

# 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 ?

### Exercice 2

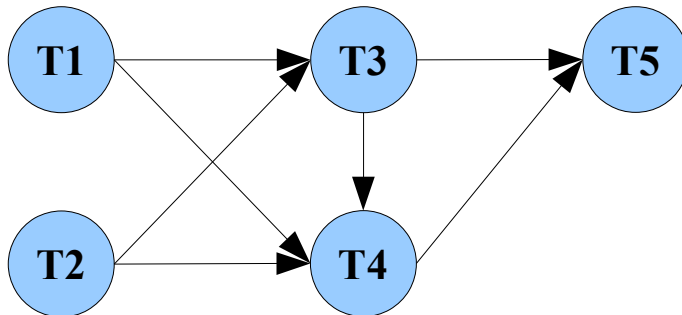
Soit un programme concurrent comportant 42 threads. La routine d'exécution de chaque thread exécute aléatoirement une fonction F à un ou plusieurs moments durant son exécution.

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) ?

### Exercice 3

Soit cinq 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.

Initialisation :

T1 :

T2 :

T3 :

T4 :

T5 :

## Exercice 4

Soit les threads T1 et T2 dont le pseudo-code est donné ci-dessous.

Ces deux threads utilisent de façon concurrente deux ressources R1 et R2, dont l'utilisation est protégée par deux sémaphores S1 et S2.

```
S1 = sem_init(1)
S2 = sem_init(1)

T1 {
    while (true) {
        sem_wait(S1)
        sem_wait(S2)
        // Utilisation de R1 et R2
        ...
        sem_post(S1)
        // Utilisation de R2
        ...
        sem_post(S2)
        // Travail n'utilisant ni R1, ni R2
        ...
    }
}

T2 {
    while (true) {
        sem_wait(S2)
        sem_wait(S1)
        // Utilisation de R1 et R2
        ...
        sem_post(S2)
        // Utilisation de R1
        ...
        sem_post(S1)
        // Travail n'utilisant ni R1, ni R2
        ...
    }
}
```

- a) Démontrez que cette implémentation présente un risque de *deadlock*.
- b) Proposez une modification du code permettant de supprimer ce *deadlock* potentiel à l'aide d'un unique sémaphore supplémentaire. Votre solution ne doit pas changer les instructions existantes, ni leur ordre : limitez-vous à l'insertion des instructions portant sur le nouveau sémaphore.

## Exercice 5

Un système est composé de 20 clients (A) qui envoient des requêtes à un serveur (B), qui lui-même a besoin des services d'un pilote externe (C). Dès que le serveur a collecté 20 requêtes, il ordonne au pilote externe d'effectuer un calcul (calcC) et attend que ce dernier aie fini ce calcul avant de continuer. A, B et C sont matérialisés par 22 threads.

Les 20 threads A doivent effectuer les opérations suivantes en boucle:

- envoyer une requête au serveur B
- attendre un temps aléatoire entre 0 et 20 ms
- tester si 200 requêtes ont été envoyées et sortir si c'est le cas

Le thread B doit effectuer les opérations suivantes en boucle 10 fois:

- chaque fois qu'une requête arrive, simule un calcul par une attente de 1 ms
- chaque fois que 20 requêtes sont collectées, avertir C de faire son travail
- attendre le retour annonçant que C a terminé son travail

Le thread C doit effectuer les opérations suivantes en boucle 10 fois:

- attendre une requête de B
- simuler calcC par une attente de 2 ms
- avertir B que le travail est terminé

1. Produisez un MSC représentatif d'un scénario possible
2. Codez ce système en utilisant des sémaphores (et mutex si nécessaire)

## Exercice 6

On désire implémenter notre propre version de barrière de synchronisation pour N threads à l'aide de **sémaphore(s)**. Votre implémentation de barrière devra avoir un comportement similaire aux barrières POSIX vu en cours, à l'exception de la réinitialisation de la barrière une fois tous les threads passés. Chaque thread doit signaler son arrivée à la barrière à l'aide d'un appel à une fonction dédiée, tout comme pour l'implémentation dans la librairie Pthreads.

On se propose d'implémenter l'interface ci-dessous :

- Un nouveau type barrière :  
`barrier_t`
- Une fonction d'initialisation d'un objet barrière :  
`void barrier_init(barrier_t *b, int count)`
- Une fonction d'attente à la barrière :  
`void barrier_wait(barrier_t *b)`
- Une fonction de destruction de la barrière :  
`void barrier_destroy(barrier_t *b)`

**Attention** : l'implémentation ne doit pas contenir de boucles !

Réalisez un programme de test permettant de montrer que votre implémentation fonctionne correctement.