TP Multitâche n°2 : Programmation concurrente

Sommaire

Programmation concurrente **2**

Synchronisation de données . . . 2

Séquence n°1 : le problème de synchronisation de données . . . 3

Séquence n°2 : le mutex dans tous ses états ! . . . 10

Séquence n°3 : le sémaphore . . . 10

Synchronisation de tâches . . . 12

Séquence n°4 : le problème de synchronisation de tâches . . . 12

Séquence n°5 : le coiffeur endormi . . . 17

Modèles de programmation concurrente **18**

Producteurs/Consommateurs . . . 19

Lecteurs/Rédacteurs . . . 19

Séquence n°6 : modèle répartiteur/travailleurs . . . 20

Séquence n°7 : modèle en groupe . . . 21

Séquence n°8 : modèle en pipeline . . . 21

Questions de révision **22**

Annexe n°1 : la synchronisation sous Windows **23**

Mutex . . . 23

Section Critique . . . 25

Sémaphores . . . 28

Les objectifs de ce tp sont de découvrir la programmation concurrente à base de *threads* sous GNU/Linux. Des exemples pour Windows sont fournis en Annexe.

Programmation concurrente

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

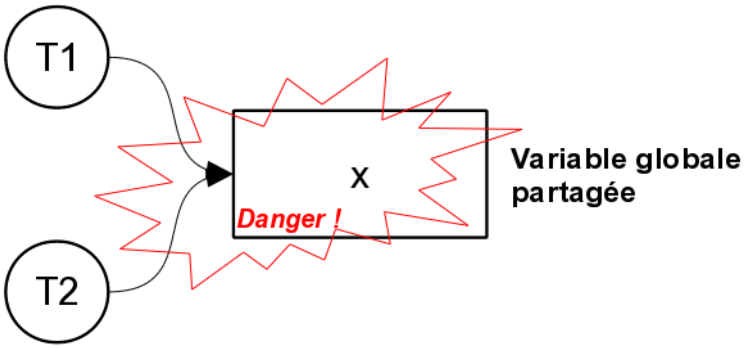
— la synchronisation de données

— la synchronisation de tâches

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.

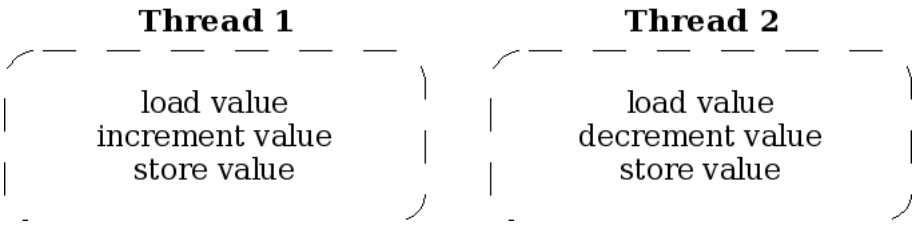
Des processus lourds, donc séparés et indépendants, qui désirent partager des données devront utiliser un mécanisme fourni par le système pour communiquer tel que IPC (*Inter Processus Communication*).

Séquence n°1 : le problème de synchronisation de données

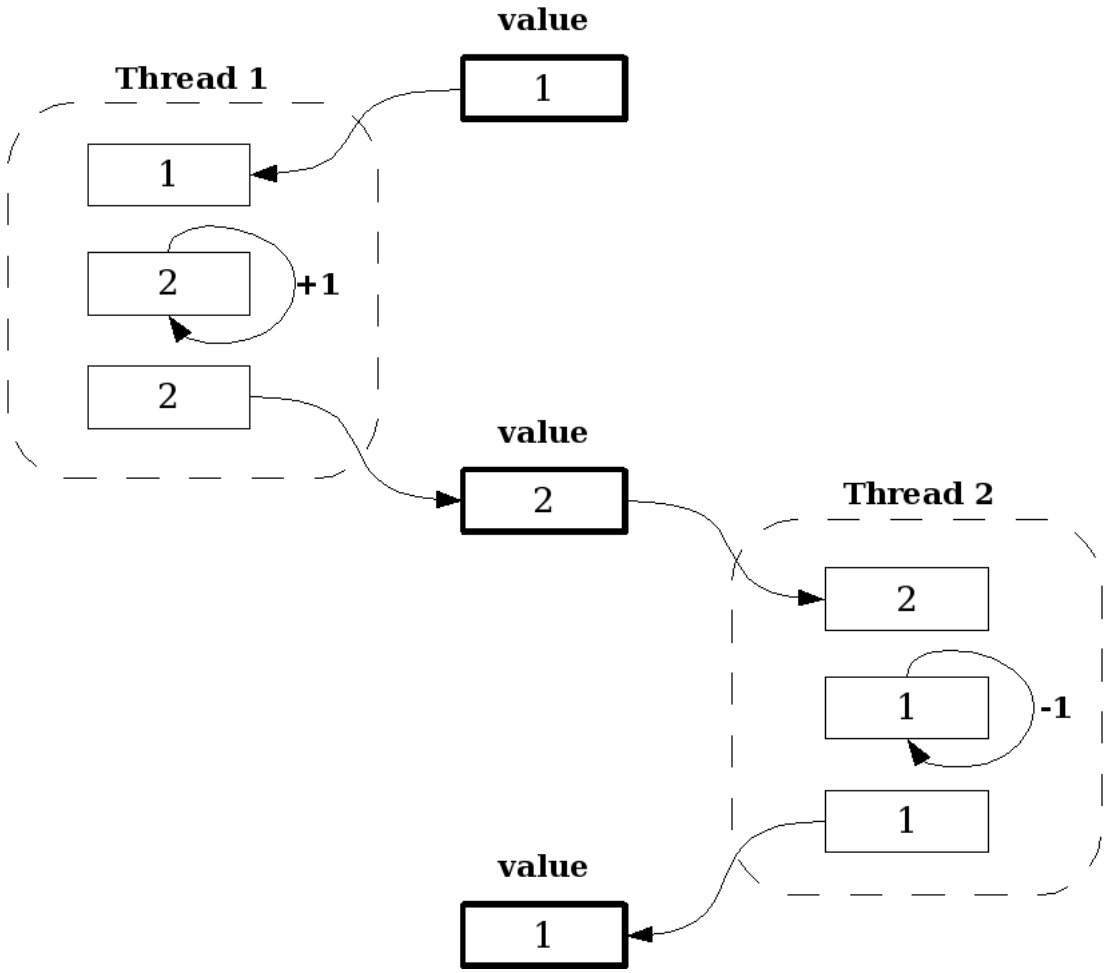
L'objectif de cette séquence est de mettre en évidence le problème classique de la synchronisation de données à base de *threads*.

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");

  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;
}
```



```c
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
$
```

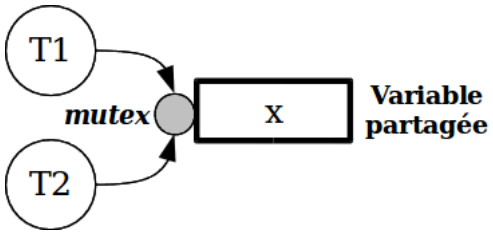
Conclusion : cette application n'assure pas une synchronisation des données entre les deux tâches.

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 ête 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>

pthread_mutex_t globale_lock = PTHREAD_MUTEX_INITIALIZER;

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;

  printf("Exemple avec le mutex\n");
  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");

  return EXIT_SUCCESS;
}

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

   while(1)
   {
      pthread_mutex_lock(&globale_lock);

      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);

      pthread_mutex_unlock(&globale_lock);
```



```
      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)
   {
      pthread_mutex_lock(&globale_lock);

      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);

      pthread_mutex_unlock(&globale_lock);

      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.

Question 1. Créer un programme `threads.2c.c` à partir du programme précédent qui met en évidence la notion de section critique. Pour cela, créer une fonction `sectionCritique()` qui regroupe les instructions où les deux threads ne peuvent se trouver en même temps. Cette fonction recevra en paramètre le numéro de thread qui entre dans la section critique et le type d'opération (incrémentation ou décrémentation) à réaliser.

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

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.

Notion de 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°3 : le sémaphore

L'objectif de cette séquence est de montrer l'utilisation des sémaphores dans une application multi-tâches.

Un sémaphore S est une **variable entière** qui n'est accessible qu'au travers de 3 opérations :

- `Init` pour initialiser la valeur du sémaphore,
- `P` pour l'acquisition (*Proberen*, tester ou P(uis-je)) : en attente jusqu'à ce qu'une ressource soit disponible,

```
SI (S > 0)
  ALORS S--;
  SINON (attendre sur S)
FSI
```

- `V` pour la libération (*Verhogen*, incrémenter ou V(as-y)) : rend simplement une ressource disponible.


```
S++;
SI (des threads sont en attente sur S)
  ALORS laisser l'un deux continuer (reveil)
FSI
```

L'utilisation d'un sémaphore assure que ces opérations sont atomiques (indivisibles), ce qui signifie qu'elles ne peuvent pas être exécutées plusieurs fois de manière concurrente. Un *thread* qui désire exécuter une opération qui est déjà en cours d'exécution par un autre *thread* doit attendre que le premier termine. La valeur d'un sémaphore ne peut jamais devenir négative.

De manière générale, un sémaphore est un mécanisme empêchant deux *threads* ou plus d'accéder simultanément à une ressource partagée. On distingue plusieurs utilisations possibles :

- le **sémaphore général** peut avoir un très grand nombre d'états car il s'agit d'un **compteur** dont la valeur initiale peut être assimilée au nombre de ressources disponibles.
- le **sémaphore binaire**, comme un mutex, n'a que deux états : 0=verrouillé (ou occupé) ou 1=déverrouillé (ou libre). Ceci a pour effet de contrôler l'accès une ressource unique. Le sémaphore binaire permet l'exclusion mutuelle (*mutex*) : une ressource est en exclusion mutuelle si seul un processus peut utiliser la ressource à un instant donné.
- le **sémaphore bloquant** est un sémpahore de synchronisation qui est initialisé à 0=verrouillé (ou occupé). Ceci a pour effet de bloquer n'importe quel *thread* qui effectue P(S) tant qu'un autre *thread* n'aura pas fait un V(S). Cette utilisation des sémaphores permet de réaliser des barrières de synchronisation.

Le sémaphore a été inventé par Edsger Dijkstra.

Les sémaphores POSIX sont disponibles dans la librairie standard C (GNU). Le sémaphore sera de type `sem_t`. La liste des appels disponibles pour gérer un sémaphore `sem_t` est la suivante :

- `sem_init`
- `sem_wait`, `sem_trywait`, sem_timedwait
- `sem_post`
- `sem_getvalue`,
- `sem_open`, `sem_close`, `sem_unlink`, `sem_destroy`

Une présentation des sémaphores POSIX est accessible en faisant : `man sem_overview`

Les opérations de base sur un sémaphore :

```c
#include <semaphore.h>

void Init(sem_t *s, unsigned int value)
{
   sem_init(s, 0, value); // 0 pour un sémaphore partagée entre threads
}

void P(sem_t *s)
{
   sem_wait(s);
}
```

```
void V(sem_t *s)
{
   sem_post(s);
}

int getValue(sem_t *s)
{
   return sem_getvalue(s, &sval);
}
```

Question 2. Créer un programme `threads.2d.c` à partir du programme précédent qui utilisera un sémaphore binaire à la place d'un mutex.

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°4 : le problème de synchronisation de tâches

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");

  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) {
         return(NULL);
      }

      pthread_yield();
   }
   return NULL;
}

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

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

      if(count >= 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_wait`, 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.

```
// threads.3b.c
// Mise en oeuvre des variables conditions

#define _GNU_SOURCE

#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

int count = 0;

// 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 du travail (%d des %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 du
      travail avant de démarrer le sien\n");

  write(1, "[", 1);// écrit un caractère sur stdout (descripteur 1)

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

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

  write(1, "]", 1);// écrit un caractère sur stdout (descripteur 1)

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

  pthread_exit(NULL);

  return EXIT_SUCCESS;
}

void *etoile(void *inutilise)
{
   char c1 = '*';

   while(1)
   {
      pthread_mutex_lock(&condition_lock);

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

      if(count == (int)(COUNT/3))
      {
         write(1, "|", 1);// écrit un caractère sur stdout (descripteur 1)
         pthread_cond_broadcast(&condition_var);
      }

      pthread_mutex_unlock(&condition_lock);

      if(count >= 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);

      while (count < (int)(COUNT/3))
```



```
      {
         pthread_cond_wait(&condition_var, &condition_lock);
      }

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

      pthread_mutex_unlock(&condition_lock);

      if(count >= 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 du travail (16 des 50 boucles) et il le
    signalera au thread2 (diese)
Le thread2 (diese) attendra que le thread1 (etoile) ait fait au moins 1/3 du travail avant
    de démarrer le sien
[****************|*****##############################]
Fin du thread principal
```

Question 3. Créer un programme `threads.3c.c` à partir du programme précédent qui utilisera des sémaphores à la place d'un mutex et d'une variable de condition.

Question 4. Donner le nom donné aux deux sémaphores utilisés.

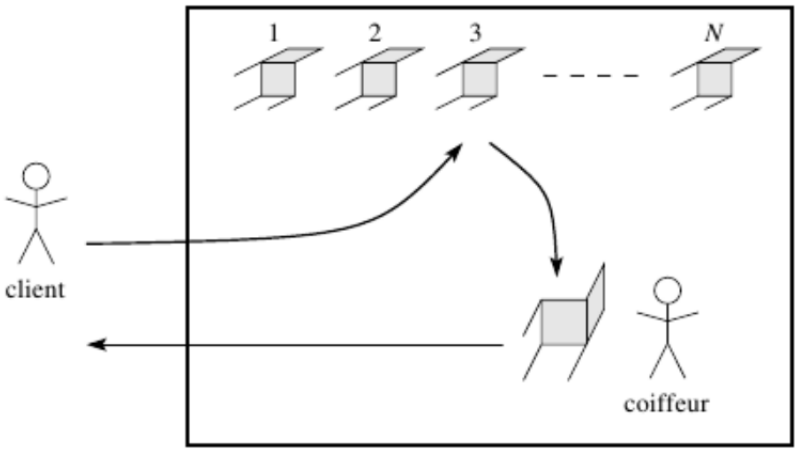
Séquence n°5 : le coiffeur endormi

Il s'agit d'un de ces problèmes de synchronisation mis sous une forme « plaisante ».

On en trouve une application presque directe dans certains mécanismes des systèmes d'exploitation (comme l'ordonnancement des accès disque).

Ce problème classique de synchronisation est le suivant : un coiffeur dispose dans son salon d'un fauteuil de coiffure et de N chaises pour les clients en attente. S'il n'y a pas de clients, le coiffeur dort. À l'arrivée du premier client, il se réveille et s'occupe de lui. Si un nouveau client arrive pendant ce temps, il attend sur l'une des chaises disponibles. Mais si toutes les chaises sont occupées, le client s'en va. Enfin lorsque le coiffeur a fini de couper les cheveux du dernier client, il peut se rendormir.

Le prix d'une coupe est estimé à 20 euros. On suivra en temps réel le montant de la caisse du coiffeur ainsi que son manque à gagner.



Indications :

— utiliser un sémaphore général pour gérer le nombre de places
— utiliser des sémaphores bloquants pour la synchronisation des tâches (coiffeur disponible, coupe terminée, ...)
— utiliser des mutex pour les accès concurrents aux variables globales partagées

Question 5. Compléter le programme `coiffeur-endormi.c`.

Modèles de programmation concurrente

Chaque programme comportant des *threads* est différent. Cependant, certains modèles communs sont apparus.

Ces modèles permettent de définir :

— comment une application attribue une activité à chaque *thread* et
— comment ces *threads* communiquent entre eux.

Pour la communication entre threads, on distingue généralement 2 modèles de synchronisation de ressources :

— Modèle Producteur/Consommateur
— Modèle Lecteurs/Rédacteurs

Pour la gestion des threads, on distingue généralement 3 modèles :

— Modèle répartiteur/travailleurs ou maître/esclaves
— Modèle en groupe
— Modèle en pipeline

Quelque soit le modèle utilisé, les problèmes liés à la synchronisation seront résolus avec les mécanismes déjà vus comme les *mutex*, les variables-conditions, les sémaphores, ...

Producteurs/Consommateurs

Le modèle Producteurs/Consommateurs est un exemple de synchronisation de ressources. Il peut s'envisager dans différents contextes, notamment en environnement multi-thread.

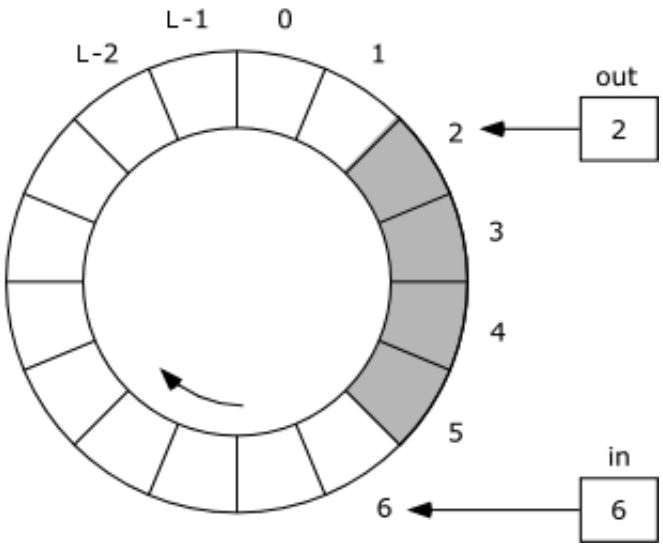
Il s'agit de partager entre tâches une zone de mémoire tampon utilisée comme une **file**.

Le principe de ce modèle est le suivant : un producteur produit des informations et les place dans une file. Un consommateur vide le tampon et consomme les informations. Cela peut être généralisé à plusieurs producteurs ou consommateurs.

Différentes solutions peuvent être envisagées. Une solution courante est de modifier le producteur afin qu'il retienne le nombre de places libres dans la file et se mette en pause au lieu d'écrire les données si la file est pleine. Il faut alors que le consommateur réveille le producteur lorsqu'il a consommé des données et que la file n'est plus pleine. On implémente également un comportement similaire pour le consommateur, qui suspend son activité si la file est vide et est réveillé par le producteur si elle ne l'est plus.

Lire : `https://fr.wikipedia.org/wiki/Problème_des_producteurs_et_des_consommateurs`

Il faut définir le comportement à avoir lorsqu'un thread souhaite lire depuis la file lorsque celle-ci est vide et lorsqu'un thread souhaite écrire dans la file mais que celle-ci est pleine.



Le tampon circulaire (la file) est géré à l'aide de 2 variables partagées `in` et `out` :

- `in` pointe sur le premier emplacement libre pour déposer une information
- `out` pointe sur la prochaine information à retirer

Question 6. Quelles sont les contraintes de synchronisation entre producteurs ?

Question 7. Quelles sont les contraintes de synchronisation entre consommateurs ?

Question 8. Quelles sont les contraintes de synchronisation entre producteur et consommateur ?

Lecteurs/Rédacteurs

Le modèle Lecteurs/Rédacteurs traite de l'accès concurrent en lecture et en écriture à une ressource. Plusieurs *threads* peuvent lire en même temps la ressource, mais il ne peut y avoir qu'un et un seul *thread* en écriture.

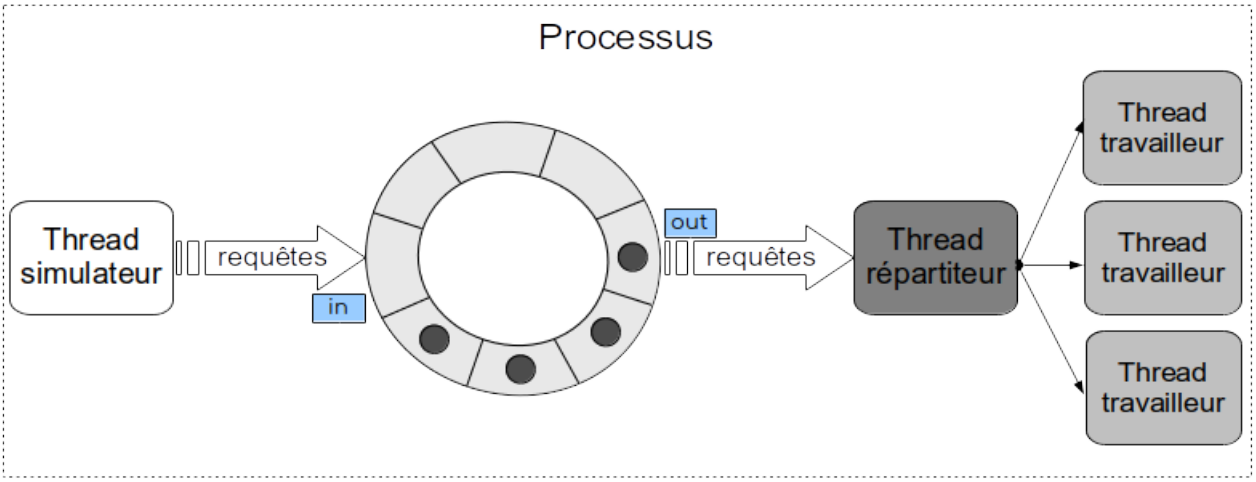
Ce problème classique peut être résolu à l'aide des sémaphores.

Lire : `https://fr.wikipedia.org/wiki/Problème_des_lecteurs_et_des_rédacteurs`

Séquence n°6 : modèle répartiteur/travailleurs

Un *thread*, appelé le répartiteur (ou le maître), reçoit des requêtes pour tout le programme. En fonction de la requête reçue, le répartiteur attribue l'activité à un ou plusieurs *threads* travailleurs (ou esclaves).

Les *threads* travailleurs peuvent être créés dynamiquement lors de l'arrivée d'une requête. Une autre variante est la création anticipée (au départ) de tous les *threads* travailleurs (un *thread* par type de requête). La création d'un intermédiaire (*thread pool*) s'occupant de la gestion des travailleurs peut être aussi envisagé. Ainsi, le répartiteur s'occupe principalement de la réception des requêtes et signale au *thread pool* le traitement à réaliser.



Le *thread* Simulateur génère des requêtes qui sont stockées dans la file. Une requête est constituée d'un `motif` à imprimer un nombre `nb` de fois :

```
typedef struct
{
    char motif;
    int nb;
    int numeroTravailleur;
} Requete;
```

Le type Requete

La file contiendra un nombre fixe de requêtes :

```
// taille de la file pour les requêtes
#define TAILLE_MAX 10

Requete requetes[TAILLE_MAX]; // la file où seront déposées les requêtes
int in = 0; // indique le premier emplacement de libre pour déposer une requête
int out = 0; // indique l'emplacement de la prochaine requête à retirer
```

La file

Le *thread* Répartiteur retire les requêtes de la file et les confie à un *thread* Travailleur. Le nombre de *threads* Travailleur est limité à `NB_THREADS_TRAVAILLEURS` (5 par exemple) :

```
// nombre de threads max
#define NB_THREADS_TRAVAILLEURS 5

pthread_t tidTravailleurs[NB_THREADS_TRAVAILLEURS];
```


Pour chaque requête prélevée dans la file, le *thread* Répartiteur crée un *thread* Travailleur et lui passe en paramètre la requête :

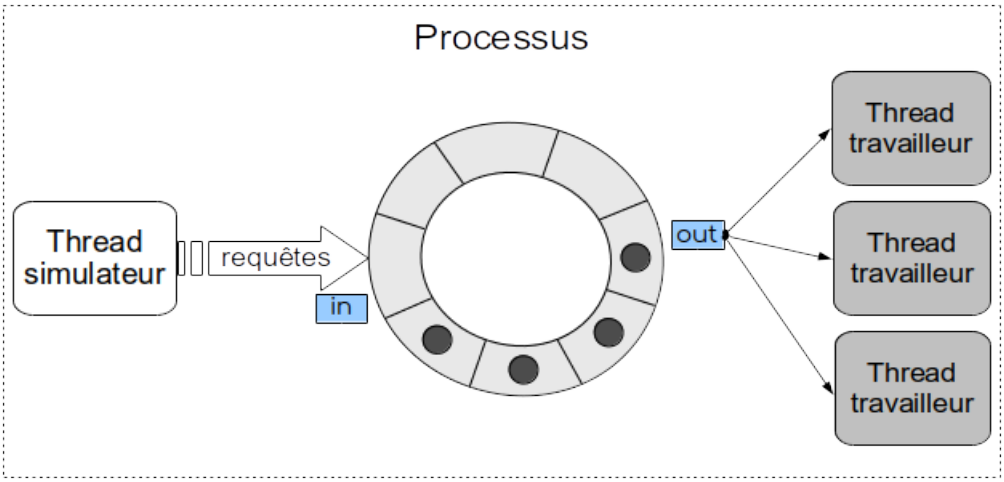
```
Requete *requete;

requete = malloc(sizeof(Requete));
requetes[out].numeroTravailleur = j;
memcpy(requete, &requetes[out], sizeof(Requete));
pthread_create(&tidTravailleurs[j], NULL, travailleur, (void *)requete);
out = (out + 1) % TAILLE_MAX;
```

Les *threads* Travailleurs sont dédiés à la fabrication des rubans. Un *thread* Travailleur traitera une requête reçue en paramètre et lui permettra d'imprimer le ruban souhaité (`nb` x `motif`). Il faut un certain temps pour imprimer un ruban complet.

Question 9. Compléter le programme `main.c` fourni.

Séquence n°7 : modèle en groupe

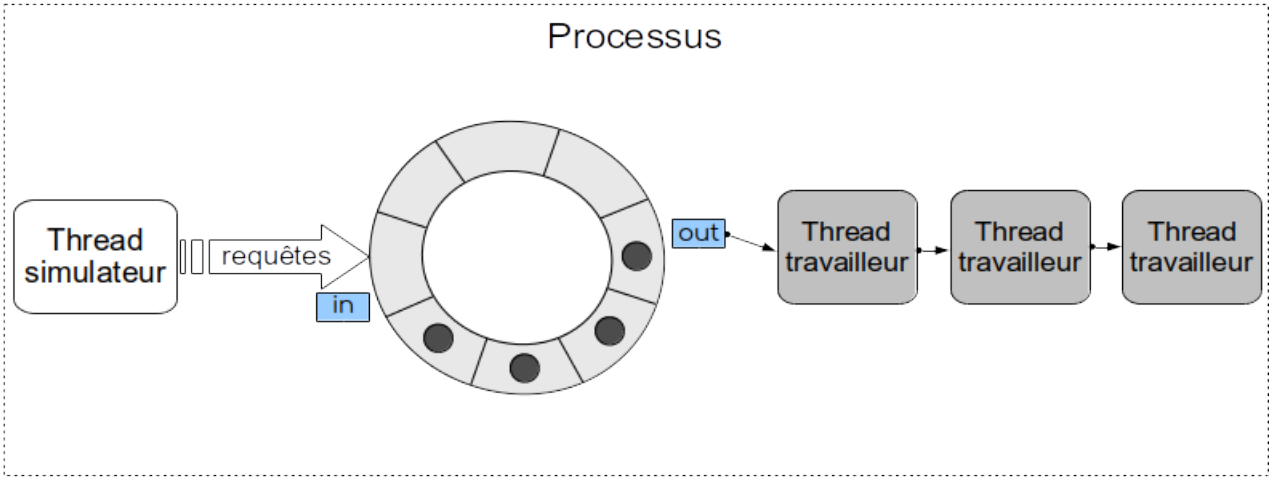
Chaque *thread* réalise les traitements concurremment sans être piloté par un répartiteur (les traitements à effectuer sont déjà connus). Chaque *thread* traite ses propres requêtes (mais il peut être spécialisé pour un certain type de travail).



Question 10. Compléter le programme `main.c` fourni.

Séquence n°8 : modèle en pipeline

L'exécution d'une requête est réalisée par plusieurs *threads* exécutant une partie de la requête en série. Les traitements sont effectués par étape du premier *thread* au dernier. Le premier *thread* peut engendrer des données qu'il passe au suivant.



Le *thread* Simulateur génère des requêtes qui sont stockées dans la file. Une requête est constituée d'un `motif` en `couleur` à imprimer un nombre `nb` de fois :

```
typedef struct
{
    char motif;
    Couleur couleur;
    int nb;
} Requete;
```

Le type Requete

Le rôle des *threads* Travailleurs est modifié : le premier *thread* a pour rôle de convertir le motif en MAJUSCULE. Le deuxième *thread* applique un code césar au motif (un décalage dans la table ASCII). Le troisième et dernier *thread* imprime le ruban : `nb` x `motif` en `couleur`. Il faut un certain temps pour effectuer chaque tâche du traitement complet.

Question 11. Compléter le programme `main.c` fourni.

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 12. Est-ce qu'un *thread* peut accéder aux variables globales de son processus ?

Question 13. Si oui, quel est le problème que cela engendre ?

Question 14. Qu'est-ce qu'un *mutex* ?

Question 15. Dans quels états peut être un *mutex* ?

Question 16. Qu'est-ce qu'une ressource en exclusion mutuelle ?

Question 17. Qu'est-ce qu'une section critique ?

Question 18. Quels dangers entraînent l'utilisation de mécanismes de synchronisation comme les *mutex* ?

Question 19. Qu'est-ce qu'un sémaphore ?

Question 20. Quelles sont les utilisations possibles d'un sémaphore ?

Annexe n°1 : la synchronisation sous Windows

L'API Win32 de Microsoft propose différents moyens pour coordonner l'exécution de plusieurs *threads*.
Les mécanismes fournis sont :

— les exclusions mutuelles
— les sections critiques
— les sémaphores

Mutex

Le mutex est de type `HANDLE`, comme tout objet ressource dans l'API Win32.
Les fonctions utilisables sont :

— Création du mutex : `CreateMutex()`
— Opération `V()`, libération ou déverrouillage du mutex : `ReleaseMutex()`
— Opération `P()`, prise ou verrouillage du mutex : `WaitForSingleObject()`

Ce processus crée le mutex et l'utilise :

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

void P(HANDLE hMutex)
{
   WaitForSingleObject(hMutex, INFINITE);
}

void V(HANDLE hMutex)
{
   ReleaseMutex(hMutex);
}

void travail(int i)
{
   int simulationTravail = 1000 + (int)((double)rand() * (i+100) / (RAND_MAX+1.0));
   printf("la tache%d travaille pendant %.2f s ...\n", i, (float)(simulationTravail/1000.0))
       ;
   Sleep(simulationTravail);
   printf("la tache%d a fini son travail ...\n", i);
}

int main(void)
{
   HANDLE hMutex;

   srand(time(NULL)); //pour la simulation
```



```
   hMutex = CreateMutex(
                     NULL, // descripteur de sécurité par défaut
                     FALSE, // mutex not owned
                     TEXT("NameOfMutexObject")); // nom de l'objet

   if (hMutex == NULL)
   {
      printf( "CreateMutex failed (%d).\n", GetLastError());
      ExitProcess(1);
   }

   P(hMutex);
   printf("le processus1 débloque le processus2 ...\n");
   travail(1);
   printf("Attente pour bloquer l'autre processus - Appuyez sur une touche pour débloquer le
       processus 2\n");
   getchar();
   V(hMutex);

   CloseHandle(hMutex);
   printf("Fin ...\n");

   return 0;
}
```

Processus créateur (mutex1.c)

Ce processus ouvre le mutex et l'utilise :

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

void P(HANDLE hMutex)
{
   WaitForSingleObject(hMutex, INFINITE);
}

void V(HANDLE hMutex)
{
   ReleaseMutex(hMutex);
}

void travail(int i)
{
   int simulationTravail = 1000 + (int)((double)rand() * (i+1000) / (RAND_MAX+1.0));
   printf("la tache%d travaille pendant %.2f s ...\n", i, (float)(simulationTravail/1000.0))
       ;
   Sleep(simulationTravail);
   printf("la tache%d a fini son travail ...\n", i);
}

int main(void)
```

```
{
   HANDLE hMutex;

   srand(time(NULL)); //pour la simulation

   hMutex = OpenMutex(
                    MUTEX_ALL_ACCESS, // Accès complet
                    FALSE, // handle not inheritable
                    TEXT("NameOfMutexObject")); // nom de l'objet

   if (hMutex == NULL)
   {
      printf( "CreateMutex failed (%d).\n", GetLastError() );
      ExitProcess(1);
   }

   printf("le processus2 attend le processus1 ...\n");
   P(hMutex);
   travail(2);
   V(hMutex);

   CloseHandle(hMutex);
   printf("Fin ...\n");

   return 0;
}
```

Processus utilisateur (mutex2.c)

Section Critique

L'objet section critique fournit une synchronisation similaire au mutex sauf que ces objets doivent être utilisées par les threads d'un même processus. Ce mécanisme est plus performant que les mutex et il est plus simple à utiliser.

Si vous ne rendez pas la main à la fin de la section critique, cela bloquera le système. De même, il faut que les sections critiques soient les plus courtes possibles. Le processus est responsable de l'allocation de la mémoire utilisée par la section critique.

Typiquement, il suffit de déclarer une variable de type `CRITICAL_SECTION`, de l'initialiser grâce à la fonction `InitializeCriticalSection()` ou `InitializeCriticalSectionAndSpinCount()`.

Les fonctions utilisables sont :

- Le thread utilise la fonction `EnterCriticalSection()` ou `TryEnterCriticalSection()` avant d'entrer dans la section critique.
- Le thread utilise la fonction `LeaveCriticalSection()` pour rendre le processeur à la fin de la section critique.
- Chaque thread du processus peut utiliser la fonction `DeleteCriticalSection()` pour libérer les ressources système qui ont été allouées quand la section critique a été initialisé. Après son appel, aucune fonction de synchronisation ne peut plus être appelée.

Exemple avec 2 threads :

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

CRITICAL_SECTION cs; // shared structure

void travail(int i)
{
   int simulationTravail = 1000 + (int)((double)rand() * (i+1000) / (RAND_MAX+1.0));
   printf("la tache%d travaille pendant %.2f s ...\n", i, (float)(simulationTravail/1000.0))
       ;
   Sleep(simulationTravail);
   printf("la tache%d a fini son travail ...\n", i);
}

DWORD WINAPI ThreadProc1(LPVOID lpParam)
{
   int n;
   int *pData;

   pData = (int*)lpParam;
   n = *pData;

   EnterCriticalSection(&cs);
   printf("la tache%d est entrée dans une section critique ...\n", n);
   travail(n);
   printf("la tache%d va sortir d'une section critique ...\n", n);
   LeaveCriticalSection(&cs);

   return 0;
}


DWORD WINAPI ThreadProc2(LPVOID lpParam)
{
   int n;
   int *pData;

   pData = (int*)lpParam;
   n = *pData;

   EnterCriticalSection(&cs);
   printf("la tache%d est entrée dans une section critique ...\n", n);
   travail(n);
   printf("la tache%d va sortir d'une section critique ...\n", n);
   LeaveCriticalSection(&cs);

   return 0;
}
```

```
int main(void)
{
   DWORD dwThreadId1, dwThreadId2;
   HANDLE hThread1, hThread2;
   int n1 = 1, n2 = 2;

   InitializeCriticalSection(&cs);

   srand(time(NULL)); //pour la simulation

   // Création du thread (autre que le primaire)
   hThread1 = CreateThread(NULL, // default security attributes
                        0, // use default stack size
                        ThreadProc1, // thread function
                        &n1, // argument to thread function
                        0, // use default creation flags
                        &dwThreadId1); // returns the thread identifier

   if (hThread1 == NULL)
   {
      printf( "CreateThread failed (%d).\n", GetLastError());
      ExitProcess(1);
   }


   // Création du thread (autre que le primaire)
   hThread2 = CreateThread(NULL, // default security attributes
                        0, // use default stack size
                        ThreadProc2, // thread function
                        &n2, // argument to thread function
                        0, // use default creation flags
                        &dwThreadId2); // returns the thread identifier

   if (hThread2 == NULL)
   {
      printf( "CreateThread failed (%d).\n", GetLastError());
      ExitProcess(1);
   }

   // Attente fin des threads
   WaitForSingleObject(hThread1, INFINITE);
   WaitForSingleObject(hThread2, INFINITE);

   CloseHandle(hThread1);
   CloseHandle(hThread2);

   DeleteCriticalSection(&cs);
   printf("Fin ...\n");

   return 0;
}
```

Exemple avec 2 threads (critical_section.c)

Sémaphores

Sous Windows, un sémaphore est un compteur qui peut prendre des valeurs entre 0 et une valeur maximale définie lors de la création du sémaphore.

Le sémaphore est de type `HANDLE`.

Les fonctions utilisables sont :

— La fonction CreateSemaphore() crée l'objet sémaphore.

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // pointer to security attributes
  LONG lInitialCount, // initial count
  LONG lMaximumCount, // maximum count
  LPCTSTR lpName // pointer to semaphore-object name
);
```

— La fonction `OpenSemaphore()` ouvre un objet sémaphore déjà créé par un autre thread ou processus.

— Opération `V()` : la fonction `ReleaseSemaphore()` libère un jeton.

— Opération `P()` : la fonction `WaitForSingleObject()` permet de prendre un sémaphore.

Exemple : 1 x Producteur et 1 x Consommateur

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

/* taille de la file */
#define L 5

/* la file */
int f[L];
int in = 0, out = 0;

/* Les semaphores */
HANDLE nbRequetes; // init 0
HANDLE nbCasesVides; // init L

void P(HANDLE s) {
   WaitForSingleObject(s, INFINITE);
}

void V(HANDLE s) {
   ReleaseSemaphore(s, 1, NULL);
}

int construireRequete(int n)
{
   int simulationTravail = 100 + (int)((double)rand() * (n+1000) / (RAND_MAX+1.0));
   int requete = 1 + (int)((double)rand() * (n+100) / (RAND_MAX+1.0));
   //printf("la tache%d construit une requete en %.3f s ...\n", n, (float)(simulationTravail
      /1000.0));
   Sleep(simulationTravail);
```



```
   return requete;
}

void deposerRequete(int n, int requete)
{
   printf("la tache%d producteur depose la requete %d dans la file a la position %d ...\n",
      n, requete, in);
   f[in] = requete;
   in = (in + 1) % L;
}

int retirerRequete(int n)
{
   int requete;

   printf("la tache%d consommateur retire une requete de la file a la position %d ...\n", n,
       out);
   requete = f[out];
   f[out] = 0; /* élement vide */
   out = (out + 1) % L;

   return requete;
}

void traiterRequete(int n, int requete)
{
   int simulationTravail = 1000 + (int)((double)rand() * (n+1000) / (RAND_MAX+1.0));
   //printf("la tache%d traite la requete en %.2f s ...\n", n, (float)(simulationTravail
      /1000.0));
   Sleep(simulationTravail*2); // on retarde un peu le consommateur !
   requete = 0; // inutilement inutile !
}

DWORD WINAPI producteur(LPVOID nt)
{
   int n = *(int *)nt; // numéro de tache
   int requete;
   int i;

   // on produit pendant un certain temps ...
   for(i=0;i<(L*3);i++)
   {
      requete = construireRequete(n);

      //printf("la tache%d producteur attend une place libre ...\n", n);
      P(nbCasesVides);

      deposerRequete(n, requete);

      V(nbRequetes);
      //printf("la tache%d producteur signale qu'une requete est disponible dans la file
         ...\n", n);
```



```
   }
   return 0;
}

DWORD WINAPI consommateur(LPVOID nt)
{
   int n = *(int *)nt; // numéro de tache
   int requete;
   int i;

   //on consomme pendant un certain temps ...
   for(i=0;i<(L*2);i++)
   {
      //printf("la tache%d consommateur attend une requete disponible dans la file ...\n", n
         );
      P(nbRequetes);

      requete = retirerRequete(n);

      V(nbCasesVides);

      printf("la tache%d consommateur va traiter la requete %d et libere la position ...\n",
          n, requete);
      traiterRequete(n, requete);
   }
   return 0;
}

int main(void)
{
   /* les threads */
   DWORD dwThreadId1, dwThreadId2;
   HANDLE hThread1, hThread2;
   int t1 = 1, t2 = 2; // numéros des taches

   /* init des semaphores */
   nbRequetes  = CreateSemaphore(NULL, 0, L, (LPCTSTR)"SemaphoreNbRequetes");
   nbCasesVides = CreateSemaphore(NULL, L, L, (LPCTSTR)"SemaphoreNbCasesVides");

   // si dans un autre processus alors ouverture d'un sémaphore déjà créé
   //nbRequetes  = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, (LPCTSTR)"SemaphoreNbRequetes")
       ;
   //nbCasesVides = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, (LPCTSTR)"
       SemaphoreNbCasesVides");

   srand(time(NULL)); //pour la simulation

   /* demarre les threads */
   hThread1 = CreateThread(NULL, // default security attributes
                           0, // use default stack size
                           consommateur, // thread function
                           &t1, // argument to thread function
                           0, // use default creation flags
```



```
                        &dwThreadId1); // returns the thread identifier
  hThread2 = CreateThread(NULL, // default security attributes
                        0, // use default stack size
                        producteur, // thread function
                        &t2, // argument to thread function
                        0, // use default creation flags
                        &dwThreadId2); // returns the thread identifier

  /* attends la fin des threads */
  WaitForSingleObject(hThread1, INFINITE);
  WaitForSingleObject(hThread2, INFINITE);
  CloseHandle(hThread1);
  CloseHandle(hThread2);
  CloseHandle(nbRequetes);
  CloseHandle(nbCasesVides);
  printf("Fin ...\n");
  return 0;
}
```

Exemple : 1 x Producteur et 1 x Consommateur (producteur1.c)

