# Feuille d'exercices n°10 - Programmation concurrente

**Notions abordées**

- Généralisation de l'algorithme de Peterson
- Exemples de calculs en parallèle
- Problème du rendez-vous
- Problème du barbier endormi

## Exercice 1 : Entrelacements

On suppose qu'on dispose de plusieurs compteurs qui affichent le compte dans leur langue :

- un compteur numérique qui compte 1,2,3,...
- un compteur latin qui compte a,b,c,d...
- un compteur grec qui compte $\alpha$, $\beta$,$\gamma$...
- un aviateur qui compte alpha, bravo, charlie, delta, echo ..

On peut lancer ces compteurs en parallèle, en leur demandant de compter jusqu'à une certaine valeur. Chacun suit son compte, à son rythme, mais on n'a aucune hypothèse sur leurs rythmes relatifs.

**Q. 1** Quel affichage est-on susceptible d'observer si on lance latin$(2)$ et grec$(3)$ ?

**Q. 2** Quel affichage est-on susceptible d'observer si on lance num$(2)$, latin$(2)$ et aviateur$(1)$ ?

**Q. 3** Combien d'affichages sont possibles si on lance num$(2)$, latin$(2)$, grec$(2)$ et aviateur$(2)$ ?

**Q. 4** Donner quelques affichages possibles si on lance latin$(5)$ et grec$(4)$, mais qu'on leur demande de s'attendre après avoir affiché 2 dans leur langue ? Donner le nombre d'affichages possibles.

## Exercice 2 : Généralisation de l'algorithme de Peterson à $N$ fils

Dans cet exercice on cherche à généraliser l'algorithme de Peterson vu en cours pour 2 fils d'exécution, à un algorithme qui implémente un verrou pouvant être partagé par $N \in \mathbb{N}^*$ fils d'exécution.

**Q. 1** Rappeler les conditions qu'un verrou doit vérifier.

**Q. 2** Pour chacune des définitions des fonctions lock et unlock ci-après, dire si elles forment une implémentation convenable du type abstrait VERROU. Si ce n'est pas le cas, dire quelle propriété de ce type abstrait n'est pas vérifiée et donner une exécution le montrant.

**Algorithme 1 :** Proposition 1

```
N ← nb. max. de fils partageant le verrou
Turn ← 0
Want ← tableau de N booléens init. à F

Procedure Lock(Turn, Want, N, id) :
    Want[id] ← V
    Turn ← id + 1 mod N
    tant que Turn ≠ id et Want[Turn] faire
        Rien

Procedure Unlock(Want, id) :
    Want[id] ← F
```

**Algorithme 2 :** Proposition 2

```
N ← nb max. de fils partageant le verrou
Turn ← 0
Want ← tableau de N booléens init. à F

Procedure Lock(Turn, Want, N, id) :
    Want[id] ← V
    Turn ← Turn + 1 mod N
    tant que Turn ≠ id faire
        si Want[Turn] = F alors
            Turn ← Turn + 1 mod N

Procedure Unlock(Want, id) :
    Want[id] ← F
```

**Algorithme 3 :** Algorithme de Dijkstra

```
1  N ← nb. max. de fils partageant le verrou
2  turn ← 0
3  Want ← tableau indexé par ⟦0, N⟦ de booléens initialisés à F
4  Ready ← tableau indexé par ⟦0, N⟦ de booléens initialisés à F

5  Procedure Lock(turn, Want, Ready, N, id) :
6      Want[id] ← V
7      feuVert ← F
8      tant que non feuVert faire
9          tant que turn≠id faire
10             Ready[id] ← F
11             si non Want[turn] alors
12                 turn← id
13         Ready[id] ← V
14         feuVert ← V
15         pour p de 0 à N − 1 faire
16             si p ≠ id et Ready[p] alors
17                 feuVert ← F

18 Procedure Unlock(Want, id) :
19     Want[id] ← F
20     Ready[id] ← F
```

**Algorithme 4 :** Algorithme des niveaux

```
1  N ← nb. max. de fils partageant le verrou (⩾ 2)
2  Niveau ← tableau indexé par ⟦0, N⟦ d'entiers initialisés à −1
3  DernierArrive ← tableau indexé par ⟦0, N⟦ d'entiers initialisés à −1

4  Procedure EnTete(Niveau, DernierArrive, N, id, j) :
5      res ← T
6      pour p de 0 à N − 1 faire
7          si p ≠ id et Niveau[p] ⩾ j alors
8              res ← F
9      retourner res

10 Procedure Lock(Niveau, DernierArrive, N, id) :
11     pour j de 0 à N − 2 faire
12         Niveau[id] ← j
13         DernierArrive[j] ← id
14         tant que DernierArrive[j] =id et non EnTete (....) faire
15             Rien

16 Procedure Unlock(Want, id) :
17     Niveau[id] ← −1
```

# Exercice 3 : Rendez-vous à l'aide de sémaphores

Dans cet exercice, on se penche sur des problèmes de rendez-vous entre plusieurs fils d'exécution. La solution proposée diffère selon le nombre de fils d'exécution qui doivent se coordonner et le nombre de fois qu'ils vont se donner rendez-vous.

## 1. Deux fils d'exécution se rencontrent une fois

On suppose qu'on souhaite synchroniser deux fils d'exécution $P_1$ et $P_2$ entre l'exécution de leurs premières instructions, rassemblées sous la notation $A$, et le reste de leurs instructions, rassemblées sous la notation $B$. Autrement dit on a la situation suivante.

| Algo. du fil d'exécution $P_1$ | Algo. du fil d'exécution $P_2$ |
|---|---|
| 1 $A_1$ ; | 1 $A_2$ ; |
| 2 RDV avec $P_2$ ; | 2 RDV avec $P_1$ ; |
| 3 $B_1$ ; | 3 $B_2$ ; |

**Q. 1** Proposer une solution qui permette de synchroniser deux fils d'exécution une fois à l'aide de sémaphores. Comment cette solution se généralise à trois fils d'exécution ? à $N$ fils d'exécution ?

## 2. Plusieurs fils d'exécution se rencontrent une fois

Dans cette section on suppose qu'on a $N$ fils d'exécution $(P_i)_{i\in[\![1,N]\!]}$ à synchroniser. Comme précédemment on note $A_i$ (resp. $B_i$) les instructions que $P_i$ doit réaliser avant (resp. après) le rendez-vous.

Pour réaliser cette synchronisation, on crée un objet qu'on appellera une **barrière**. Cette barrière $\mathcal{B}$ est connue des $N$ fils d'exécution, qui peuvent alors la solliciter pour passer à travers la fonction appel_barriere $(\mathcal{B})$, comme les piétons appellent le feu vert avant de traverser une route. Lorsqu'un fil d'exécution $P_i$ appelle cette fonction, c'est qu'il est au rendez-vous : il a fini $A_i$ et il attend les autres fils d'exécution pour passer la barrière et faire $B_i$ après.

**Q. 2** Proposer une implémentation en pseudo-code d'un tel objet barrière et de la fonction appel_barriere. On pourra utiliser verrous et sémaphores. On veillera aussi à expliciter comment un tel objet doit être initialisé dans une fonction cree_barriere.

### 3. Plusieurs fils d'exécution se rencontrent plusieurs fois

Dans la section précédente, on a proposé une solution pour que $N \in \mathbb{N}^*$ se rencontrent une fois. Cependant on peut aisément imaginer des algorithmes où une famille de fils d'exécution doit se réunir à chaque tour de boucle.

**Q. 3** Que dire de la solution qui consiste à créer, avant de lancer les fils d'exécution, une barrière `b` comme proposé ci-avant, et de demander à chaque fil d'exécution d'attendre les autres à chaque tour en appelant à `attend_barriere b` ? Est-elle robuste ? Plus précisément, cette solution convient-elle si les fils d'exécution ne s'exécutent pas tous autant de fois ?

**Q. 4** Proposer une amélioration de l'implémentation de barrière proposée ci-avant qui soit robuste à la disparition de certains fils d'exécution initialement rattachés à la barrière. On pourra supposer qu'il s'agit plus de départ volontaire que de disparition, et que les fils d'exécution se signalent avant de disparaître.

## Exercice 4 : Producteurs consommateurs en mémoire non bornée

L'hypothèse d'une mémoire infinie en machine n'est pas très raisonnable, en revanche, travailler sans tenir compte de la finitude de la mémoire est, dans certains cas, une attitude opportune. En effet, si on peut déterminer un volume maximal de données que vont produire nos fils d'exécution producteurs, on peut à l'avance allouer un espace mémoire suffisant. Dès lors, on n'a plus à se préoccuper d'éventuels problèmes de dépassement de capacité, mais uniquement du problème que peuvent rencontrer les fils d'exécution consommateurs, à savoir l'absence de données à consommer. On s'intéresse dans cet exercice à plusieurs solutions au problème de producteurs/consommateurs, où l'on entend par solution la donnée de l'algorithme que suit le producteur, de celui que suit le consommateur ainsi que des opérations d'initialisations nécessaires.

### 1. Version allégée de la solution vue en cours

**Q. 1** Dans la solution vue en cours et reproduite ci-dessous, quel objet avait pour rôle d'éviter les problèmes de dépassement de capacité ? Simplifier alors ce modèle avec l'hypothèse de mémoire non bornée.

**Algorithme 7 :** Algorithme des producteurs/consommateurs - version 1

1 Créer un verrou Accès
2 Créer un sémaphore Vide initialisé à $0$
3 Créer un sémaphore Plein initialisé à $n$

```
1 Procedure Production :
2     Génération d'une donnée d ;
3     wait (Plein) ;
4     lock(Accès) ;
5     Écriture de d ;
6     unlock(Accès) ;
7     post (Vide) ;
```

```
1 Procedure Consommation :
2     wait (Vide) ;
3     lock(Accès) ;
4     Lecture d'une donnée d ;
5     unlock(Accès) ;
6     post (Plein) ;
7     Traitement de d ;
```

Dans la suite de cet exercice on suppose qu'il y a un seul consommateur et on cherche à diminuer le nombre de sollicitations du sémaphore. Pour cela on se munit d'une variable $k$, partagée entre tous les fils d'exécution, consommateurs et producteurs, indiquant combien de données sont disponibles en mémoire.

**Q. 2** Quelle précaution doit-on prendre avec la manipulation d'une telle variable ?

**Q. 3** Modifier la solution obtenue à la question 1 afin de faire de maintenir la variable $k$ égale au nombre de données disponibles en mémoire.

**Q. 4** Pour quelle valeur de $k$ un fil d'exécution consommateur risque de rencontrer un problème d'absence de données à consommer ?

Dans tout le reste de cet exercice, on suppose qu'il n'y a plus qu'un seul fil d'exécution consommateur.

**Q. 5** Modifier la solution obtenue à la question 3 afin de faire moins d'appels au sémaphore grâce à la variable $k$.

## 2. Modèle du barbier endormi

On s'intéresse dans cette partie à une nouvelle solution plus adaptée au cas où producteur et consommateurs seraient presque synchrones c'est-à-dire que la production d'une donnée ou son traitement sont de durées comparables et grandes devant les durées de gestion de la mémoire : lecture, écriture, mise à jour des variables et appels au verrou et aux sémaphores. Dans ce cas de figure, on peut donc imaginer qu'une donnée produite est très vite consommée. La solution précédemment proposée risque alors de n'être pas tellement profitable au sens où il risque d'y avoir beaucoup de sollicitations de sémaphore : il arrive souvent que le consommateur attende car la mémoire est vide, et souvent le producteur range sa production dans une mémoire vide et doit donc annoncer qu'elle ne l'est plus. On cherche donc une solution qui solliciterait moins les sémaphores.
Afin d'illustrer les deux solutions pour mieux les comparer, on utilise l'analogie suivante, dite du barbier endormi (*sleeping barber*). Un barbier attend qu'il y ait des clients pour les traiter, et ne peut traiter qu'un client à la fois. Lorsqu'il n'a pas de client, le barbier dort. Son salon est constitué de deux pièces : une salle d'attente, munie d'un banc, et une salle de rasage, munie d'un fauteuil de barbier. On peut entrer de la rue dans la salle d'attente, passer de la salle d'attente à la salle de rasage, et de la salle de rasage ressortir dans la rue.

**Q. 6** En voyant le barbier comme un consommateur et les clients comme des données à consommer, décrire en français le comportement du barbier et des clients pour la solution obtenue à la question 5.

On propose, toujours avec cette analogie, une autre solution décrite par les comportements suivants.
**Comportement du barbier** : Lorsque le barbier a fini avec un client, il le fait sortir et se précipite alors dans la salle d'attente, et une fois qu'il y est il se demande s'il y a des clients. S'il n'y a personne d'autre que lui, il dort sur le banc de la salle d'attente. S'il y a un client, il le fait entrer dans la salle de rasage et le traite.

**Comportement d'un client** : Le client entre dans la salle d'attente et regarde s'il y est seul. S'il est seul, il attend dans la salle. Si le barbier dormait sur le banc, il le réveille.

**Q. 7** Donner le pseudo code du modèle correspondant à ces comportements. On autorisera $k$ à descendre jusqu'à la valeur -1. Expliquer en quoi ce modèle sollicite moins le sémaphore.