Initiation à la programmation multitâche en C avec Pthreads

Table des matières

- I. Introduction
- II. Avant de commencer
- III. Qu'est-ce qu'un thread ?
- IV. Création et exécution des threads
  - IV-A. pthread_create
- V. Annulation et fin des threads
  - V-A. pthread_exit
  - V-B. pthread_cancel
    - V-B-1. pthread_setcancelstate
  - V-C. pthread_join
- VI. Mise en pratique
  - VI-A. Code complet
  - VI-B. Sortie du programme
  - VI-C. Observations
- VII. Les mutex
  - VII-A. pthread_mutex_lock
  - VII-B. pthread_mutex_unlock
- VIII. Mise en pratique
  - VIII-A. Code complet
  - VIII-B. Sortie du programme
  - VIII-C. Observations
- IX. Les conditions
  - IX-A. pthread_cond_wait
  - IX-B. pthread_cond_signal
  - IX-C. pthread_cond_broadcast
- X. Mise en pratique
  - X-A. Fonction : fn_store
  - X-B. Fonction : fn_clients
  - X-C. Code complet
  - X-D. Sortie du programme
  - X-E. Observations
- XI. Test des threads
  - XI-A. pthread_equal
  - XI-B. pthread_self
- XII. Conclusion
- XIII. Remerciements

Les threads permettent de créer des programmes multitâches, ce tutoriel vous propose une approche par la pratique en partant d'un exemple unique !

Commentez cet article : 6 commentaires ⭐⭐⭐⭐⭐ ♪

Article lu 209282 fois.

L'auteur

Franck Hecht

L'article

Publié le 4 novembre 2007

Version PDF Version hors-ligne

ePub, Azw et Mobi

Liens sociaux

I. Introduction ▲

La programmation multitâche a toujours été et est toujours, un sujet assez complexe, essentiellement dû au manque de documentation et tutoriels en français !

Ce tutoriel ne se veut pas une documentation complète (il existe des ouvrages dédiés sur ce sujet), mais plutôt une initiation pour vous donner un aperçu de ce qu'il est possible de faire et nous ferons un petit tour d'horizon de la bibliothèque **pthread** au fil de l'eau.

Nous nous baserons sur un sujet d'exemple unique soit, une gestion correcte d'un stock de magasin et des clients. Le but bien entendu, est que le stock ne descende jamais en dessous de zéro, ce qui dans la réalité n'est pas faisable donc, si le stock descend à zéro ou si le client courant demande plus de produits qu'il y en a en stock, il faut renflouer celui-ci. Chaque client est représenté par un thread, dans notre exemple nous allons en créer **5**, la gestion du stock est également un thread supplémentaire.

II. Avant de commencer ▲

La bibliothèque Pthreads est portable, elle existe sur Linux ainsi que sur Windows. Si vous êtes sur Windows, il vous faudra cependant l'installer, car elle ne l'est pas d'office sur ce système ! Vous pouvez télécharger la bibliothèque sur le site suivant: http://sourceware.org/pthreads-win32/.

Pour pouvoir compiler un projet (tous systèmes) avec pthread, il faut pour commencer, ajouter l'en-tête :

Sélectionnez

```
#include <pthread.h>
```

ainsi qu'ajouter à l'éditeur de lien la bibliothèque :

Sélectionnez

```
-lpthread
```

et spécifier au compilateur la constante :

Sélectionnez

```
-D_REENTRANT
```

Vous voilà prêt pour compiler des programmes utilisant la bibliothèque Pthreads !

Attention, avec le compilateur MingW sous Windows, si vous développez une application C++ utilisant des exceptions, il est nécessaire de compiler et de réaliser l'édition des liens avec l'option **-mthreads**. En effet, d'origine, les exceptions ne sont pas thread-safe, l'option mthreads permet qu'elles le soient.

Les utilisateurs de Linux devront spécifier dans la ligne de compilation des exemples de ce tutoriel, la constante **-DLinux** et les utilisateurs de Windows **-DWin32**. Cela sert à prendre en charge de façon portable la mise en pause des portions du code d'exemple. Un fichier Makefile est également disponible pour les utilisateurs de Linux !

III. Qu'est-ce qu'un thread ? ▲

Un **thread**(Fil ou encore Fil d'exécution) est une portion de code (fonction) qui se déroule en parallèle au thread principal (aussi appelé main). Ce principe est un peu semblable à la fonction fork sur Linux par exemple sauf que nous ne faisons pas de copie du processus père, nous définissons des fonctions qui vont se lancer en même temps que le processus, ce qui permet de faire de la programmation multitâche. Le but est donc de permettre au programme de réaliser plusieurs actions au même moment (imaginez un programme qui fait un gros calcul et une barre de progression qui avance en même temps).

On peut également considérer un thread comme un processus allégé pour mieux imager le tout ! En comparaison des threads, un fork prend en moyenne 30 fois plus de temps à faire !

IV. Création et exécution des threads ▲

IV-A. pthread_create ▲

Un thread se crée avec la fonction :

Sélectionnez

```
int pthread_create (pthread_t * thread, pthread_attr_t * attr, void * (*
start_routine) (void *), void * arg);
```

Voyons dans l'ordre, à quoi correspondent ses arguments.

1. Le type **pthread_t** est un type opaque, sa valeur réelle dépend de l'implémentation (sur Linux il s'agit en générale du type **unsigned long**). Ce type correspond à l'identifiant du thread qui sera créé, tout comme les processus ont leur propre identifiant.
2. Le type **pthread_attr_t** est un autre type opaque permettant de définir des attributs spécifiques pour chaque thread, mais cela dépasse le cadre de ce tutoriel. Il faut simplement savoir que l'on peut changer le comportement de la gestion des threads par exemple, les régler pour qu'ils tournent sur un système temps réel ! En général on se contente des attributs par défaut donc en mettant cet argument à NULL.
3. Chaque thread dispose d'une fonction à exécuter, c'est en même temps sa raison de vivre... Cet argument permet de transmettre un pointeur sur la fonction qu'il devra exécuter.
4. Ce dernier argument représente un argument que l'on peut passer à la fonction que le thread doit exécuter.

Si la création réussit, la fonction renvoie **0**(zéro) et l'identifiant du thread nouvellement créé est stocké à l'adresse fournie en premier argument. En cas d'erreur, la valeur **EAGAIN** est retournée par la fonction s'il n'y a pas assez de ressources système pour créer un nouveau thread ou bien si le nombre maximum de threads définis par la constante **PTHREAD_THREADS_MAX** est atteint !

Le nombre de threads simultanés est limité suivant les systèmes. La constante **PTHREAD_THREADS_MAX** définit le nombre maximum qui est de **1024** sur les Unixoïdes !

Lorsque le thread est créé, il est lancé immédiatement et exécute la fonction passée en troisième argument. L'exécution du thread se fait soit jusqu'à la fin de sa fonction ou bien jusqu'à son annulation, c'est ce que nous allons voir au prochain chapitre.

Il est possible d'attribuer la même fonction à plusieurs threads !

V. Annulation et fin des threads ▲

V-A. pthread_exit ▲

On peut arrêter le thread courant avec la fonction :

Sélectionnez
```
void pthread_exit (void * retval);
```

Son seul argument est le retour de la fonction du thread appelant. Cet argument peut aussi être récupéré par la fonction **pthread_join** que nous verrons plus bas.

V-B. pthread_cancel ▲

On peut annuler un thread à partir d'un autre à n'importe quel moment avec la fonction :

Sélectionnez
```
int pthread_cancel (pthread_t thread);
```

L'argument de cette fonction est le thread à annuler. Elle renvoie **0**(zéro) si elle réussie ou la valeur **ESRCH** si aucun thread ne correspond à celui passé en argument.

Il faut cependant être très vigilant lors de l'utilisation de cette fonction. En effet, si le thread dont on demande l'arrêt possède un verrou et ne l'a toujours pas relâché, il y a un risque de laisser ce verrou dans l'état verrouillé, il sera alors dans ce cas impossible de le récupérer !

Pour éviter ce genre de phénomène, on peut avoir recours à une fonction permettant de changer le comportement du thread par rapport aux requêtes d'annulations, ce que nous allons voir ci-dessous !

V-B-1. pthread_setcancelstate ▲

Sélectionnez
```
int pthread_setcancelstate (int state, int * etat_pred);
```

Cette fonction permet de changer le comportement du thread appelant par rapport aux requêtes d'annulations. Ces arguments sont dans l'ordre :

- état d'annulation. Il peut prendre les deux valeurs suivantes :
  - **PTHREAD_CANCEL_ENABLE :** autorise les annulations pour le thread appelant,
  - **PTHREAD_CANCEL_DISABLE :** désactive les requêtes d'annulation ;
- adresse vers l'état précédent (ou NULL) permettant ainsi sa restauration lors d'un prochain appel de la fonction.

La fonction renvoie la valeur **0** en cas de succès ou **EINVAL** si l'argument ne correspond ni à PTHREAD_CANCEL_ENABLE et ni à PTHREAD_CANCEL_DISABLE !

V-C. pthread_join ▲

Lorsque nous créons des threads puis nous laissons continuer par exemple la fonction main, nous prenons le risque de terminer le programme complètement sans avoir pu exécuter les threads. Nous devons en effet attendre que les différents threads créés se terminent. Pour cela, il existe la fonction :

Sélectionnez
```
int pthread_join (pthread_t th, void ** thread_return);
```

Ses arguments sont dans l'ordre :

1. Le thread à attendre ;
2. La valeur de retour de la fonction du thread **th**.

L'appel de cette fonction met en pause l'exécution du thread appelant jusqu'au retour de la fonction. Si aucun problème n'a eu lieu, elle retourne **0**(zéro) et la valeur de retour du thread est passé à l'adresse indiquée (second argument) si elle est différente de NULL. En cas de problème, la fonction retourne une des valeurs suivantes :

- **ESRCH :** aucun thread ne correspond à celui passé en argument.
- **EINVAL :** le thread a été détaché ou un autre thread attend déjà la fin du même thread.
- **EDEADLK :** le thread passé en argument correspond au thread appelant.

Un thread terminé ne peut être relancé, il faut en créer un nouveau, car un thread qui touche à sa fin est implicitement détruit !

VI. Mise en pratique ▲

Dans ce programme, nous créons **un** thread pour la gestion du stock du magasin et **cinq** threads pour les clients. Les deux fonctions **fn_store** et **fn_clients** sont des boucles infinies (pour cet exemple, mais dans la réalité, ça ne sera pas toujours le cas) exécutant les mêmes tâches. Les threads clients prennent dans le stock et le thread du magasin va renflouer le stock dès qu'il devient trop bas pour satisfaire les clients. Le nombre d'articles pris du stock est un nombre aléatoire ainsi que l'ordre de passage des clients.

Nous pouvons voir qu'à la fin de la fonction main, nous avons une boucle qui parcourt chaque thread en lançant la fonction pthread_join. Ceci permet d'attendre la fin des threads et évite donc que le programme ne se termine et quitte prématurément les threads !

VI-A. Code complet ▲

Exemple numéro 1. Vous pouvez télécharger l'archive ici !

Sélectionnez

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

#if defined (Win32)
#  include <windows.h>
#  define psleep(sec) Sleep ((sec) * 1000)
#elif defined (Linux)
#  include <unistd.h>
#  define psleep(sec) sleep ((sec))
#endif

#define INITIAL_STOCK   20
#define NB_CLIENTS      5


/* Structure stockant les informations des threads clients et du magasin. */
typedef struct
{
   int stock;

   pthread_t thread_store;
   pthread_t thread_clients [NB_CLIENTS];
}
store_t;

static store_t store =
{
   .stock = INITIAL_STOCK,
};


/* Fonction pour tirer un nombre au sort entre 0 et max. */
static int get_random (int max)
{
   double val;

   val = (double) max * rand ();
   val = val / (RAND_MAX + 1.0);

   return ((int) val);
}


/* Fonction pour le thread du magasin. */
static void * fn_store (void * p_data)
{
   while (1)
   {
      if (store.stock <= 0)
      {
         store.stock = INITIAL_STOCK;
         printf ("Remplissage du stock de %d articles !\n", store.stock);
      }
   }

   return NULL;
}


/* Fonction pour les threads des clients. */
static void * fn_clients (void * p_data)
{
   int nb = (int) p_data;

   while (1)
   {
      int val = get_random (6);

      psleep (get_random (3));

      store.stock = store.stock - val;
      printf (
         "Client %d prend %d du stock, reste %d en stock !\n",
         nb, val, store.stock
      );
   }

   return NULL;
}


int main (void)
```

```
{
   int i = 0;
   int ret = 0;


   /* Creation du thread du magasin. */
   printf ("Creation du thread du magasin !\n");
   ret = pthread_create (
      & store.thread_store, NULL,
      fn_store, NULL
   );

   /* Creation des threads des clients si celui du magasin a reussi. */
   if (! ret)
   {
      printf ("Creation des threads clients !\n");
      for (i = 0; i < NB_CLIENTS; i++)
      {
         ret = pthread_create (
            & store.thread_clients [i], NULL,
            fn_clients, (void *) i
         );

         if (ret)
         {
            fprintf (stderr, "%s", strerror (ret));
         }
      }
   }
   else
   {
      fprintf (stderr, "%s", strerror (ret));
   }


   /* Attente de la fin des threads. */
   i = 0;
   for (i = 0; i < NB_CLIENTS; i++)
   {
      pthread_join (store.thread_clients [i], NULL);
   }
   pthread_join (store.thread_store, NULL);


   return EXIT_SUCCESS;
}
```

L'exemple de code ci-dessus utilise une variable globale ! Ici ce n'est qu'à titre d'exemple, mais je vous encourage à éviter ce genre de pratique autant que possible !

VI-B. Sortie du programme ▲

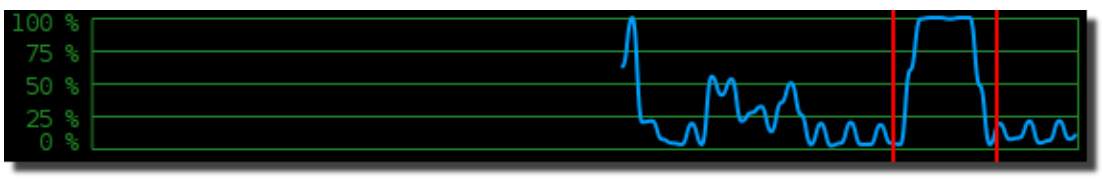
Voici la sortie du programme sur la console avec annulation utilisateur :

Sélectionnez

```
Creation du thread du magasin !
Creation des threads clients !
Client 2 prend 5 du stock, reste 15 en stock !
Client 0 prend 5 du stock, reste 10 en stock !
Client 3 prend 1 du stock, reste 9 en stock !
Client 4 prend 2 du stock, reste 7 en stock !
Client 1 prend 4 du stock, reste 3 en stock !
Client 2 prend 2 du stock, reste 1 en stock !
Client 2 prend 0 du stock, reste 1 en stock !
Client 0 prend 2 du stock, reste -1 en stock !
Remplissage du stock de 20 articles !
Client 1 prend 0 du stock, reste 20 en stock !
Client 1 prend 0 du stock, reste 20 en stock !
Client 1 prend 5 du stock, reste 15 en stock !
Client 0 prend 0 du stock, reste 15 en stock !
Client 0 prend 3 du stock, reste 12 en stock !
Client 3 prend 5 du stock, reste 7 en stock !
Client 4 prend 3 du stock, reste 4 en stock !
Client 2 prend 0 du stock, reste 4 en stock !
Client 0 prend 3 du stock, reste 1 en stock !
Client 1 prend 3 du stock, reste -2 en stock !
Client 3 prend 2 du stock, reste -4 en stock !
Client 4 prend 1 du stock, reste -5 en stock !
Remplissage du stock de 20 articles !
Client 2 prend 3 du stock, reste 17 en stock !
Client 2 prend 3 du stock, reste 14 en stock !
Client 1 prend 1 du stock, reste 13 en stock !
Client 0 prend 2 du stock, reste 11 en stock !
Client 0 prend 0 du stock, reste 11 en stock !
Client 0 prend 2 du stock, reste 9 en stock !

<CTRL-C>
```

Et la charge CPU utilisée durant le déroulement du processus et ses threads :



Charge CPU de l'exemple 1

VI-C. Observations ▲

Nous pouvons voir grâce à ces données crées lors du déroulement du programme que les clients se servent même si plus aucun produit n'est en stock, le stock est donc durant un court moment en négatif, ce qu'il faut éviter dans la réalité !

Non seulement les clients se servent au petit bonheur, mais en plus, sans prendre en considération que d'autres clients peuvent également prendre dans le même stock et bien sûr, sans prendre en compte que le magasin peut renflouer son stock quand il est au plus bas.

Ceci pour noter une chose importante : Les threads ne tiennent en aucun cas compte qu'une autre fonction puisse également accéder à la même variable (accès concurrentiels), en lecture, mais aussi en écriture. Cela peut avoir des répercussions dramatiques dans une application critique !

Le prochain chapitre va donc aborder la protection pour les accès concurrentiels autrement dit, les **mutex !**

En outre, nous pouvons observer la charge CPU utilisée lors du déroulement du programme. On peut noter que les ressources sont utilisées au maximum, ceci surtout dû au fait que la fonction **fn_store** tourne sans cesse jusqu'à ce qu'on mette fin au programme ! Nous aborderons ce sujet dans la dernière partie de ce tutoriel.

VII. Les mutex ▲

Le(s)**mutex**(**mut**ual **ex**clusion ou zone d'exclusion mutuelle) est(sont) un système de verrou donnant ainsi une garantie sur la viabilité des données manipulées par les threads. En effet, il arrive même très souvent que plusieurs threads doivent accéder en lecture et/ou en écriture aux mêmes variables. Si un thread possède le verrou, seulement celui-ci peut lire et écrire sur les variables étant dans la portion de code protégée (aussi appelée zone critique). Lorsque le thread a terminé, il libère le verrou et un autre thread peut le prendre à son tour.

Pour créer un mutex, il faut tout simplement déclarer une variable du type **pthread_mutex_t** et l'initialiser avec la constante **PTHREAD_MUTEX_INITIALIZER** soit par exemple :

Sélectionnez

```
static pthread_mutex_t mutex_stock = PTHREAD_MUTEX_INITIALIZER;
```

Un mutex n'a que deux états possibles, il est soit verrouillé, soit déverrouillé. On utilise les deux fonctions ci-dessous pour changer les états.

VII-A. pthread_mutex_lock ▲

Sélectionnez

```
int pthread_mutex_lock (pthread_mutex_t * mutex);
```

Cette fonction permet de déterminer le début d'une zone critique. Son seul argument est l'adresse d'un mutex de type **pthread_mutex_t**. La fonction renvoie **0** en cas de succès ou l'une des valeurs suivantes en cas d'échec :

- **EINVAL :** mutex non initialisé.
- **EDEADLK :** mutex déjà verrouillé par un thread différent.

VII-B. pthread_mutex_unlock ▲

Sélectionnez
```
int pthread_mutex_unlock (pthread_mutex_t * mutex);
```

Cette fonction permet de relâcher le verrou passé en argument qui est l'adresse d'un mutex de type **pthread_mutex_t**. La fonction renvoie **0** en cas de succès ou l'une des valeurs suivantes en cas d'échec :

- **EINVAL :** mutex non initialisé ;
- **EPERM :** le thread n'a pas la main sur le mutex.

VIII. Mise en pratique ▲

Dans la seconde version de notre exemple, des zones critiques ont été définies dans les fonctions **fn_store** et **fn_clients**. On peut remarquer que nous prenons et libérons le mutex à chaque tour de boucle ce qui permet de ne pas bloquer le programme et ainsi, chaque thread aura l'opportunité de le prendre à son tour pour accomplir sa tâche.

Avant chaque arrêt/annulation/fin d'un thread, il ne faut surtout pas oublier de libérer les verrous, car vous risquez le cas échéant, d'obtenir ce qu'on appelle un Dead Lock, le mutex est verrouillé et le restera. Tous les threads voulant l'utiliser vont s'arrêter.

VIII-A. Code complet ▲

Exemple numéro 2. Vous pouvez télécharger l'archive ici !

Sélectionnez

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

#if defined (Win32)
#  include <windows.h>
#  define psleep(sec) Sleep ((sec) * 1000)
#elif defined (Linux)
#  include <unistd.h>
#  define psleep(sec) sleep ((sec))
#endif

#define INITIAL_STOCK   20
#define NB_CLIENTS      5


/* Structure stockant les informations des threads clients et du magasin. */
typedef struct
{
   int stock;

   pthread_t thread_store;
   pthread_t thread_clients [NB_CLIENTS];

   pthread_mutex_t mutex_stock;
}
store_t;

static store_t store =
{
   .stock = INITIAL_STOCK,
   .mutex_stock = PTHREAD_MUTEX_INITIALIZER,
};


/* Fonction pour tirer un nombre au sort entre 0 et max. */
static int get_random (int max)
{
   double val;

   val = (double) max * rand ();
   val = val / (RAND_MAX + 1.0);

   return ((int) val);
}


/* Fonction pour le thread du magasin. */
static void * fn_store (void * p_data)
{
   while (1)
   {
      /* Debut de la zone protegee. */
      pthread_mutex_lock (& store.mutex_stock);

      if (store.stock <= 0)
      {
         store.stock = INITIAL_STOCK;
         printf ("Remplissage du stock de %d articles !\n", store.stock);
      }

      /* Fin de la zone protegee. */
      pthread_mutex_unlock (& store.mutex_stock);
   }

   return NULL;
}


/* Fonction pour les threads des clients. */
static void * fn_clients (void * p_data)
{
   int nb = (int) p_data;

   while (1)
   {
      int val = get_random (6);


      /* Debut de la zone protegee. */
      pthread_mutex_lock (& store.mutex_stock);

      psleep (get_random (3));
```

```
      store.stock = store.stock - val;
      printf (
         "Client %d prend %d du stock, reste %d en stock !\n",
         nb, val, store.stock
      );

      /* Fin de la zone protegee. */
      pthread_mutex_unlock (& store.mutex_stock);
   }

   return NULL;
}


int main (void)
{
   int i = 0;
   int ret = 0;


   /* Creation du thread du magasin. */
   printf ("Creation du thread du magasin !\n");
   ret = pthread_create (
      & store.thread_store, NULL,
      fn_store, NULL
   );

/* Creation des threads des clients si celui du magasin a reussi. */
if (! ret)
{
   printf ("Creation des threads clients !\n");
   for (i = 0; i < NB_CLIENTS; i++)
   {
      ret = pthread_create (
         & store.thread_clients [i], NULL,
         fn_clients, (void *) i
      );

      if (ret)
      {
         fprintf (stderr, "%s", strerror (ret));
      }
   }
}
else
{
   fprintf (stderr, "%s", strerror (ret));
}


/* Attente de la fin des threads. */
i = 0;
for (i = 0; i < NB_CLIENTS; i++)
{
   pthread_join (store.thread_clients [i], NULL);
}
pthread_join (store.thread_store, NULL);


   return EXIT_SUCCESS;
}
```

L'exemple de code ci-dessus utilise une variable globale ! Ici ce n'est qu'à titre d'exemple, mais je vous encourage à éviter ce genre de pratique autant que possible !

VIII-B. Sortie du programme ▲

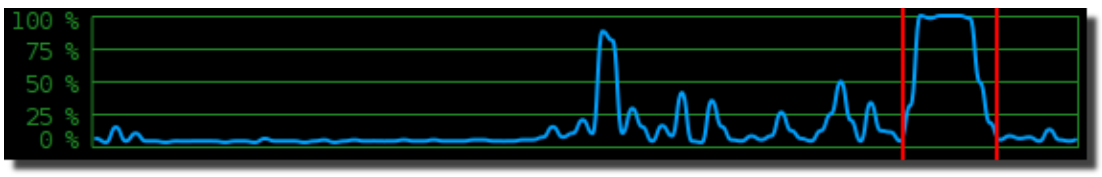
Voici la sortie du programme sur la console avec annulation utilisateur :

Sélectionnez

```
Creation du thread du magasin !
Creation des threads clients !
Client 2 prend 5 du stock, reste 15 en stock !
Client 0 prend 5 du stock, reste 10 en stock !
Client 3 prend 1 du stock, reste 9 en stock !
Client 4 prend 2 du stock, reste 7 en stock !
Client 1 prend 4 du stock, reste 3 en stock !
Client 2 prend 2 du stock, reste 1 en stock !
Client 2 prend 0 du stock, reste 1 en stock !
Client 0 prend 2 du stock, reste -1 en stock !
Remplissage du stock de 20 articles !
Client 1 prend 0 du stock, reste 20 en stock !
Client 1 prend 0 du stock, reste 20 en stock !
Client 1 prend 5 du stock, reste 15 en stock !
Client 3 prend 5 du stock, reste 10 en stock !
Client 3 prend 3 du stock, reste 7 en stock !
Client 4 prend 3 du stock, reste 4 en stock !
Client 0 prend 0 du stock, reste 4 en stock !
Client 2 prend 0 du stock, reste 4 en stock !
Client 3 prend 3 du stock, reste 1 en stock !
Client 1 prend 3 du stock, reste -2 en stock !
Client 4 prend 2 du stock, reste -4 en stock !
Client 0 prend 1 du stock, reste -5 en stock !
Remplissage du stock de 20 articles !
Client 2 prend 3 du stock, reste 17 en stock !
Client 2 prend 3 du stock, reste 14 en stock !
Client 3 prend 2 du stock, reste 12 en stock !
Client 1 prend 1 du stock, reste 11 en stock !
Client 1 prend 0 du stock, reste 11 en stock !
Client 1 prend 2 du stock, reste 9 en stock !

<CTRL-C>
```

Et la charge CPU utilisée durant le déroulement du processus et ses threads :



Charge CPU de l'exemple 2

VIII-C. Observations ▲

Il n'y a pas à vrai dire, de changement radical par rapport à l'affichage du résultat sur la sortie console, mais nous avons néanmoins protégé l'accès aux données, ce qui fait un risque en moins. Bien sûr, ce programme d'exemple n'est pas un programme critique, mais sur d'autres applications cela peut avoir des effets désastreux !

Nous pouvons également remarquer que la charge CPU reste inchangée, le programme prend presque toutes les ressources disponibles du processeur. Pour pallier ce problème, nous pouvons mettre des threads en attente jusqu'à ce que des conditions de réveil soient remplies ! Nous allons donc étudier dans la partie suivante, les **conditions !**

IX. Les conditions ▲

Les conditions sont un autre mécanisme de synchronisation, le principe est simple. Lorsqu'un thread doit attendre une condition (comme dans notre exemple, le thread du magasin doit attendre que le stock ne suffise plus aux clients pour le renflouer) pour exécuter sa tâche, nous pouvons le mettre en attente. Un autre thread qui possède le verrou réveille alors le thread dormant lorsque la condition est remplie. Tous les threads ne sont pas des boucles infinies, certains sont créés juste pour faire une action précise puis sont automatiquement détruits, mais pour d'autres, comme notre exemple, les conditions sont un atout dans la consommation des ressources, car l'attente d'une condition met systématiquement le thread en pause !

Créer une condition repose sur le même principe que les mutex à savoir, la création et initialisation d'une variable de type **pthread_cond_t** soit par exemple :

Sélectionnez

```
static pthread_cond_t cond_stock = PTHREAD_COND_INITIALIZER;
```

Les conditions reposent essentiellement sur deux fonctions. Une permet de mettre en attente un thread et la seconde permet de signaler que la condition est remplie ce qui réveille alors le thread qui est en attente de cette condition.

Plusieurs threads peuvent surveiller la même condition.

Les fonctions principales sont les suivantes (elles retournent toutes **0,** mais ne renvoient jamais de code d'erreur) :

IX-A. pthread_cond_wait ▲

Sélectionnez

```
int pthread_cond_wait (pthread_cond_t * cond, pthread_mutex_t * mutex);
```

Cette fonction permet de mettre le thread appelant en attente de la condition, il suspend donc son exécution temporairement. Ses deux arguments sont dans l'ordre :

- l'adresse de la variable condition de type **pthread_cond_t** ;
- l'adresse d'un mutex. Une condition est en effet, toujours associée à un mutex.

IX-B. pthread_cond_signal ▲

Sélectionnez
```
int pthread_cond_signal (pthread_cond_t * cond);
```

pthread_cond_signal est la fonction qui permet de signaler la condition au thread qui l'attend. Elle prend en paramètre l'adresse de la variable-condition surveillée. Cette fonction ne permet de réveiller qu'un seul thread. Pour en réveiller davantage, il faut utiliser la fonction ci-dessous.

IX-C. pthread_cond_broadcast ▲

Sélectionnez
```
int pthread_cond_broadcast (pthread_cond_t * cond);
```

Comme il a été signalé plus haut, plusieurs threads peuvent surveiller la même condition. Cette fonction permet de tous les réveiller. Tout comme pthread_cond_signal, elle prend en paramètre l'adresse de la variable-condition surveillée.

X. Mise en pratique ▲

Le raisonnement de notre programme change à partir de maintenant. En effet, prenons le cas d'un client **X** qui veut prendre **N** articles du stock du magasin. Si le stock ne suffit pas, le magasin doit le renflouer, mais cependant, le client doit attendre que le stock soit suffisant pour satisfaire sa demande.

Le client signale donc au magasin qu'il doit remplir à nouveau son stock et en attendant, il se met en attente. Une fois que le magasin a renfloué son stock, il le signal au client en attente et se met à nouveau lui-même en attente jusqu'à ce que la condition soit de nouveau signalée !

Quelques changements ont donc eu lieu dans notre programme. Nous avons pour commencer, deux variables-conditions qui viennent s'ajouter à notre structure, une pour les clients et une pour le magasin. La raison en est simple, lorsque le magasin remplit son stock, le client en cours doit attendre que le stock soit à nouveau à un niveau correct, ceci permet d'éviter qu'il descende à un chiffre négatif, nous gardons de ce fait un cheminement logique pour notre programme. Des changements ont aussi été faits sur les deux fonctions **fn_store** et **fn_clients**.

X-A. Fonction : fn_store ▲

Sélectionnez
```
static void * fn_store (void * p_data)
{
   while (1)
   {
      pthread_mutex_lock (& store.mutex_stock);
      pthread_cond_wait (& store.cond_stock, & store.mutex_stock);

      store.stock = INITIAL_STOCK;
      printf ("Remplissage du stock de %d articles !\n", store.stock);

      pthread_cond_signal (& store.cond_clients);
      pthread_mutex_unlock (& store.mutex_stock);
   }

   return NULL;
}
```

Le thread attendant que la condition soit remplie pour s'activer, nous n'avons plus besoin du test à l'intérieur de la boucle, nous pouvons donc directement remplir le stock. Nous pouvons voir un processus particulier en ce qui concerne les conditions. En effet, si on regarde le début du corps de la boucle, on peut s'apercevoir que le thread prend le mutex et se met en attente de la condition. En réalité, le thread relâche le mutex aussitôt et le reprend automatiquement lorsque la condition est vraie et qu'il soit réveillé par un autre thread. Lorsque le stock est rempli, la fonction le signale au thread du client courant.

X-B. Fonction : fn_clients ▲

Sélectionnez

```
static void * fn_clients (void * p_data)
{
   int nb = (int) p_data;

   while (1)
   {
      int val = get_random (6);


      psleep (get_random (3));
      pthread_mutex_lock (& store.mutex_stock);

      if (val > store.stock)
      {
         pthread_cond_signal (& store.cond_stock);
         pthread_cond_wait (& store.cond_clients, & store.mutex_stock);
      }

      store.stock = store.stock - val;
      printf (
         "Client %d prend %d du stock, reste %d en stock !\n",
         nb, val, store.stock
      );

      pthread_mutex_unlock (& store.mutex_stock);
   }

   return NULL;
}
```

Cette fonction a aussi eu droit à un petit lifting. En effet, un test a été rajouté qui permet de déterminer si le stock est en quantité suffisante par rapport à la demande du client. Si ce n'est pas le cas, le thread le signal au thread du magasin qui prend le relais, le thread appelant ce met en attente le temps que le magasin réalise sa tâche.

X-C. Code complet ▲

Exemple numéro 3. Vous pouvez télécharger l'archive ici !

Sélectionnez

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

#if defined (Win32)
#  include <windows.h>
#  define psleep(sec) Sleep ((sec) * 1000)
#elif defined (Linux)
#  include <unistd.h>
#  define psleep(sec) sleep ((sec))
#endif

#define INITIAL_STOCK   20
#define NB_CLIENTS      5


/* Structure stockant les informations des threads clients et du magasin. */
typedef struct
{
   int stock;

   pthread_t thread_store;
   pthread_t thread_clients [NB_CLIENTS];

   pthread_mutex_t mutex_stock;
   pthread_cond_t cond_stock;
   pthread_cond_t cond_clients;
}
store_t;

static store_t store =
{
   .stock = INITIAL_STOCK,
   .mutex_stock = PTHREAD_MUTEX_INITIALIZER,
   .cond_stock = PTHREAD_COND_INITIALIZER,
   .cond_clients = PTHREAD_COND_INITIALIZER,
};


/* Fonction pour tirer un nombre au sort entre 0 et max. */
static int get_random (int max)
{
   double val;

   val = (double) max * rand ();
   val = val / (RAND_MAX + 1.0);

   return ((int) val);
}


/* Fonction pour le thread du magasin. */
static void * fn_store (void * p_data)
{
   while (1)
   {
      /* Debut de la zone protegee. */
      pthread_mutex_lock (& store.mutex_stock);
      pthread_cond_wait (& store.cond_stock, & store.mutex_stock);

      store.stock = INITIAL_STOCK;
      printf ("Remplissage du stock de %d articles !\n", store.stock);

      pthread_cond_signal (& store.cond_clients);
      pthread_mutex_unlock (& store.mutex_stock);
      /* Fin de la zone protegee. */
   }

   return NULL;
}


/* Fonction pour les threads des clients. */
static void * fn_clients (void * p_data)
{
   int nb = (int) p_data;

   while (1)
   {
      int val = get_random (6);


      psleep (get_random (3));
```

```
      /* Debut de la zone protegee. */
      pthread_mutex_lock (& store.mutex_stock);

      if (val > store.stock)
      {
         pthread_cond_signal (& store.cond_stock);
         pthread_cond_wait (& store.cond_clients, & store.mutex_stock);
      }

      store.stock = store.stock - val;
      printf (
         "Client %d prend %d du stock, reste %d en stock !\n",
         nb, val, store.stock
      );

      pthread_mutex_unlock (& store.mutex_stock);
      /* Fin de la zone protegee. */
   }

   return NULL;
}


int main (void)
{
   int i = 0;
   int ret = 0;


/* Creation des threads. */
printf ("Creation du thread du magasin !\n");
ret = pthread_create (
   & store.thread_store, NULL,
   fn_store, NULL
);

/* Creation des threads des clients si celui du magasin a reussi. */
if (! ret)
{
   printf ("Creation des threads clients !\n");
   for (i = 0; i < NB_CLIENTS; i++)
   {
      ret = pthread_create (
         & store.thread_clients [i], NULL,
         fn_clients, (void *) i);

      if (ret)
      {
         fprintf (stderr, "%s", strerror (ret));
      }
   }
}
else
{
   fprintf (stderr, "%s", strerror (ret));
}


   /* Attente de la fin des threads. */
   i = 0;
   for (i = 0; i < NB_CLIENTS; i++)
   {
      pthread_join (store.thread_clients [i], NULL);
   }
   pthread_join (store.thread_store, NULL);


   return EXIT_SUCCESS;
}
```

L'exemple de code ci-dessus utilise une variable globale ! Ici ce n'est qu'à titre d'exemple, mais je vous encourage à éviter ce genre de pratique autant que possible !

X-D. Sortie du programme ▲

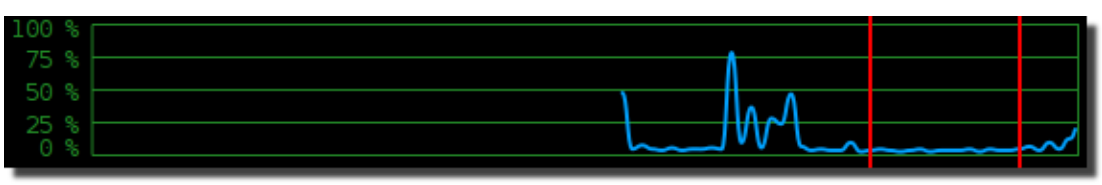
Voici la sortie du programme sur la console avec annulation utilisateur :

Sélectionnez

```
Creation du thread du magasin !
Creation des threads clients !
Client 2 prend 5 du stock, reste 15 en stock !
Client 0 prend 5 du stock, reste 10 en stock !
Client 3 prend 1 du stock, reste 9 en stock !
Client 4 prend 2 du stock, reste 7 en stock !
Client 1 prend 4 du stock, reste 3 en stock !
Client 0 prend 2 du stock, reste 1 en stock !
Client 0 prend 0 du stock, reste 1 en stock !
Remplissage du stock de 20 articles !
Client 2 prend 2 du stock, reste 18 en stock !
Client 1 prend 0 du stock, reste 18 en stock !
Client 1 prend 0 du stock, reste 18 en stock !
Client 1 prend 5 du stock, reste 13 en stock !
Client 2 prend 0 du stock, reste 13 en stock !
Client 1 prend 3 du stock, reste 10 en stock !
Client 3 prend 5 du stock, reste 5 en stock !
Client 4 prend 3 du stock, reste 2 en stock !
Client 0 prend 0 du stock, reste 2 en stock !
Remplissage du stock de 20 articles !
Client 1 prend 3 du stock, reste 17 en stock !
Client 2 prend 3 du stock, reste 14 en stock !
Client 3 prend 2 du stock, reste 12 en stock !
Client 4 prend 1 du stock, reste 11 en stock !
Client 0 prend 3 du stock, reste 8 en stock !
Client 2 prend 1 du stock, reste 7 en stock !
Client 2 prend 3 du stock, reste 4 en stock !
Client 1 prend 2 du stock, reste 2 en stock !
Client 0 prend 0 du stock, reste 2 en stock !
Client 0 prend 2 du stock, reste 0 en stock !
Remplissage du stock de 20 articles !
Client 1 prend 5 du stock, reste 15 en stock !
Client 2 prend 1 du stock, reste 14 en stock !
Client 3 prend 4 du stock, reste 10 en stock !
Client 4 prend 0 du stock, reste 10 en stock !

<CTRL-C>
```

Et la charge CPU utilisée durant le déroulement du processus et ses threads :



Charge CPU de l'exemple 3

X-E. Observations ▲

Hormis les changements importants dans notre programme d'exemple, on peut apercevoir que maintenant dans le graphique de la charge CPU, les ressources sont utilisées de façon raisonnable. Ceci est dû au fait que le thread du magasin soit mis en attente et ne tourne donc plus en continu tout le long de la durée de vie du programme, les conditions peuvent donc jouer un rôle important dans ce domaine !

On peut également voir sur la sortie console que le stock ne descend plus à une valeur en dessous de zéro.

XI. Test des threads ▲

Il existe des fonctions permettant de tester des threads entre eux. Les threads étant des types opaques, il est recommandé de les utiliser ! Nous avons à disposition, par exemple, une fonction pour réaliser un test d'égalité entre deux identifiants de threads ou même récupérer l'identifiant d'un thread. Voici ces deux fonctions qui peuvent avoir leur utilité.

XI-A. pthread_equal ▲

Sélectionnez

```
int pthread_equal (pthread_t thread1, pthread_t thread2);
```

Cette fonction teste l'égalité entre deux identifiants de threads passés en argument. La fonction renvoie **0** si les deux threads sont différents, une valeur non nulle s'ils sont identiques.

XI-B. pthread_self ▲

Sélectionnez

```
pthread_t pthread_self (void);
```

Cette fonction retourne l'identifiant du thread appelant.

XII. Conclusion ▲

Nous avons vu dans ce tutoriel comment créer des threads, que sont les mutex et les conditions et comment les utiliser. Nous avons aussi parcouru en vitesse quelques fonctions supplémentaires. Ceci n'est bien sûr qu'un petit avant-goût de la bibliothèque Pthreads, des ouvrages complets sur ce thème existent, notamment :

PThreads Programming

Il n'existe malheureusement pas de livres en français ! Vous pouvez néanmoins poser vos questions sur le forum C de developpez.com ;)

XIII. Remerciements ▲

Un très grand merci à Alp, millie, Skyrunner pour leurs avis et suggestions ainsi qu'à Guardian et RideKick pour la relecture et correction de ce tutoriel !

Vous avez aimé ce tutoriel ? Alors partagez-le en cliquant sur les boutons suivants :

Linus Torvalds annonce que Rust for Linux est susceptible d'être prêt pour la version 5.20 du noyau	Augmentation de la popularité de C#, selon l'indice Tiobe	JetBrains lance le programme d'accès anticipé (EAP) à CLion 2022.2, la deuxième mise à jour majeure de l'année de son EDI C/C++ multiplateforme	Apprendre à installer Gtk+ et Code::Blocks sous Windows, un tutoriel de Gérald Dumas

Contacter le responsable de la rubrique C

Nous contacter Participez Hébergement Publicité / Advertising Informations légales Partenaire : Hébergement Web