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
```

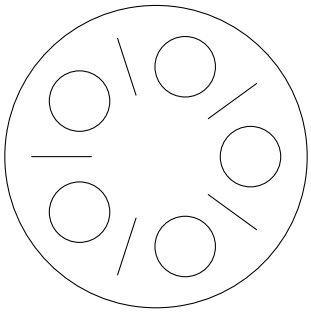
Solution : On peut découper le tableau en n parties, où n est le nombre de threads que l'on souhaite créer. Chaque thread calcule la somme de sa partie du tableau, et on utilise un mutex pour protéger la variable qui stocke la somme totale.

```
let somme (t : int array) (n : int) : int =
  let m = Mutex.create () in
  let s = ref 0 in
  let f i =
    let debut = (Array.length t * i) / n in
    let fin = (Array.length t * (i + 1)) / n in
    let s = ref 0 in
    for j = debut to fin - 1 do
      s := !s + t.(j)
    done;
    Mutex.lock m;
    s := !s + !s;
    Mutex.unlock m
      in
  let threads = Array.init n (fun i -> Thread.create f i) in
  Array.iter Thread.join threads;
  !s
```

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 $\mathtt{p}(i)$ 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 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).

   (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.

Solution :

1. Le nombre maximum de philosophes qui peuvent manger simultanément est $\lfloor \frac{n}{2} \rfloor$ (un sur deux). En effet, il faut au moins $2k$ baguettes pour que k philosophes puissent manger simultanément, et il n'y a que n baguettes.

2. (a)

```
philosophe(i)

Tant que true :
    Verrouiller m
    Prendre(i)
    Prendre(i + 1)
    Manger()
    Poser(i + 1)
    Poser(i)
    Déverrouiller m.
```

   L'exclusion mutuelle et l'absence de famine sont respectées (c'est le principe du mutex). Cependant, il n'y a pas de parallélisme, car un seul philosophe peut manger à la fois.

   (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 ?

   (c)

```
philosophe(i)

Tant que vrai :
    Tant que ¬(D[i] ∧ D[i + 1]) :
        Attendre()
    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.

   (d)

```
philosophe(i)
Tant que vrai :
    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. Pour éviter l'interblocage, on peut faire en sorte que les philosophes d'indice pair commencent par s'emparer de leur baguette de gauche, tandis que les philosophes d'indice impair commencent par s'emparer de leur baguette de droite. Supposons par l'absurde qu'il y ait interblocage et soit i (supposé pair sans perte de généralité) un philosophe ayant pris une baguette. Si la baguette $i+1$ est prise par le philosophe $i+1$, alors celui-ci a ses deux baguettes (car il a dû prendre sa baguette droite avant la gauche). Sinon, i peut prendre deux baguettes.

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 $n-1$.

```
philosophe(i)
Tant que vrai :
    wait s
    Verrouiller M[i]
    Verrouiller M[i + 1]
    Manger()
    Déverrouiller M[i]
    Déverrouiller M[i + 1]
    post s.
```

Preuve : Supposons qu'il y ait interblocage. Alors tous les philosophes ayant passé le sémaphore (au maximum $n-1$) possèdent une baguette et attendent la seconde. Comme il y a n baguettes et au plus $n-1$ philosophes à table, il reste au moins une baguette libre. Cette baguette permet à l'un des philosophes (celui qui l'a à sa droite) de manger, ce qui contredit l'interblocage.

III 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`.

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 :

```
creer(n)
Renvoyer {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
Si b.compteur = 0 :
  Post b.s.
Wait b.s.
```

(b)
```
rdv(b)
Verrouiller b.m
b.compteur←b.compteur - 1
Si 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 $\leftarrow B$.compteur $- 1$
- **Si** 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`)`
- **Renvoyer** $\{\text{total} = n, \text{compteur} = n, m = \texttt{Mutex()}, s_1 = \texttt{Semaphore(0)}, s_2 = \texttt{Semaphore(1)}\}$

Fonction `Rendez_vous(`B`)`
- Verrouiller $B.m$
- B.compteur $\leftarrow B$.compteur $- 1$
- **Si** 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 $+ 1$
- **Si** B.compteur $= B$.total **:**
  - Wait $B.s_1$
  - Post $B.s_2$.
- Déverrouiller $B.m$
- Wait $B.s_2$
- Post $B.s_2$