

Programmation Concurrente

Sémaphores

Exercice 1

On désire réaliser un « rendez-vous » pour deux threads.

Soit le thread A effectuant les opérations a1 puis a2 et le thread B effectuant les opérations b1 puis b2.

Thread A
a1
a2

Thread B
b1
b2

Comment garantir que l'opération a1 se produira toujours avant b2 et que b1 se produira toujours avant a2 ?

Solution

Initialisation :

```
s1 = semaphore(0)
s2 = semaphore(0)
```

Thread A
a1
s1.post()
s2.wait()
a2

Thread B
b1
s2.post()
s1.wait()
b2

Exercice 2

Soit un programme concurrent comportant 42 threads. La routine d'exécution de chaque thread exécute aléatoirement une fonction F à un ou plusieurs moments durant son exécution.

On aimerait toutefois garantir que seulement 7 threads au maximum exécuteront la fonction F à un instant donné, les autres threads devant être bloqués.

Comment garantir ce comportement à l'aide de sémaphore(s) ?

Solution

Initialisation :

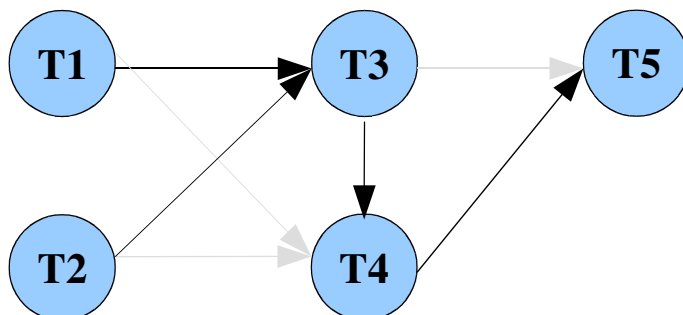
```
lock = semaphore(7)
```

Dans chaque thread:

```
lock.wait()  
F()  
lock.post()
```

Exercice 3

Soit 5 threads dont le graphe d'exécution est défini ci-dessous :



T3 ne s'exécutera que lorsque T1 et T2 seront terminés. T5 s'exécutera lorsque T3 et T4 seront terminés, etc.

Donnez le pseudo-code permettant de réaliser l'ordre d'exécution donné par le graphe ci-dessus à l'aide de sémaphores. Remplissez la phase d'initialisation ainsi que les opérations à effectuer dans chaque thread. L'instruction `work` dans chaque thread symbolise le travail effectué par le thread.

Solution

Le nombre de dépendances peut être réduit comme illustré ci-dessus : dès lors 3 sémaphores suffisent pour les assurer, **à condition que les tâches ne soient exécutées qu'une seule fois !**

Init :

```
S1 = S2 = S3 = semaphore(0)
```

T1 :

```
work  
S1.post
```

T2 :
work
S1.post

T3 :
S1.wait
S1.wait
work
S2.post

T4 :
S2.wait
work
S3.post

T5 :
S3.wait
work

Exercice 4

Solution

a)

```
T1:      sem_wait(S1)
         changement contexte
T2:      sem_wait(S2)
         changement contexte
T1:      sem_wait(S2)
         ==> T1 bloqué
T2:      sem_wait(S1)
         ==> T2 bloqué
```

b)

Il faut rendre indivisible l'acquisition des deux sémaphores pour éviter que les processus commencent à acquérir des ressources sans les obtenir toutes et rester mutuellement bloqués. Cela peut être réalisé, avec un sémaphore initialisé à 1.

```
mutex = sem_init(1)
```

```
T1:      ...
         sem_wait(mutex);
```

```

sem_wait(S1);
sem_wait(S2);
sem_post(mutex);
...

```

T2:

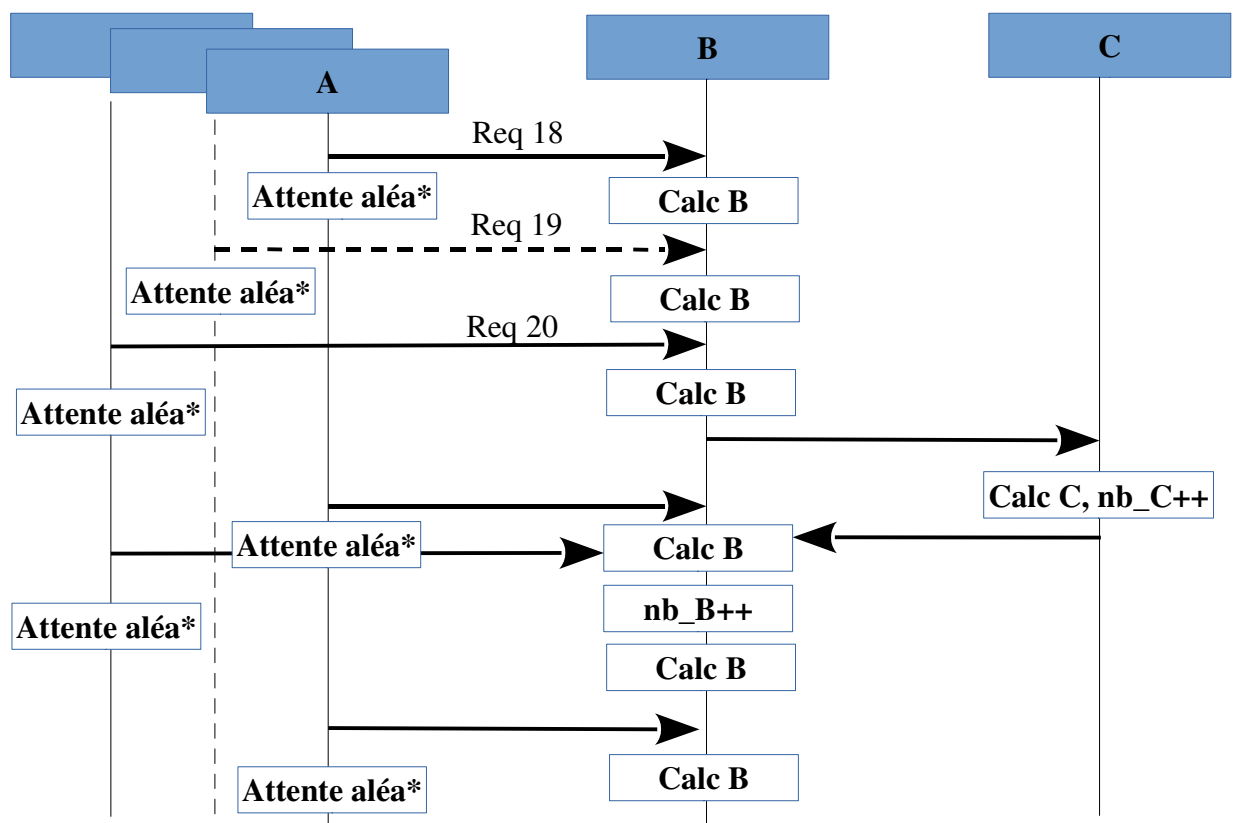
```

...
sem_wait(mutex);
sem_wait(S2);
sem_wait(S1);
sem_post(mutex);
...

```

Exercice 5

MSC possible :



* + test de fin par comptage de requêtes, protégé par un mutex

Solution : cf. `ex_sem.c`

Exercice 6

Solution : cf. `barrier.c`