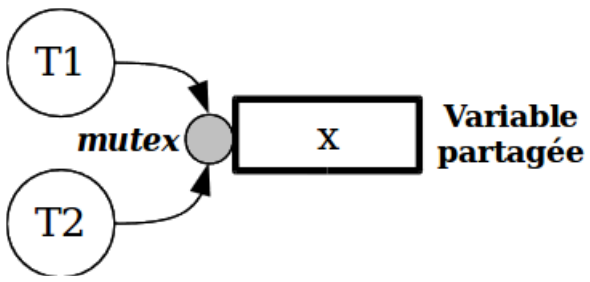


Extrait du référentiel : BTS Systèmes Numériques option A (Informatique et Réseaux)		Niveau(x)
S6. Systèmes d'exploitation S6.2. S.E. Multitâches professionnelles	Processus lourds / légers, diagramme des états d'une tâche Ordonnancement des processus	3

Objectifs du TD :

- La programmation concurrente
- La synchronisation des données

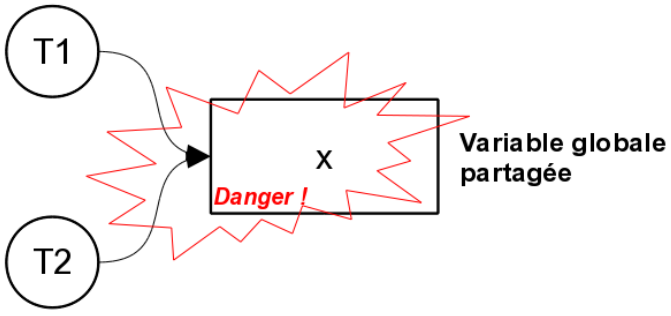
Pré-requis :

- Cours sur les mécanismes de synchronisation

Dans la programmation concurrente (avec des processus lourds ou légers), le terme de **synchronisation** se réfère à deux concepts distincts (mais liés) :

– la synchronisation de tâches
– la synchronisation de données

La synchronisation de données est un mécanisme qui vise à conserver la cohérence entre différentes données dans un environnement multitâche.

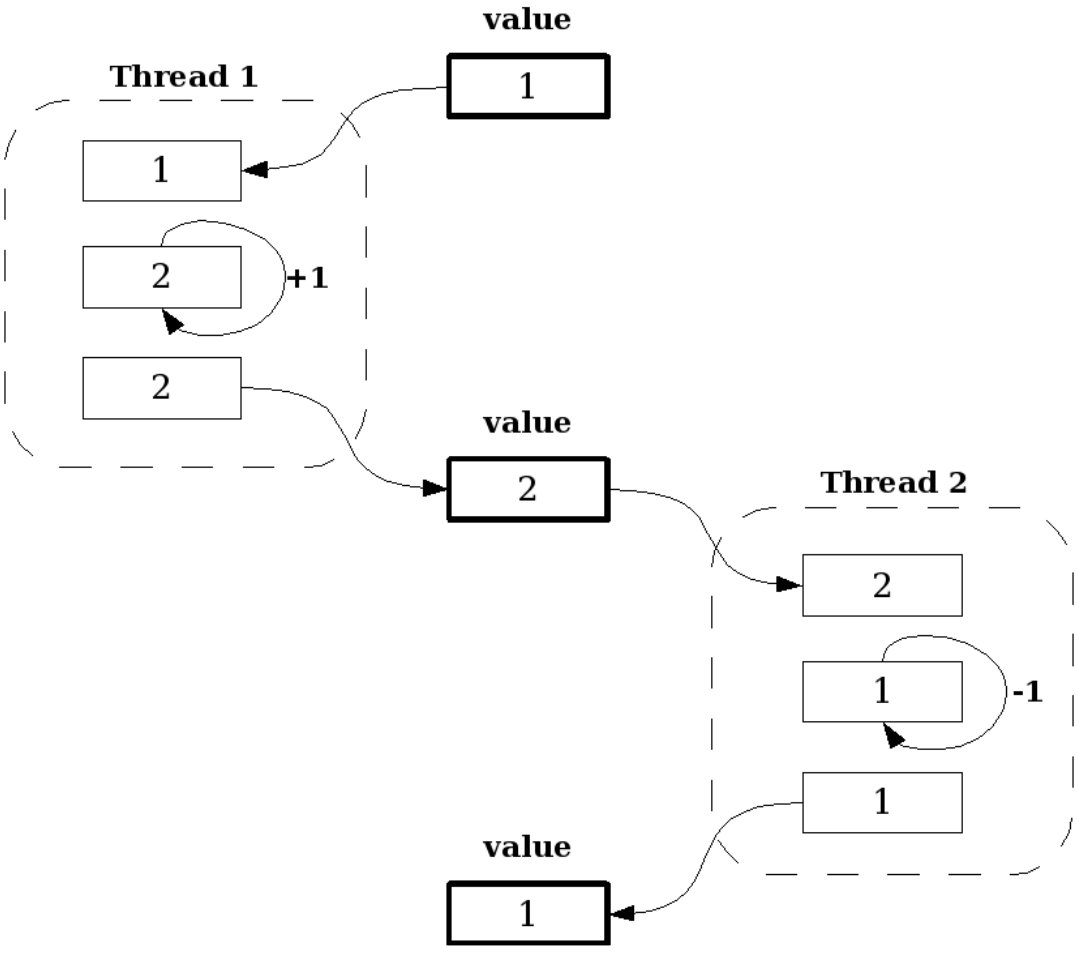


Rappels : les threads sont englobés dans un processus lourd et partagent donc sa mémoire virtuelle. Cela permet aux threads de partager les données globales mais de disposer de leur propre pile pour implanter leurs variables locales.
Des processus lourds, donc séparés et indépendants, qui désirent partager des données devront utiliser un mécanisme fourni par le système pour communiquer tel que IPC (Inter Processus Communication).

PROBLÈME DE SYNCHRONISATION DE DONNÉES

L'objectif de cette activité est de mettre en évidence le problème classique de la synchronisation de données à base de threads.

On va créer deux tâches : une incrémente une variable partagée et l'autre la décrémente.



Un déroulement possible

En réalité, le résultat est imprévisible du fait que vous ne pouvez pas savoir ni prévoir l'ordre d'exécution des instructions.

On va écrire un programme en C qui illustre ce problème. Les deux tâches réalisent le même nombre de traitement (COUNT). On suppose donc que la variable globale (value_globale) doit

revenir à sa valeur initiale (1) puisqu'il y aura le même nombre d'incrémentation et de décrémentation.

Question

Compilez (en CLI) et exécutez le code ci-dessous.
Quel résultat donne l'exécution du programme ? Faîtes une petite conclusion.
Vous pouvez tester plusieurs fois et même avec des valeurs différentes de COUNT.

……..

……..

……..

……..

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>


int value_globale = 1;

// Chaque thread (tache) va faire ses COUNT boucles
#define COUNT  5

// Fonctions correspondant au corps d'un thread (tache)
void *increment(void *inutilise);
void *decrement(void *inutilise);

int main(void)
{
  pthread_t thread1, thread2;

  printf("Avant les threads : value = %d\n", value_globale);

  pthread_create(&thread1, NULL, increment, NULL);
  pthread_create(&thread2, NULL, decrement, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf("Apres les threads : value = %d\n", value_globale);

  printf("Fin du thread principal\n");

  pthread_exit(NULL);

  return EXIT_SUCCESS;
}

void *increment(void *inutilise)
{
   int value;
   int count = 0;
```

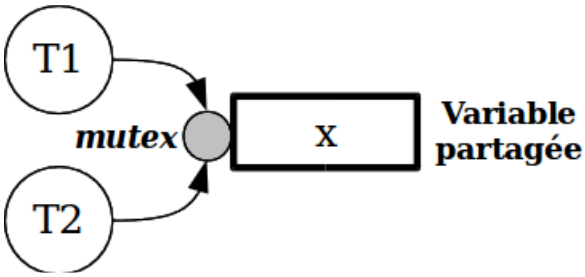
```
    while(1)
    {
        value = value_globale;
        printf("Thread1 : load value (value = %d) ", value);
        value += 1;
        printf("Thread1 : increment value (value = %d) ", value);
        value_globale = value;
        printf("Thread1 : store value (value = %d) ", value_globale);
        count++;

        if(count >= COUNT)
        {
            printf("Le thread1 a fait ses %d boucles\n", count);
            return(NULL);
        }
    }
    return NULL;
}

void *decrement(void *inutilise)
{
    int value;
    int count = 0;

    while(1)
    {
        value = value_globale;
        printf("Thread2 : load value (value = %d) ", value);
        value -= 1;
        printf("Thread2 : decrement value (value = %d) ", value);
        value_globale = value;
        printf("Thread2 : store value (value = %d) ", value_globale);
        count++;

        if(count >= COUNT)
        {
            printf("Le thread2 a fait ses %d boucles\n", count);
            return(NULL);
        }
    }
    return NULL;
}
```

MISE EN ŒUVRE D'UN MUTEX

Pour résoudre ce genre de problème, le système doit permettre au programmeur d'utiliser un **verrou d'exclusion mutuelle**, c'est-à-dire de pouvoir bloquer, en une seule instruction « atomique », toutes les tâches tentant d'accéder à cette donnée, puis, que ces tâches puissent y accéder lorsque la variable est libérée.

Une instruction atomique est une instruction qui ne peut pas être divisée (donc interrompue).

Un mutex est un **objet d'exclusion mutuelle** (**MUT**ual **EX**clusion), et est très pratique pour protéger des données partagées de modifications concurrentes et pour implémenter des **sections critiques**.



Ce qu'il faut retenir :

- un mutex peut être dans deux états : **déverrouillé** ou **verrouillé** (cela signifie qu'il est donc possédé par un thread).
- un mutex est une ressource booléenne car il ne peut être pris que par un seul thread à la fois.
- un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.
- on peut donc faire deux opérations sur un mutex : **verrouiller** (lock) ou **déverrouiller** (unlock).

Il existe parfois l'opération trylock, équivalent à lock, mais qui en cas d'échec ne bloquera pas le thread.

On va écrire maintenant le programme en C qui corrige le problème. Les deux tâches réalisent le même nombre de traitement (COUNT). On aura donc la variable globale (value_globale) qui revient à sa valeur initiale (1) puisqu'il y aura le même nombre d'incrémentation et de décrémentation.

Question

Compilez (en CLI) et exécutez le code ci-dessous.
Quel résultat donne l'exécution du programme ? Faîtes une petite conclusion.

……...

……...

……...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define MUTEX

#ifdef MUTEX
pthread_mutex_t globale_lock = PTHREAD_MUTEX_INITIALIZER;
#endif

int value_globale = 1;

// Chaque thread (tache) va faire ses COUNT boucles
#define COUNT  25
```

```
// Fonctions correspondant au corps d'un thread (tache)
void *increment(void *inutilise);
void *decrement(void *inutilise);
int main(void)
{
  pthread_t thread1, thread2;

  #ifdef MUTEX
  printf("Exemple avec le mutex\n");
  #else
  printf("Exemple sans le mutex\n");
  #endif

  printf("Avant les threads : value = %d\n", value_globale);

  pthread_create(&thread1, NULL, increment, NULL);
  pthread_create(&thread2, NULL, decrement, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf("Apres les threads : value = %d\n", value_globale);

  printf("Fin du thread principal\n");

  pthread_exit(NULL);

  return EXIT_SUCCESS;
}

void *increment(void *inutilise)
{
   int value;
   int count = 0;

   while(1)
   {
      #ifdef MUTEX
      pthread_mutex_lock(&globale_lock);
      #endif
      value = value_globale;
      printf("Thread1 : load value (value = %d) ", value);
      value += 1;
      printf("Thread1 : increment value (value = %d) ", value);
      value_globale = value;
      printf("Thread1 : store value (value = %d) ", value_globale);
      #ifdef MUTEX
      pthread_mutex_unlock(&globale_lock);
      #endif
      count++;

      if(count >= COUNT)
      {
         printf("Le thread1 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}
```

```
void *decrement(void *inutilise)
{
   int value;
   int count = 0;

   while(1)
   {
      #ifdef MUTEX
      pthread_mutex_lock(&globale_lock);
      #endif
      value = value_globale;
      printf("Thread2 : load value (value = %d) ", value);
      value -= 1;
      printf("Thread2 : decrement value (value = %d) ", value);
      value_globale = value;
      printf("Thread2 : store value (value = %d) ", value_globale);
      #ifdef MUTEX
      pthread_mutex_unlock(&globale_lock);
      #endif
      count++;

      if(count >= COUNT)
      {
         printf("Le thread2 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}
```

Il faut lire la page man de pthread_mutexattr_init(3) pour plus d'informations sur les attributs de mutex et leur utilisation.

Les définitions à retenir :

- **Exclusion mutuelle** : une ressource est en exclusion mutuelle si seul un thread peut utiliser la ressource à un instant donné.
- **Section Critique** : c'est une partie de code telle que deux threads ne peuvent s'y trouver au même instant.
- **Chien de garde (watchdog)** : un chien de garde est une technique logicielle utilisée pour s'assurer qu'un programme ne reste pas bloqué à une étape particulière du traitement qu'il effectue. C'est une protection destinée généralement à redémarrer le système, si une action définie n'est pas exécutée dans un délai imparti. Il s'agit d'un compteur qui est régulièrement remis à zéro. Si le compteur dépasse une valeur donnée (timeout) alors on procède à un redémarrage (reset) du système. Si une routine entre dans une boucle infinie, le compteur du chien de garde ne sera plus remis à zéro et un reset est ordonné.