads, c'est-à-dire qu'il respecte l'absence de famine et l'exclusion mutuelle.

### IV Barrière de synchronisation

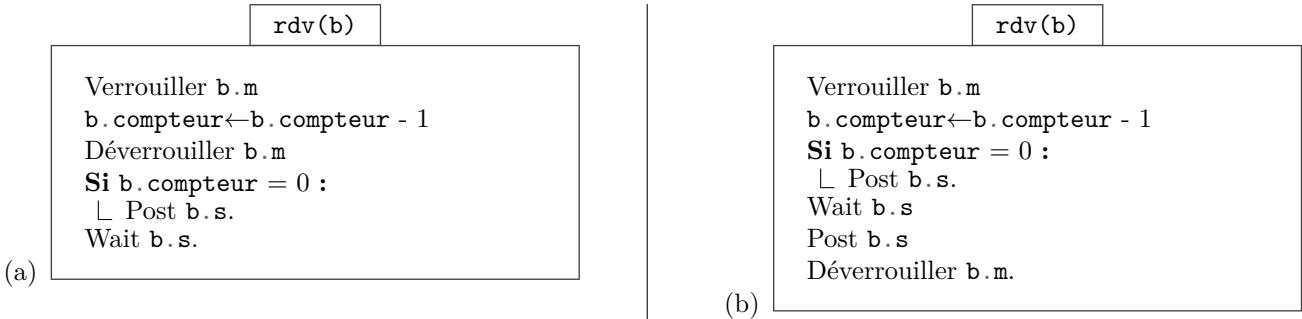
Une barrière de synchronisation, ou rendez-vous est un objet de synchronisation permettant à des fils d'exécution de se synchroniser au cours de leur exécution (donc pas une jointure, qui permet de se synchroniser après l'exécution). On veut pouvoir réaliser les opérations suivantes :

- **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 :



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



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) ?