# Synchronisation

**Exercice 1. Questions de cours** Les questions de cours sont à destinées à vous permettre de vérifier votre compréhension du cours. Elles sont à travailler à l'avance et ne seront pas traitées en TD ou TP.

1. Qu'est-ce qu'une section critique ?
2. Quelles sont les propriétés attendues d'une section critique ?
3. Quels sont les inconvénients du Mutex ?
4. Quels sont les inconvénients des sémaphores ?

**Exercice 2. Création de threads** La creation de threads se fait à l'aide de l'appel système `pthread_create`. Contrairement à l'appel système `fork`, il prend en argument une fonction qui sera exécutée dans un thread séparé et un éventuel argument à donner à cette fonction. Il est possible de configurer certains paramètres du thread créé via un autre argument (*attr*) que nous n'utiliserons pas ici et que l'on va donc toujours laisser à `NULL`.

Avant de commencer, lisez la documentation des appels systèmes suivant : `pthread_create` et `pthread_join`.

Afin d'utiliser les appels systèmes de manipulation des threads, il est nécessaire d'inclure le fichier d'en-tête `pthread.h`. Sur certains systèmes, au moment de la compilation il peut aussi être nécessaire d'ajouter l'option `-l pthreads`.

**1.** Écrivez en C un programme qui crée 10 threads et attend la fin de leur exécution. Chacun de ces threads affichera "Bonjour" avant de se terminer.

**Correction:** *On a ici la forme la plus simple d'utilisation des threads. À la création des threads on donne juste la fonction à exécuter et celle-ci réalise l'affichage demandé. Attention de bien commencer par créer tous les threads avant d'attendre qu'ils se terminent. Faire une seule boucle qui successivement lance un thread et attend sa fin semble marcher mais les threads ne seront pas exécutés en parallèle.*

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void *work(void *ud) {
5          printf("Bonjour\n");
6          return NULL;
7  }
8
9  int main(void) {
10         pthread_t T[10];
11         for (int i = 0; i < 10; i++)
12                 pthread_create(&T[i], NULL, work, NULL);
13         for (int i = 0; i < 10; i++)
14                 pthread_join(T[i], NULL);
15         return EXIT_SUCCESS;
16 }
```

**2.** Modifiez votre programme de manière à passer à chacun des threads un numéro qui l'identifie. Chaque thread devra afficher le numéro qu'il a reçu.

**Correction:** *Il suffit de passer le numéro en paramètre, pour cela il est nécessaire d'utiliser un tableau de numéros, on ne peut utiliser une simple variable dans la boucle `for` car sa durée de vie est insufisante. Le passage de l'argument est légèrement compliqué par le fait que la fonction exécutée par le thread prend en paramètre un entier.*

```
1  void *work(void *ud) {
2          int id = *(int *)ud;
```

```
3         printf("%d\n", id);
4         return NULL;
5 }
6
7 int main(void) {
8         pthread_t T[10];
9         int       D[10];
10        for (int i = 0; i < 10; i++) {
11                D[i] = i;
12                pthread_create(&T[i], NULL, work, &D[i]);
13        }
14        for (int i = 0; i < 10; i++)
15                pthread_join(T[i], NULL);
16        return EXIT_SUCCESS;
17 }
```

**Exercice 3. Recherche parallèle** Dans l'exercice 4 du td 2, nous avons implémenté la recherche en parallèle d'une valeur dans un tableau à l'aide de processus. Pour une tâche aussi simple, l'utilisation de processus est en général très inefficace car les processus sont lourds à créer, les threads sont plus adaptés.

**1.** Reprenez le code que vous avez écris pour la première question de l'exercice 4 du td 2 et modifiez le pour que le tableau soit alloué dans une variable globale et que la recherche soit faite dans une fonction séparée.

**Correction:** *Rien de particulier ici, le code est très similaire à celui produit précédement. Cette version est juste plus adaptée à être parallèlisée avec des threads.*

```
1 #define N 1000
2 int *T;
3
4 void work() {
5         for (int i = 0; i < N; i++) {
6                 usleep(10000);
7                 if (T[i] == 0)
8                         printf("0 trouvé à l'index %d\n", i);
9         }
10 }
11
12 int main(void) {
13        T = malloc(sizeof(int) * N);
14        for (int i = 0; i < N; i++)
15                T[i] = rand() % 100;
16    work();
17        return EXIT_SUCCESS;
18 }
```

**2.** Parallèliser ce code en utilisant 2 threads. Vous devez passer à chaque threads un identifiant qui lui permette de calculer la partie du tableau dans laquelle il doit effectuer la recherche.

**Correction:** *Il faut adapter la fonction qui effectue la recherche de manière à ce qu'elle puisse être lancée dans un thread et qu'elle effectue la recherche sur une seule partie du tableau.*

```
1 void *work(void *ud) {
2         int deb, fin;
3         if (*(int *)ud == 0) {
4                 deb = 0; fin = N / 2;
5         } else {
6                 deb = N / 2; fin = N;
7         }
8         for (int i = deb; i < fin; i++) {
9                 usleep(10000);
10                if (T[i] == 0)
11                        printf("0 trouvé à l'index %d\n", i);
12        }
```

```
13          return NULL;
14  }
```

*Ensuite on remplace l'appel à cette fonction par la création de 2 threads. Cette fois-ci on peut utiliser de simples variables car il n'y a que deux threads, mais un tableau comme dans l'exercice précédent est aussi possible.*

```
1           pthread_t T1, T2;
2           int       D1 = 0, D2 = 1;
3           pthread_create(&T1, NULL, work, &D1);
4           pthread_create(&T2, NULL, work, &D2);
5           pthread_join(T1, NULL);
6           pthread_join(T2, NULL);
```

**Exercice 4. Synchronisation simple** Comme on l'a vu en cours, si les threads on l'avantage de rendre le partage de données très simple et efficace, il est nécessaire de faire extrêmemnt attention car il n'offrent aucune protections automatique lors d'accès concurent à une même donnée.

**1.** Écrivez un programme avec une variable entière globale qui lance 10 threads et attend la fin de leur exécution avant d'afficher la valeur de cette variable. Chaque thread exécutera simplement une boucle qui incrémente dix mille fois la variable. Lancer votre programme plusieurs fois afin d'observer le résultat.

*Attention, le code que nous écrivons ici est très naïf et le compilateur risque de réaliser des optimisations qui vont artificiellement cacher l'effet que l'on souhaite mettre en évidence. Afin d'éviter ce problème, il est possible de désactiver les optimisations sur la variable globale en la déclarant* `volatile` *:*

```
volatile int count = 0 ;
```

**Correction:** *Le code ici ne pose pas de soucis, il n'y a rien de plus que dans les exercices précédent. Le résultat attendu est 100000 mais en pratique on observe dans la majorité des cas une valeur inferieure et variable car les différents threads entre en conflits lorsqu'il incrémentent la variable.*

```
1  volatile int count = 0;
2
3  void *work(void *ud) {
4      for (int i = 0; i < 10000; i++)
5          count++;
6      return NULL;
7  }
8
9  int main(void) {
10     pthread_t T[10];
11     for (int i = 0; i < 10; i++)
12         pthread_create(&T[i], NULL, work, NULL);
13     for (int i = 0; i < 10; i++)
14         pthread_join(T[i], NULL);
15     printf("%d\n", count);
16     return EXIT_SUCCESS;
17 }
```

**2.** Le probleme de synchronisation se trouve au niveau de l'incrémentation du compteur, il est nécessaire de la protéger afin qu'un seul thread à la fois puisse accéder à cette variable. Modifiez votre code manière à utiliser une exclusion mutuelle pour protéger ces accès lorsque cela est nécessaire.

Les appels systèmes principaux nécessaires pour l'utilisation des exclusions nmutuelles en C sont : `pthread_mutex_lock`, `pthread_mutex_unlock` et `pthread_mutex_init`, ainsi que la constante `PTHREAD_MUTEX_INITIALIZER`.

**Correction:** *La synchronisation demandée ici nécessite que l'on crée un* `pthread_mutex_t` *accessible par tous les threads. Cela peut ce faire de deux manières, soit comme ici en déclarant une variable globale et en l'initialisant à l'aide de la constante* `PTHREAD_MUTEX_INITIALIZER`*, soit en l'initialisant dans la fonction main à l'aide de* `pthread_mutex_init` *ce qui permet d'éventuellement configurer différentes options.*

*Il reste ensuite juste à encadrer l'incrémentation du compteur par un vérouillage et un déverouillage de la section critique.*

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *work(void *ud) {
    for (int i = 0; i < 10000; i++) {
        pthread_mutex_lock(&mutex);
        count++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

**Exercice 5. Lecteurs-écrivains** Le problème des lecteurs-écrivains est un problème classique de synchronisation. On considère une base de données et un ensemble de threads qui souhaitent lire (les *lecteurs*) et écrire (les *écrivains*) dans cette base. Chaque thread attend un temps aléatoire puis demande à acquérir la base pour lire ou pour écrire. Le travail sur la base dure un certain temps puis le thread libère la base. On souhaite réunir les conditions suivantes :

**C1.** Plusieurs lecteurs peuvent accéder à la base en lecture en même temps ;

**C2.** Un seul écrivain peut accéder à la base à la fois ;

**C3.** Un écrivain ne peut pas accéder à la base en même temps que des lecteurs.

Chaque lecture ou écriture dans la base prend un temps variable, pour la suite de cet exercice, on les simulera à l'aide des deux fonctions suivantes :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void lire() {
    printf("lire debut\n");
    usleep(rand() % 500000);
    printf("lire fin\n");
}
void ecrire() {
    printf("ecrire debut\n");
    usleep(rand() % 500000);
    printf("ecrire fin\n");
}
```

Vous testerez votre code en lancant à chaque fois plusieurs threads en parallèle. Vous êtes encouragés à expérimenter avec différentes quantités de threads et comparer les résultats.

**1.** La fonction suivante va aléatoirement réaliser des lectures et des écritures dans la base. Si elle est exécutée par plusieurs threads en même temps, les conditions C1, C2 et C3 sont-elles satisfaites ?

```c
void *run(void *arg) {
    while (1) {
        if ((rand() % 2) == 0)
            lire();
        else
            ecrire();
    }
}
```

**Correction:** *Seule la condition C1 est satisfaite, C2 et C3 ne le sont pas. Un thread peut entrer dans* `ecrire` *à un moment ou un autre thread est en train de lire ou d'écrire dans la base.*

**2.** Proposez une première solution utilisant une section critique satisfaisant les conditions C2 et C3. (dans cette première version un seul lecteur est autorisé à accéder à la base en même temps)

**Correction:** *Il suffit ici d'encadrer les appels à lire et à écrire dans une section critique. Attention, la variable* `mutex` *doit être globale afin d'être partagée par toutes les threads.*

```c
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void *run(void *arg) {
    while (1) {
        if ((rand() % 2) == 0) {
            pthread_mutex_lock(&mutex);
```

```
7             lire();
8             pthread_mutex_unlock(&mutex);
9         } else {
10            pthread_mutex_lock(&mutex);
11            ecrire();
12            pthread_mutex_unlock(&mutex);
13        }
14    }
15 }
```

*Les questions suivantes de cet exercice sont difficiles. Trouver une solution qui marche correctement dans tous les cas est difficile et il est encore plus difficile de le prouver. Vous êtes encouragés à travailler ces questions afin de proposer des solutions puis d'étudier leurs corrections afin de comprendre comment les résoudre correctement. La synchronisation de threads est un sujet difficile que l'on ne peut maitriser que par la pratique et l'experience.*

**3.** Proposez maintenant une solution au problème respectant les trois conditions. (Un seul vérou ne sera pas suffisant ici)

**Correction:** *Il va falloir plus de code cette fois-ci donc pour simplifier les choses on modifie la fonction* `run` *pour quelle appelle quatre fonctions chargées de gérer la synchronisation.*

```
1 void run(void) {
2     while (1) {
3         if ((rand() % 2) == 0) {
4             PrendreVerouLecteur();
5             lire();
6             LibererVerouLecteur();
7         } else {
8             PrendreVerouEcrivain();
9             ecrire();
10            LibererVerouEcrivain();
11        }
12    }
13 }
```

*Un seul vérou ne va pas suffir, deux seront nécessaires. Le premier à le même rôle que dans la première version, pour accéder à la base il est nécessaire qu'il soit vérouillé. Pour l'écriture, il est juste nécessaire de vérouiller et déverouiller ce vérou, les fonctions sont donc très simples.*

```
1 pthread_mutex_t base = PTHREAD_MUTEX_INITIALIZER;
2 void PrendreVerrouEcrivain() {
3     pthread_mutex_lock(&base);
4 }
5 void LibererVerrouEcrivain() {
6     pthread_mutex_unlock(&base);
7 }
```

*Pour la lecture maintenant, il est aussi nécessaire que le vérou* `base` *soit vérouillé afin d'empêcher une écriture pendant une lecture et inversement. Par contre, si une lecture est déjà en cours, une nouvelle lecture peut commencer sans avoir besoin d'obtenir le vérou, le premier thread l'ayant déjà obtenu, il faut par contre faire attention de ne le déverouiller que lorsque toutes les lectures sont finies. Pour cela, on compte le nombre de lecture en cours, la première à commencer se charge de vérouiller la base, et la dernière à terminer se charge de la déverouiller. Cette procedure de comptage doit être elle-même protégée par un vérou afin d'éviter les problèmes de synchronisation.*

```
1 pthread_mutex_t lecture = PTHREAD_MUTEX_INITIALIZER;
2 int nbLecteurs = 0;
3 void PrendreVerrouLecteur() {
4     pthread_mutex_lock(&lecteur);
5     nbLecteurs++;
6     if (nbLecteurs == 1)
7         pthread_mutex_lock(&base);
8     pthread_mutex_unlock(&lecteur);
9 }
```

```
10 void LibererVerrouLecteur() {
11     pthread_mutex_lock(&lecteur);}
12     nbLecteurs--;}
13     if (nbLecteurs == 0)}
14         pthread_mutex_unlock(&base);}
15     pthread_mutex_unlock(&lecteur);}
16 }
```

**4.** Votre solution prévient-elle du risque de famine ? Proposez si besoin une solution à ce problème. *(on parle de famine si un thread peu se retrouver à attendre un temps infini alors que les autres threads progressent.)*

**Correction:** *Il y a un risque de famine, en effet un thread désirant faire une écriture peut être bloqué indéfiniment si des threads voulant faire des lecture arrivent en permanence et que donc le nombre de lecteur ne tombe jamais à zéro.*

*Une solution simple à ce problème est de rajouter un troisième vérou autour des fonctions* `prendre*`. *Si un thread veut démarer une écriture, elle va le vérouiller et aucune nouvelle lecture ne pourra commencer tant que les lectures en cours n'aurons pas toutes terminées afin de permettre à l'écriture d'être réalisée.*

```
1  pthread_mutex_t ecriture = PTHREAD_MUTEX_INITIALIZER;
2  void PrendreVerrouLecteur() {
3      pthread_mutex_lock(&ecrivain);
4      pthread_mutex_lock(&lecteur);
5      nbLecteurs++;
6      if (nbLecteurs == 1)}
7          pthread_mutex_lock(&base);
8      pthread_mutex_unlock(&lecteur);
9      pthread_mutex_unlock(&ecrivain);
10 }
11 void PrendreVerrouEcrivain() {
12     pthread_mutex_lock(&ecrivain);
13     pthread_mutex_lock(&base);
14     pthread_mutex_unlock(&ecrivain);
15 }
```