Td n° 17

Concurrence

Hugo Salou MPI*

Dernière mise à jour le 29 mars 2023

1 Addition binaire en parallèle

2 Producteurs–consommateurs en mémoire non bornée

2.1 Version allégée de la solution vue en cours

1. C'est le sémaphore Plein qui a pour rôle d'éviter les problèmes de dépassement. Avec l'hypothèse d'une mémoire non bornée, on retire l'appel `acquire` Plein de la procédure Production, l'appel `release` Plein de la procédure Consommation, et la création du sémaphore Plein de l'algorithme 1.
2. L'utilisation d'une telle variable sur plusieurs fils d'exécution nécessite un *mutex* afin que l'écriture en mémoire ne soit faite qu'un fil à la fois.
3.

3 Rendez-vous à l'aide de sémaphores

3.1 Deux fils d'exécutions se rencontrent une fois

1. On considère les fils P_1 et P_2. Dans le programme principal, on crée deux sémaphores $\mathcal{S}_1$ et $\mathcal{S}_2$, initialisés à 0.

Algorithme 1 Fil d'exécution P_1

```
1: A_1
2: release S_2
3: acquire S_1
4: B_1
```

Algorithme 2 Fil d'exécution P_2

```
1: A_2
2: release S_1
3: acquire S_2
4: B_2
```

Pour généraliser, on considère 3 sémaphores $\mathcal{S}_1$, $\mathcal{S}_2$ et $\mathcal{S}_3$ initialement à 0. On définit donc les fils P_1, P_2 et P_3 définis ci-dessous.

Algorithme 3 Fil d'exécution P_1

```
1: A_1
2: release S_2
3: release S_3
4: acquire S_1
5: acquire S_1
6: B_1
```

Algorithme 4 Fil d'exécution P_2

```
1: A_2
2: release S_1
3: release S_3
4: acquire S_2
5: acquire S_2
6: B_2
```

Algorithme 5 Fil d'exécution P_3

```
1: A_3
2: release S_1
3: release S_2
4: acquire S_3
5: acquire S_3
6: B_3
```

2. On perds l'indépendance des fils P_1 et P_2. Pour généraliser la solution, on considère les algorithmes ci-dessous.

Algorithme 6 Fil P_1

```
1: A_1
```

Algorithme 8 Fil P_2

```
1: A_2
```

Algorithme 10 Fil P_3

```
1: A_3
```

Algorithme 7 Fil P'_1

```
1: B_1
```

Algorithme 9 Fil P'_2

```
1: B_2
```

Algorithme 11 Fil P'_3

```
1: B_3
```

Algorithme 12 Programme principal

```
1: lancer P_1, P_2 et P_3 en parallèle
2: attendre la fin de P_1, P_2 et P_3
3: lancer P'_1, P'_2 et P'_3 en parallèle
```

3. Non, les instructions B_2 et A_1 pourraient être exécutées en parallèle, mais ce n'est pas possible avec la subdivision des fils.

3.2 Plusieurs fils se rencontrent une fois

4. On considère l'algorithme ci-dessous.

Algorithme 13 Implémentation de la structure barrière $\mathcal{b}$

```
1:  Procédure CréeBarrière
2:      Soit un verrou 𝒱.
3:      Soit i = 0.
4:      Soit un sémaphore 𝒮 initialisé à 0.
5:      retourner 𝒷 = (𝒱, 𝒮, i).
6:  Procédure AppelBarrière(b)
7:      (𝒱, 𝒮, i) ← 𝒷
8:      lock(𝒱)
9:      i ← i + 1
10:     si i = n alors
11:         pour k ∈ ⟦1, n⟧ faire
12:             release 𝒮
13:     unlock(𝒱)
14:     acquire 𝒮
```

3.3 Plusieurs fils d'exécution se rencontrent plusieurs fois

5.

6. On considère l'algorithme suivant.

Algorithme 14 Implémentation de la structure barrière robuste $\mathcal{b}$

```
1:  Procédure CréeBarrière(n)
2:      Soit un verrou 𝒱.
3:      Soit i = 0, et soit nb = i.
4:      Soit un sémaphore 𝒮 initialisé à 0.
5:      retourner 𝒷 = (𝒱, 𝒮, i, nb).
6:  Procédure AppelBarrière(b)
7:      (𝒱, 𝒮, i) ← 𝒷
8:      lock(𝒱)
9:      i ← i + 1
10:     si i = n alors
11:         pour k ∈ ⟦1, n⟧ faire
12:             release 𝒮
13:     unlock(𝒱)
14:     acquire 𝒮
15: Procédure JeNeViendraisPlus(𝒷)
16:     (𝒱, 𝒮, i, nb) ← 𝒷
17:     lock(𝒱)
18:     nb ← nb − 1
19:     si i = nb alors
20:         pour k ∈ ⟦1, nb⟧ faire
21:             release 𝒮
22:         i ← 0
23:     unlock(𝒱)
```