TP Multitâche n°2 : Programmation concurrente

Sommaire

Les objectifs de ce tp sont de découvrir la programmation concurrente à base de *threads*.

Programmation concurrente

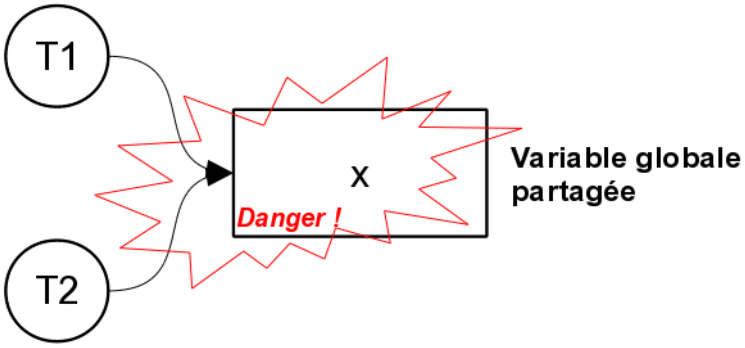
Dans la programmation concurrente (avec des processus lourds ou légers), le terme de **synchronisation** se réfère à deux concepts distincts (mais liés) :

- la synchronisation de tâches
- la **synchronisation de données**

Les problèmes liés à la synchronisation rendent toujours la programmation plus difficile.

Synchronisation de données

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



Rappel : 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*).

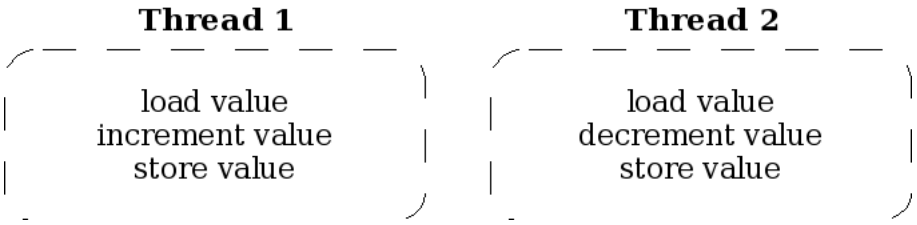
Séquence n°1 : le problème de synchronisation de données

Objectifs

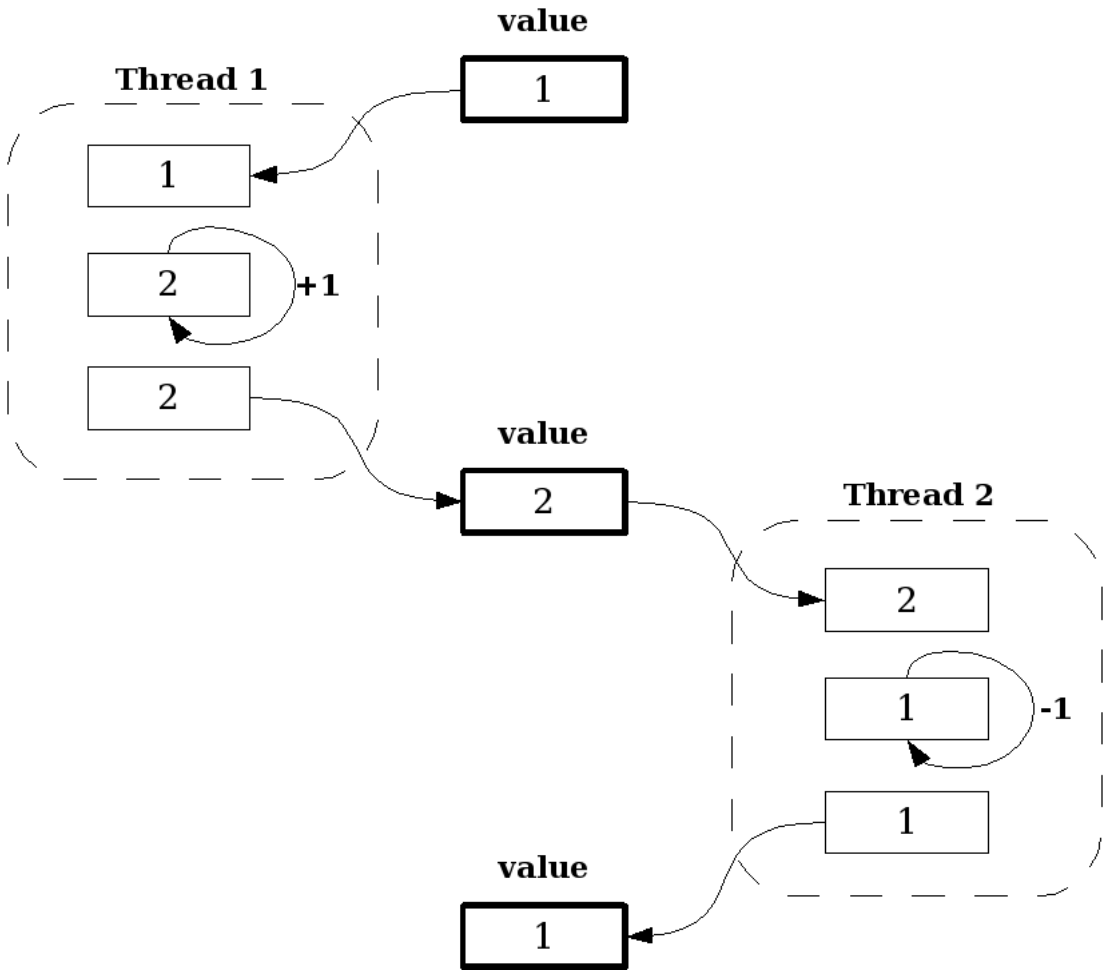
L'objectif de cette séquence est de mettre en évidence le problème classique de la synchronisation de données à base de *threads*.

Étape n°0 : deux tâches qui ne s'entendent pas !

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



Un déroulement possible serait :



En réalité, le résultat n'est pas prévisible, 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.

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

// Variable globale partagée
int value_globale = 1; // valeur initiale

// 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;
}
```

threads.2a.c

L'exécution de ce programme montre que la variable globale ne revient pas à son état initial !

Vous pouvez tester plusieurs fois et même avec des valeurs différentes de `COUNT`.

```
$ ./threads.2a
Avant les threads : value = 1
Thread1 : load value (value = 1) Thread1 : increment value (value = 2) Thread1 : store value
     (value = 2) Thread1 : load value (value = 2) Thread1 : increment value (value = 3)
    Thread1 : store value (value = 3) Thread1 : load value (value = 3) Thread1 : increment
    value (value = 4) Thread1 : store value (value = 4) Thread1 : load value (value = 4)
    Thread1 : increment value (value = 5) Thread1 : store value (value = 5) Thread1 : load
    value (value = 5) Thread1 : increment value (value = 6) Thread1 : store value (value =
    6) Le thread1 a fait ses 5 boucles
Thread2 : load value (value = 2) Thread2 : decrement value (value = 1) Thread2 : store value
     (value = 1) Thread2 : load value (value = 1) Thread2 : decrement value (value = 0)
    Thread2 : store value (value = 0) Thread2 : load value (value = 0) Thread2 : decrement
    value (value = -1) Thread2 : store value (value = -1) Thread2 : load value (value = -1)
    Thread2 : decrement value (value = -2) Thread2 : store value (value = -2) Thread2 : load
     value (value = -2) Thread2 : decrement value (value = -3) Thread2 : store value (value
    = -3) Le thread2 a fait ses 5 boucles
Apres les threads : value = -3
Fin du thread principal
$
```

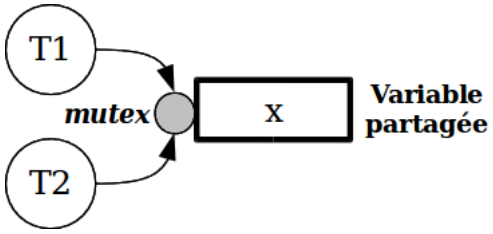
Conclusion : cette application n'assure pas une synchronisation des données entre les deux tâches.

Étape n°1 : mise en oeuvre 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.

Remarque : une instruction atomique est une instruction qui ne peut être divisée (donc interrompue).

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



Ce qu'il faut savoir et retenir :
- Un *mutex* peut êtte 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*).

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

```
// déclaration et initialisation d'un mutex
pthread_mutex_t globale_lock = PTHREAD_MUTEX_INITIALIZER;

int value_globale = 1; // la variable globale partagée

#define COUNT 5
...

pthread_mutex_lock(&globale_lock); // demande de verrouillage du mutex

... // je suis dans une zone d'exclusion mutuelle (section critique)
// je peux donc lire ou écrire dans la variable globale partagée en toute sécurité

pthread_mutex_unlock(&globale_lock); // déverrouillage du mutex
...
```

Exemple d'utilisation d'un mutex

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

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.

```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; // la variable globale partagée

// 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");
  #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++;
      usleep(100000);
      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++;
      usleep(100000);
      if(count >= COUNT)
      {
         printf("Le thread2 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}
```

threads.2b.c

L'exécution de ce programme montre la **synchronisation effectuée grâce au *mutex***.

```
$ ./threads.2b
Exemple avec le mutex
Avant les threads : value = 1
Thread1 : load value (value = 1) Thread1 : increment value (value = 2) Thread1 : store value
     (value = 2) Thread2 : load value (value = 2) Thread2 : decrement value (value = 1)
    Thread2 : store value (value = 1) Thread1 : load value (value = 1) Thread1 : increment
    value (value = 2) Thread1 : store value (value = 2) Thread2 : load value (value = 2)
    Thread2 : decrement value (value = 1) Thread2 : store value (value = 1) Thread1 : load
    value (value = 1) Thread1 : increment value (value = 2) Thread1 : store value (value =
    2) Thread2 : load value (value = 2) Thread2 : decrement value (value = 1) Thread2 :
    store value (value = 1) Thread2 : load value (value = 1) Thread2 : decrement value (
    value = 0) Thread2 : store value (value = 0) Thread1 : load value (value = 0) Thread1 :
    increment value (value = 1) Thread1 : store value (value = 1) Thread2 : load value (
    value = 1) Thread2 : decrement value (value = 0) Thread2 : store value (value = 0)
    Thread1 : load value (value = 0) Thread1 : increment value (value = 1) Thread1
: store value (value = 1) Le thread2 a fait ses 5 boucles
Le thread1 a fait ses 5 boucles
Apres les threads : value = 1
Fin du thread principal
$
```

Bilan

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 en général 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.

Séquence n°2 : le mutex dans tous ses états !

Objectifs

L'objectif de cette séquence est de montrer les **dangers** liés à la synchronisation de données dans une application multi-tâches.

Étape n°1 : aie !

Si on reprend le programme en C précédent, que se passerait-il si un des deux *threads* ne dévérrouille pas le mutex ? Pour tester cela, il vous suffit de mettre en commentaire un des deux appels à `pthread_mutex_unlock(&globale_lock)`. On obtient alors une situation de **blocage** infini entre les deux *threads* :

```
$ ./threads.2b
Exemple avec le mutex
Avant les threads : value = 1
Ctrl-C
$
```

Bilan

Les mécanismes de synchronisation peuvent conduire aux problèmes suivants :
- **Interblocage (*deadlocks*)** : Le phénomène d'interblocage est le problème le plus courant. L'interblocage se produit lorsque deux *threads* concurrents s'attendent mutuellement. Les *threads* bloqués dans cet état le sont définitivement.
- **Famine (*starvation*)** : Un processus léger ne pouvant jamais accéder à un verrou se trouve dans une situation de famine. Cette situation se produit, lorsqu'un processus léger, prêt à être exécuté, est toujours devancé par un autre processus léger plus prioritaire.
- **Endormissement (*dormancy*)** : cas d'un processus léger suspendu qui n'est jamais réveillé.

Synchronisation de tâches

La synchronisation de tâches (ou de processus) est un mécanisme qui vise à bloquer l'exécution des différentes tâches (ou processus) à des points précis de leur programme de manière à ce que tous les processus passent les étapes bloquantes au moment prévu par le programmeur.

On cherche par exemple à empêcher des programmes d'exécuter la même portion de code en même temps, ou au contraire forcer l'exécution de deux parties de code en même temps. Dans la première hypothèse, le processus bloque l'accès au code en avant d'entrer dans la portion de code critique ou si cette section est en train d'être exécutée, se met en attente. Le processus libère l'accès en sortant de la partie du code. Ce mécanisme peut être implémenté de multiples manières.

Séquence n°3 : le problème de synchronisation de tâches

Objectifs

L'objectif de cette séquence est de mettre en évidence le problème de la synchronisation de tâches.

Étape n°0 : deux tâches qui ne s'attendent pas !

On crée deux tâches (*threads*) : une affichera des étoiles '*' et l'autre des dièses '#'.

Les deux tâches réaliseront le même nombre de traitement (`COUNT`) mais la tâche 1 devra d'abord faire "seule" un 1/3 des ses boucles. La tâche 2 ne doit démarrer ses boucles que si la tâche 1 a fait au moins 1/3 de son travail.

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

// Chaque thread (tache) va faire ses COUNT boucles
#define COUNT 50

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

int main(void)
{
  pthread_t thread1, thread2;

  printf("Le thread1 (etoile) doit faire au moins 1/3 de son travail (%d de ses %d boucles)\
     n", (int)(COUNT/3), COUNT);
  printf("Le thread2 (diese) attendra que le thread1 (etoile) ait fait au moins 1/3 de son
     travail avant de démarrer le sien\n");

  pthread_create(&thread1, NULL, etoile, NULL);
  pthread_create(&thread2, NULL, diese, NULL);

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

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

  pthread_exit(NULL);

  return EXIT_SUCCESS;
}

void *etoile(void *inutilise)
{
   char c1 = '*';
   int count = 0;
```



```
   while(1)
   {
      write(1, &c1, 1);// écrit une '*' sur stdout (descripteur 1)
      count++;

      if(count == (int)(COUNT/3))
      {
         //printf("Le thread1 a fait au moins 1/3 (%d) de ses %d boucles\n", count, COUNT);
      }

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

      pthread_yield();
   }
   return NULL;
}

void *diese(void *inutilise)
{
   char c1 = '#';
   int count = 0;

   //printf("Thread2 : attendra que thread1 ait fait une partie de son travail ");
   while(1)
   {
      write(1, &c1, 1);// écrit un '#' sur stdout (descripteur 1)
      count++;

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

      pthread_yield();
   }
   return NULL;
}
```

threads.3a.c

Évidemment aucun mécanisme de synchronisation a été ajouté pour réaliser ce qui a été demandé !

Étape n°1 : mise en oeuvre d'une variable de condition

L'implémentation des threads intègre la notion de **variable de condition** (de type `pthread_cond_t`), qui, associée à une variable normale et à un mutex (ou un sémaphore), va permettre de synchroniser des activités sur les changements de valeur de la variable et les conditions satisfaites par la nouvelle valeur.

Une condition (abréviation pour variable-condition) est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition (un prédicat) soit vérifiée.

Les opérations fondamentales sur les conditions sont :
– signaler la condition (quand le prédicat devient vrai) et attendre la condition
– suspendre la condition jusqu'à ce qu'un autre thread signale la condition

Une variable de condition est associée à un ensemble d'attributs (de type `pthread_condattr_t`). La valeur prédéfinie pour les attributs de variable de condition est `pthread_condattr_default`. Les variables de type `pthread_cond_t` peuvent également être statiquement initialisées, en utilisant la constante `PTHREAD_COND_INITIALIZER`.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

On dispose d'un certains nombres d'appels pour gérer les variables conditions :

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct
    timespec *abstime);
int pthread_cond_destroy(pthread_cond_t *cond);
```

– `pthread_cond_signal` relance l'un des threads attendant la variable condition `cond`. S'il n'existe aucun thread répondant à ce critère, rien ne se produit. Si plusieurs threads attendent sur `cond`, seul l'un d'entre eux sera relancé, mais il est impossible de savoir lequel.
– `pthread_cond_broadcast` relance tous les threads attendant sur la variable condition `cond`. Rien ne se passe s'il n'y a aucun thread attendant sur `cond`.
– `pthread_cond_wait` déverrouille atomiquement le mutex (comme pthread_unlock_mutex) et attend que la variable condition `cond` soit signalée. L'exécution du thread est suspendu et ne consomme pas de temps CPU jusqu'à ce que la variable condition soit signalée. Le mutex doit être verrouillé par le thread appelant à l'entrée de pthread_cond_wait. Avant de rendre la main au thread appelant, pthread_cond_wait reverrouille `mutex` (comme pthread_lock_mutex).
– `pthread_cond_timedwait` déverrouille atomiquement `mutex` et attend sur `cond`, comme le fait `pthread_cond_` cependant l'attente est bornée temporellement. Si `cond` n'a pas été signalée après la période spécifiée par abstime, le mutex `mutex` est reverrouillé et `pthread_cond_timedwait` rend la main avec l'erreur `ETIMEDOUT`.
– `pthread_cond_destroy` détruit une variable condition, libérant les ressources qu'elle possède. Aucun thread ne doit attendre sur la condition à l'entrée de `pthread_cond_destroy`.

Déverrouiller le mutex et suspendre l'exécution sur la variable condition est effectué atomiquement. Donc, si tous les threads verrouillent le mutex avant de signaler la condition, il est garanti que la condition ne peut être signalée (et donc ignorée) entre le moment où un thread verrouille le mutex et le moment où il attend sur la variable condition.

Exemple d'utilisation d'une variable-condition :

```
pthread_mutex_t condition_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition_var = PTHREAD_COND_INITIALIZER;
...
pthread_mutex_lock(&condition_lock);
pthread_cond_wait(&condition_var, &condition_lock);
pthread_mutex_unlock(&condition_lock);
...
pthread_mutex_lock(&condition_lock);
pthread_cond_signal(&condition_var);
pthread_mutex_unlock(&condition_lock);
```

On va écrire maintenant le programme en C qui met en œuvre la synchronisation de tâches. Les deux tâches réalisent le même nombre de traitement (`COUNT`) mais la tâche 1 (qui affiche simplement des étoiles '*') doit d'abord faire "seule" un 1/3 des ses boucles puis le signale à la tâche 2 (qui elle affiche simplement des dièses '#'). La tâche 2 ne démarre ses boucles que si la tâche 1 a fait au moins 1/3 de son travail.

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

pthread_mutex_t condition_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition_var = PTHREAD_COND_INITIALIZER;

// Chaque thread (tache) va faire ses COUNT boucles
#define COUNT 50

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

int main(void)
{
  pthread_t thread1, thread2;

  printf("Le thread1 (etoile) doit faire au moins 1/3 de son travail (%d de ses %d boucles)
      et il le signalera au thread2 (diese)\n", (int)(COUNT/3), COUNT);
  printf("Le thread2 (diese) attendra que le thread1 (etoile) ait fait au moins 1/3 de son
      travail avant de démarrer le sien\n");

  pthread_create(&thread1, NULL, etoile, NULL);
  pthread_create(&thread2, NULL, diese, NULL);

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

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

  pthread_exit(NULL);

  return EXIT_SUCCESS;
}
```



```
void *etoile(void *inutilise)
{
   char c1 = '*';
   int count = 0;

   while(1)
   {
      write(1, &c1, 1);// écrit une '*' sur stdout (descripteur 1)
      count++;

      if(count == (int)(COUNT/3))
      {
         pthread_mutex_lock(&condition_lock);
         //printf("Le thread1 a fait au moins 1/3 (%d) de ses %d boucles et il le signale au
             thread2\n", count, COUNT);
         pthread_cond_signal(&condition_var);
         pthread_mutex_unlock(&condition_lock);
      }

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

      pthread_yield();
   }
   return NULL;
}

void *diese(void *inutilise)
{
   char c1 = '#';
   int count = 0;

   //printf("Thread2 : attendra que thread1 ait fait une partie de son travail ");
   while(1)
   {
      pthread_mutex_lock(&condition_lock);
      if(count == 0)
      {
         pthread_cond_wait(&condition_var, &condition_lock);
      }
      pthread_mutex_unlock(&condition_lock);

      write(1, &c1, 1);// écrit un '#' sur stdout (descripteur 1)
      count++;

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



```
      }

      pthread_yield();
   }
   return NULL;
}
```

threads.3b.c

On obtient une synchronisation des tâches :

```
$ ./threads.3b
Le thread1 (etoile) doit faire au moins 1/3 de son travail (16 de ses 50 boucles) et il le
    signalera au thread2 (diese)
Le thread2 (diese) attendra que le thread1 (etoile) ait fait au moins 1/3 de son travail
    avant de démarrer le sien
**********##########**********##########**********##########**********##########**********
#########
Fin du thread principal
```

Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés des séquences précédentes.

Question 1. Est-ce qu'un *thread* peut accéder aux variables globales de son processus ? Si oui, Quel est le problème que cela engendre ?

Question 2. Qu'est-ce qu'un *mutex* ?

Question 3. Dans quels états peut être un *mutex* ?

Question 4. Qu'est-ce qu'une ressource en exclusion mutuelle ?

Question 5. Qu'est-ce qu'une section critique ?

Question 6. Quels dangers entraînent l'utilisation de mécanismes de synchronisation comme les *mutex* ?

