# Utilisation de la bibliothèque pthread

POSIX threads

par Emmanuel Delahaye (Espace personnel d'Emmanuel Delahaye)

Date de publication : 27 janvier 2008

Dernière mise à jour : 21 aout 2009

Cet article fournit les bases de l'utilisation de la bibliothèque pthread (POSIX threads) par une approche pragmatique.

*Votre avis et vos suggestions sur cet article nous intéressent !*
*Alors après votre lecture, n'hésitez pas :*

I - Systèmes multi-tâches....3
I-A - Introduction....3
I-B - Processus....3
I-B-1 - Exécution....3
I-B-2 - Création dynamique de processus....4
I-C - Processus légers....4
II - Introduction....5
II-A - Note pour Windows....5
II-B - Objectif....5
III - Hello worlds....6
IV - Données....10
IV-A - Éviter les globales 'à-la-barbare'....10
IV-B - Un seul code pour plusieurs tâches....11
IV-C - Le cas de l'écrivain et du lecteur....14
V - Synchronisation....20
VI - Résumé des types et fonctions utilisés....22
VII - Ressources....23

# I - Systèmes multi-tâches

## I-A - Introduction

Force est de constater qu'un programme courant (sauf traitements lourds comme un scan d'antivirus) passe 90 à 99% de son temps à attendre des évènements extérieurs, notamment, évidemment, les entrées (stdin, lecture fichier, attentes de données reçues sur un socket...) à moins qu'on ait voulu délibérément retarder l'exécution de telle ou telle action (sleep(), Sleep() etc.).

Donc, un processeur est, la plupart du temps, en train de ne rien faire (ou alors des boucles inutiles). Dès les débuts de l'informatique, cette constatation a amené les informaticiens à réflechir à la meilleure façon d'utiliser ce temps libre.

L'idée de répartir le temps du processeur entre plusieurs application s'est alors imposée.

Les performances des systèmes modernes reposent en grande partie sur leur possibilité de charger et exécuter plusieurs applications 'en même temps'. Cette simultanéité n'est évidemment qu'apparente, sur une machine mono-processeur. Mais elle a l'avantage d'exister et d'être efficace, et nous le constatons tous les jours sur notre PC ou notre téléphone mobile.

## I-B - Processus

Un processus est une application (ou un programme) en cours d'exécution. Un programme, est avant tout un fichier stocké sur un disque quelconque contenant du code exécutable.

Pour pouvoir être exécuté, il faut réunir un certain nombre de conditions :

- charger le programme en mémoire
- réserver de l'espace pour les données statiques
- réserver de l'espace dans la pile
- initialiser les pointeurs d'instructions et de pile
- lancer l'exécution

A un processus est donc associé un espace mémoire composé d'un programme (code exécutable) et de données (statiques et pile).

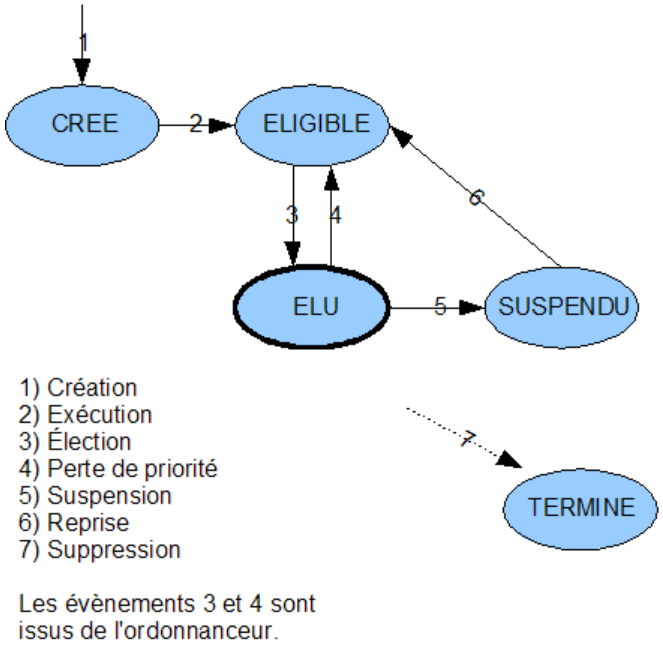
### I-B-1 - Exécution

Dans un système multi-tâches comme Unix ou Windows (mais pas MS-DOS), "lancer l'exécution" signifie rendre le processus "éligible", c'est à dire que dès que le processeur est libre, il pourra l'exécuter réellement (il devient alors "élu").

Dès que le programme rencontre une fonction qui le fait 'attendre' (suspension temporisée, attente d'une entrée etc.) le système (en fait, l'ordonnanceur, qui est le mécanisme de répartition du temps machine) en profite pour suspendre l'exécution du processus ("suspendu"). Le premier processus éligible passe alors en mode "élu". (il "prend la main" comme on dit vulgairement).

Il faut savoir qu'il existe certaines tâches dites 'immédiates' (interruptions) qui peuvent s'exécuter en temps réel à chaque instant. C'est notamment le cas pour les traitements de bas niveau de type timer (déterminer une échéance) ou clavier, réseau, disque etc. Les évènements détéctés sont transmis au système qui détermine alors à quel processus suspendu ils correspondent et si il s'agit bien d'un évènement de déblocage. ("en attente d'un <enter>", "en attente d'une trame réseau" etc). Si c'est le cas, le processus passe en état "éligible". La suite, on la connait.

En résumé, les états d'un processus sont :




Il peut y avoir des schémas plus complexes, mais le principe est là.

## I-B-2 - Création dynamique de processus

Il est possible par logiciel de créer une 'copie' ou un 'fils' du processus courant. Cela revient à créer un nouvel espace mémoire contenant une copie des données (le code reste unique).

Les valeurs des données du processus fils sont celles de l'instant de la copie (sous unixoïde, fork()). Si le processus fils modifie ses données, les données du père sont inchangées. De même, si le père modifie ses données, les données du fils sont inchangées. Les deux espaces mémoires sont complètement étanches. au point que si il fallait échanger des données entre les deux processus, il faudrait utiliser des mécanismes assez complexes (IPC : Inter Processus Communication, SHM : SHared Memory, sockets UNIX etc).

## I-C - Processus légers

Dans un processus donné, il est possible de créer un processus léger appelé aussi thread qui est un espace mémoire limité à une simple pile et un code exécutable (la fonction du thread). La mémoire statique est celle du processus courant. Si plusieurs threads sont créés, ils partagent le même espace de mémoire statique, mais ils ont chacun leur propre pile (variables locales).

Ce mécanisme est simple et permet d'écrire facilement des applications multi-tâches. Les données statiques étant partagées, leur accès est direct (mais on évite les globales, on n'est pas des barbares)

Revers de la médaille, les accès aux données étant directs, et des accès concurrents étant possibles, les données peuvent être incorrectes si des précautions ne sont pas prises. Il existe donc des mécanismes de protection qui permettent de réguler l'accès aux données (sémaphores).

La gestion des threads a été normalisée par POSIX.1. C'est l'API pthreads.

# II - Introduction

La norme POSIX.1 fournit un ensemble de primitives permettant de réaliser des processus légers. Contrairement aux processus habituels, la mémoire est commune. Ces processus légers sont communément appelés 'threads' ou 'tâches'. Selon l'implémentation, l'ensemble des primitives est, par exemple, regroupé dans la bibliothèque libpthread.a (gcc) qu'il faut bien sûr penser à ajouter au projet (-lpthread).

## II-A - Note pour Windows

Les pthreads sont disponibles sur **ce site**.

## II-B - Objectif

Le but de cet article est de fournir les bases de l'utilisation de la bibliothèque pthread par une approche pragmatique.

## III - Hello worlds

Il est simple de réaliser la création, l'exécution et la fin de 2 tâches 'concurrentes'[1]. Pour cela, on utilise le type opaque (ADT) pthread_t qui va contenir le contexte de chaque tâche, et une fonction de création de tâche pthread_create() qui va associer la tâche avec son contexte, et informer l'ordonnanceur du système de son existence. L'ordonnanceur lancera la tâche dès que possible

[1] En fait, la concurrence est une notion relative. Sur un système mono-processeur, l'exécution des tâches se fait selon des critères de temps (time slicing) ou de suspensions.

```
/* ATTENTION CODE ERRONE */

#include <stdio.h>

#include <pthread.h>

static void *task_a (void *p_data)
{
   puts ("Hello world A");

   (void) p_data;
   return NULL;
}

static void *task_b (void *p_data)
{
   puts ("Hello world B");

   (void) p_data;
   return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;

   puts ("main init");

   pthread_create (&ta, NULL, task_a, NULL);
   pthread_create (&tb, NULL, task_b, NULL);

   puts ("main end");

   return 0;
}
```

La sortie est, par exemple, celle-ci. Attention, l'ordre dans lequel l'exécution des tâches se fait n'est pas défini (normal, puisque dans l'idéal, elles sont concurrentes).

```
main init
main end
Hello world A
Hello world B
```

Il est important de bien interpréter les lignes produites. Dans cet exemple, il y a un processus (l'application) et 3 tâches :

- main()
- task_a()
- task_b()

Il est clair, à la vue des lignes produites, que la tâche main() se termine avant que les autres ne soient lancées. Cela signifie que, étant donné que le contexte des tâches n'existe plus, le comportement des tâches devient indéterminé. C'est pourquoi il existe la fonction de synchronisation pthread_join() qui permet de suspendre l'exécution de la tâche courante en attendant la fin d'une tâche déterminée.

```
#include <stdio.h>

#include <pthread.h>

static void *task_a (void *p_data)
{
   puts ("Hello world A");

   (void) p_data;
   return NULL;
}

static void *task_b (void *p_data)
{
   puts ("Hello world B");

   (void) p_data;
   return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;

   puts ("main init");

   pthread_create (&ta, NULL, task_a, NULL);
   pthread_create (&tb, NULL, task_b, NULL);

#if 1
   pthread_join (ta, NULL);
   pthread_join (tb, NULL);
#endif

   puts ("main end");

   return 0;
}
```

La sortie donne maintenant ceci:

```
main init
Hello world A
Hello world B
main end
```

Dans la réalité, une tâche est généralement une boucle (infinie ou non).

```
#include <stdio.h>

#include <pthread.h>

static void *task_a (void *p_data)
{
   int i;

   for (i = 0; i < 5; i++)
   {
      printf ("Hello world A (%d)\n", i);
   }

   (void) p_data;
```

```
   return NULL;
}

static void *task_b (void *p_data)
{
   int i;

   for (i = 0; i < 7; i++)
   {
      printf ("Hello world B    (%d)\n", i);
   }
   (void) p_data;
   return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;

   puts ("main init");

   pthread_create (&ta, NULL, task_a, NULL);
   pthread_create (&tb, NULL, task_b, NULL);

#if 1
   pthread_join (ta, NULL);
   pthread_join (tb, NULL);
#endif

   puts ("main end");

   return 0;
}
```

Les informations produites sont variables. Par exemple :

```
main init
Hello world A (0)
Hello world B    (0)
Hello world B    (1)
Hello world A (1)
Hello world A (2)
Hello world B    (2)
Hello world B    (3)
Hello world A (3)
Hello world A (4)
Hello world B    (4)
Hello world B    (5)
Hello world B    (6)
main end
```

ou

```
main init
Hello world A (0)
Hello world B    (0)
Hello world B    (1)
Hello world A (1)
Hello world A (2)
Hello world B    (2)
Hello world B    (3)
Hello world B    (4)
Hello world A (3)
Hello world A (4)
Hello world B    (5)
Hello world B    (6)
main end
```

mais on constate que le mécanisme de rendez-vous fonctionne et que les deux boucles sont terminées correctement. Sans le mécanisme de synchronisation :

```
#if 0
   pthread_join (ta, NULL);
   pthread_join (tb, NULL);
#endif
```

on aurait, par exemple, obtenu ceci, ce qui n'est pas loin d'une catastrophe...

```
main init
main end
Hello world A (0)
Hello world A (1)
Hello world B    (0)
Hello world B    (1)
Hello world B    (2)
Hello world A (2)
Hello world A (3)
Hello world B    (3)
Hello world B    (4)
Hello world A
```

# IV - Données

L'application étant formée d'un seul processus, les données sont communes à toutes les tâches. Cela signifie qu'une donnée est partageable directement entre les processus. Le terme 'données communes' recouvre comme il se doit :

- les données statiques
- les données allouées
- les données locales

## IV-A - Éviter les globales 'à-la-barbare'

Il est possible d'utiliser le 4ème paramètre de pthread_create() pour passer l'adresse d'un objet quelconque à une tâche (variable simple, tableau, structure) au moment de la création de la tâche. Il est bien évident que la durée de vie de cette variable doit être supérieure ou égale à la durée de vie de la tâche qui l'utilise.

```
#include <stdio.h>

#include <pthread.h>

static void *task_a (void *p_data)
{
   if (p_data != NULL)
   {
      /* recuperer le contexte applicatif */
      char const *s = p_data;
      int i;

      for (i = 0; i < 5; i++)
      {
         printf ("'%s' (%d)\n", s, i);
      }
   }
   return NULL;
}

static void *task_b (void *p_data)
{
   if (p_data != NULL)
   {
      /* recuperer le contexte applicatif */
      char const *s = p_data;
      int i;

      for (i = 0; i < 7; i++)
      {
         printf ("'%s'     (%d)\n", s, i);
      }
   }
   return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;

   puts ("main init");

/* La chaine etant definie const sur ma machine, je force en (char*).
 * Dans la tâche, 's' est bien de type 'char const *'
 */
   pthread_create (&ta, NULL, task_a, (char*)"Tache A");
   pthread_create (&tb, NULL, task_b, (char*)"Tache B");

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);
```

```
   puts ("main end");

   return 0;
}
```

qui produit par exemple :

```
main init
'Tache A' (0)
'Tache B'      (0)
'Tache B'      (1)
'Tache A' (1)
'Tache A' (2)
'Tache A' (3)
'Tache B'      (2)
'Tache B'      (3)
'Tache B'      (4)
'Tache A' (4)
'Tache B'      (5)
'Tache B'      (6)
main end
```

## IV-B - Un seul code pour plusieurs tâches

Il est possible de lancer plusieurs fois la même tâche :

```
#include <stdio.h>

#include <pthread.h>

static void *task (void *p_data)
{
   int i;

   for (i = 0; i < 5; i++)
   {
      printf ("Hello world (%d)\n", i);
   }

   (void) p_data;
   return NULL;
}


int main (void)
{
   pthread_t ta;
   pthread_t tb;

   puts ("main init");

   pthread_create (&ta, NULL, task, NULL);
   pthread_create (&tb, NULL, task, NULL);

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);

   puts ("main end");

   return 0;
}
```

Ce qui donne, par exemple :

```
main init
Hello world (0)
```

```
Hello world (1)
Hello world (2)
Hello world (3)
Hello world (4)
Hello world (0)
Hello world (1)
Hello world (2)
Hello world (3)
Hello world (4)
main end

Process returned 0 (0x0)   execution time : 0.041 s
Press any key to continue.
```

Evidemment, on ne sait plus différencier les traitements, et on ne sait pas qui fait quoi. Mais grace au paramètre 'données', on va pouvoir personnaliser le fonctionnement des tâches :

- Identificateur ("A" ou "B")
- Nombre de tours de boucles
- etc.

Comme il y a plusieurs paramètres, on définit une structure, on crée une instance par tâche avec les paramètres voulus et on passe l'adresse de la structure au moment de la création de la tâche. Ensuite la tâche récupère l'adresse en paramètre. Il suffit alors d'initialiser un pointeur du bon type avec l'adresse, et on a alors accès aux champs.

Par exemple :

```c
#include <stdio.h>

#include <pthread.h>

struct data
{
   char const *id;
   int n;
};

static void *task (void *p_data)
{
   if (p_data != NULL)
   {
      struct data *p = p_data;
      int i;

      for (i = 0; i < p->n; i++)
      {
         printf ("Hello world %s (%d)\n", p->id, i);
      }
   }

   return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;
   struct data data_a = { "A", 5 };
   struct data data_b = { "B", 7 };

   puts ("main init");

   pthread_create (&ta, NULL, task, &data_a);
   pthread_create (&tb, NULL, task, &data_b);

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);
```

```
   puts ("main end");

   return 0;
}
```

Qui donne, par exemple (Vista SP1)

```
main init
Hello world A (0)
Hello world A (1)
Hello world A (2)
Hello world A (3)
Hello world A (4)
Hello world B (0)
Hello world B (1)
Hello world B (2)
Hello world B (3)
Hello world B (4)
Hello world B (5)
Hello world B (6)
main end

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

*Les essais on été faits sous Windows Vista SP1, qui ne semble pas pratiquer le time slicing... Chaque boucle va donc jusqu'au bout...*

Pour un fonctionnement plus réaliste, on peut introduire une suspension dans les traitements :

```
#include <stdio.h>

#include <pthread.h>

/* http://www.bien-programmer.fr/clib/psleep/ */
#include "psleep/inc/psleep.h"

struct data
{
   char const *id;
   int n;
};

static void *task (void *p_data)
{
   if (p_data != NULL)
   {
      struct data *p = p_data;
      int i;

      for (i = 0; i < p->n; i++)
      {
         printf ("Hello world %s (%d)\n", p->id, i);
         msleep(1);
      }
   }

   return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;
   struct data data_a = { "A", 5 };
   struct data data_b = { "B", 7 };
```

```
   puts ("main init");

   pthread_create (&ta, NULL, task, &data_a);
   pthread_create (&tb, NULL, task, &data_b);

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);

   puts ("main end");

   return 0;
}
```

Ce qui donne, par exemple (Vista SP1)

```
main init
Hello world B (0)
Hello world A (0)
Hello world B (1)
Hello world A (1)
Hello world B (2)
Hello world A (2)
Hello world A (3)
Hello world B (3)
Hello world B (4)
Hello world A (4)
Hello world B (5)
Hello world B (6)
main end

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

*msleep() n'est pas standard. C'est une macro que j'ai inventée qui permet d'écrire des suspensions standards sous Windows et unixoides. Les détails sont **ici**.*

## IV-C - Le cas de l'écrivain et du lecteur

Soit une variable partagée qui est modifiée par une tâche, et lue par une autre. Le code 'naïf' suivant :

```
#include <stdio.h>

#include <pthread.h>

struct shared
{
   int data;
};

struct data
{
   /* paramètres */
   int nb;
   char const *sid;

   /* contexte partage' */
   struct shared *psh;
};

static void *task_w (void *p)
{
   if (p != NULL)
   {
      /* recuperer le contexte applicatif */
      struct data *p_data = p;
      int i;
```

```
      for (i = 0; i < p_data->nb; i++)
      {
         int x = p_data->psh->data;
         x++;
         p_data->psh->data = x;

         printf ("'%s' (%d) data <- %d\n", p_data->sid, i, p_data->psh->data);
      }
   }
   return NULL;
}

static void *task_r (void *p)
{
   if (p != NULL)
   {
      /* recuperer le contexte applicatif */
      struct data *p_data = p;
      int i;

      for (i = 0; i < p_data->nb; i++)
      {
         printf ("                          "
                 "'%s' (%d) data == %d\n", p_data->sid, i, p_data->psh->data);
      }
   }
   return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;

   struct shared sh =
   {
      .data = 0,
   };

   struct data da =
   {
      .nb = 5,
      .sid = "Writer",
      .psh = &sh,
   };

   struct data db =
   {
      .nb = 7,
      .sid = "Reader",
      .psh = &sh,
   };

   puts ("main init");

   pthread_create (&ta, NULL, task_w, &da);
   pthread_create (&tb, NULL, task_r, &db);

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);

   puts ("main end");

   return 0;
}
```

provoque des aberrations comme

```
main init
'Writer' (0) data <- 1
                        'Reader' (0) data == 1
```

```
                         'Reader' (1) data == 2
'Writer' (1) data <- 2
'Writer' (2) data <- 3
                         'Reader' (2) data == 2
                         'Reader' (3) data == 4
'Writer' (3) data <- 4
'Writer' (4) data <- 5
                         'Reader' (4) data == 4
                         'Reader' (5) data == 5
                         'Reader' (6) data == 5
main end
```

où l'on constate que les valeurs lues sont parfois fausses par rapport aux valeurs réelles.

Pour se prémunir contre ce problème, on doit évaluer quelle est la 'zone critique', et ensuite la protéger avec un 'sémaphore'. C'est tout simplement un mécanisme qui va empêcher une autre tâche d'interrompre la section critique. Dans l'environnement pthread, ce mécanisme est appelé mutex. Il met en oeuvre le type opaque pthread_mutex_t et les fonctions pthread_mutex_lock() et pthread_mutex_unlock().

Voici un exemple de protection de la section critique de l'écrivain (empêcher le lecteur d'accéder en lecture pendant l'écriture).

```c
#include <stdio.h>

#include <pthread.h>

struct shared
{
   int data;
   pthread_mutex_t mut;
};

struct data
{
   /* paramètres */
   int nb;
   char const *sid;

   /* contexte partage' */
   struct shared *psh;
};


static void *task_w (void *p)
{
   if (p != NULL)
   {
      struct data *p_data = p;
      int i;

      for (i = 0; i < p_data->nb; i++)
      {

/* debut de la zone critique */
         pthread_mutex_lock (&p_data->psh->mut);
         {
            int x = p_data->psh->data;
            x++;
            p_data->psh->data = x;
         }
         printf ("'%s' (%d) data <- %d\n", p_data->sid, i, p_data->psh->data);
         pthread_mutex_unlock (&p_data->psh->mut);
/* fin de la zone critique */

      }
   }
   return NULL;
}
```

```
static void *task_r (void *p)
{
   if (p != NULL)
   {
      struct data *p_data = p;
      int i;

      for (i = 0; i < p_data->nb; i++)
      {
         printf ("                          "
               "'%s' (%d) data == %d\n", p_data->sid, i, p_data->psh->data);
      }
   }
   return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;

   struct shared sh =
   {
      .data = 0,
      .mut = PTHREAD_MUTEX_INITIALIZER,
   };

   struct data da =
   {
      .nb = 5,
      .sid = "Writer",
      .psh = &sh,
   };

   struct data db =
   {
      .nb = 7,
      .sid = "Reader",
      .psh = &sh,
   };

   puts ("main init");

   pthread_create (&ta, NULL, task_w, &da);
   pthread_create (&tb, NULL, task_r, &db);

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);

   puts ("main end");

   return 0;
}
```

L'amélioration est flagrante, mais insuffisante :

```
main init
'Writer' (0) data <- 1
'Writer' (1) data <- 2
'Writer' (2) data <- 3
                         'Reader' (0) data == 3
                         'Reader' (1) data == 4
'Writer' (3) data <- 4
'Writer' (4) data <- 5
                         'Reader' (2) data == 4
                         'Reader' (3) data == 5
                         'Reader' (4) data == 5
                         'Reader' (5) data == 5
                         'Reader' (6) data == 5
main end
```

Il faut aussi protéger la lecture, c'est à dire empêcher l'écrivain d'écrire pendant la lecture :

```
#include <stdio.h>

#include <pthread.h>

struct shared
{
   int data;
   pthread_mutex_t mut;
};

struct data
{
   /* paramètres */
   int nb;
   char const *sid;

   /* contexte partage' */
   struct shared *psh;
};

static void *task_w (void *p)
{
   if (p != NULL)
   {
      struct data *p_data = p;
      int i;

      for (i = 0; i < p_data->nb; i++)
      {

/* debut de la zone critique */
         pthread_mutex_lock (&p_data->psh->mut);
         {
            int x = p_data->psh->data;
            x++;
            p_data->psh->data = x;
         }
         printf ("'%s' (%d) data <- %d\n", p_data->sid, i, p_data->psh->data);
         pthread_mutex_unlock (&p_data->psh->mut);
/* fin de la zone critique */

      }
   }
   return NULL;
}

static void *task_r (void *p)
{
   if (p != NULL)
   {
      struct data *p_data = p;
      int i;

      for (i = 0; i < p_data->nb; i++)
      {
/* debut de la zone critique */
         pthread_mutex_lock (&p_data->psh->mut);
         printf ("                         "
               "'%s' (%d) data == %d\n", p_data->sid, i, p_data->psh->data);
         pthread_mutex_unlock (&p_data->psh->mut);
/* fin de la zone critique */
      }
   }
   return NULL;
}

int main (void)
{
   pthread_t ta;
```

```
    pthread_t tb;

    struct shared sh =
    {
        .data = 0,
        .mut = PTHREAD_MUTEX_INITIALIZER,
    };

    struct data da =
    {
        .nb = 5,
        .sid = "Writer",
        .psh = &sh,
    };

    struct data db =
    {
        .nb = 7,
        .sid = "Reader",
        .psh = &sh,
    };

    puts ("main init");

    pthread_create (&ta, NULL, task_w, &da);
    pthread_create (&tb, NULL, task_r, &db);

    pthread_join (ta, NULL);
    pthread_join (tb, NULL);

    puts ("main end");

    return 0;
}
```

Le comportement est maintenant conforme aux attentes :

```
main init
'Writer' (0) data <- 1
                          'Reader' (0) data == 1
'Writer' (1) data <- 2
                          'Reader' (1) data == 2
'Writer' (2) data <- 3
                          'Reader' (2) data == 3
'Writer' (3) data <- 4
                          'Reader' (3) data == 4
'Writer' (4) data <- 5
                          'Reader' (4) data == 5
                          'Reader' (5) data == 5
                          'Reader' (6) data == 5
main end
```

# V - Synchronisation

Nous avons pu constater qu'en environnement multi-tâches, l'ordre d'exécution des instructions de 2 tâches différentes était indéterminé. C'est pourquoi il existe un mécanisme permettant la synchronisation. Par exemple, bloquer la tâche courante en attente d'un évènement ou la débloquer dès le déclenchement de cet évènement.

Dans l'environnement pthread, on parle de 'condition'. L'objet est pthread_cond_t, les fonctions sont pthread_cond_wait() et pthread_cond_signal().

Dans l'exemple initial, les deux tâches s'exécutaient de façon désordonnée. Voici un mécanisme de synchronisation de la tâche B par la tâche A. Il a fallu ajouter une suspension dans la tâche A afin de laisser du temps à B de s'exécuter.

```
#include <stdio.h>

#include <pthread.h>

/* http://www.bien-programmer.fr/clib/psleep/ */
#include "psleep/inc/psleep.h"

struct shared
{
   pthread_mutex_t mut;
   pthread_cond_t synchro;
};

struct data
{
   /* paramètres */
   int nb;
   char const *sid;

   /* contexte partage' */
   struct shared *psh;
};

static void *task_a (void *p)
{
   if (p != NULL)
   {
      struct data *p_data = p;
      struct shared *psh = p_data->psh;
      int i;

      for (i = 0; i < p_data->nb; i++)
      {
         pthread_mutex_lock (&psh->mut);
         printf ("'%s' (%d)\n", p_data->sid, i);
         pthread_cond_signal (&psh->synchro);
         pthread_mutex_unlock (&psh->mut);
         msleep (1000);
      }
   }
   return NULL;
}

static void *task_b (void *p)
{
   if (p != NULL)
   {
      struct data *p_data = p;
      struct shared *psh = p_data->psh;
      int i;

      for (i = 0; i < p_data->nb; i++)
      {
         pthread_mutex_lock (&psh->mut);
         pthread_cond_wait (&psh->synchro, &psh->mut);
```

```
            printf ("                          "
                    "'%s' (%d)\n", p_data->sid, i);
            pthread_mutex_unlock (&psh->mut);
         }
      }
      return NULL;
}

int main (void)
{
   pthread_t ta;
   pthread_t tb;

   struct shared sh =
   {
      .mut = PTHREAD_MUTEX_INITIALIZER,
      .synchro = PTHREAD_COND_INITIALIZER,
   };

   struct data da =
   {
      .nb = 5,
      .sid = "task A",
      .psh = &sh,
   };

   struct data db =
   {
      .nb = 4,
      .sid = "Task B",
      .psh = &sh,
   };

   puts ("main init");

   pthread_create (&ta, NULL, task_a, &da);
   pthread_create (&tb, NULL, task_b, &db);

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);

   puts ("main end");

   return 0;
}
```

## VI - Résumé des types et fonctions utilisés

| Types et fonctions | Description |
|---|---|
| pthread_t | Contexte de tâche |
| pthread_create() | Création d'une tâche |
| pthread_join() | Attente de fin de tâche |
| pthread_mutex_t | Sémaphore d'exclusion mutuelle |
| pthread_mutex_lock() | Début de zone critique |
| pthread_mutex_unlock() | Fin de zone critique |
| pthread_cond_t | Condition |
| pthread_cond_wait() | Mise en attente |
| pthread_cond_signal() | Déclenchement |

*Les essais ont été faits sous Windows 98 avec Dev-C++ et sa bibliothèque pthread. L'ordonnanceur de Windows étant préemptif, les aberrations de fonctionnement sont claires et évidentes. Les essais réalisés sous Linux embarqué (Kernel 2.4 pour Power PC) sont moins parlants, car sur ma configuration de test, l'ordonnanceur n'est pas préemptif. Il semble aussi que, sous Windows XP, l'ordonnanceur ne soit pas préemptif (time slicing), mais coopératif... A moins que j'interprète mal le fonctionnement... Remarques bienvenues.*

Les remarques reçues font état des faits suivants :

- L'ordonnanceur de GNU/Linux 2.4 est soit coopératif, soit préemptif selon une option de compilation du noyau
- L'ordonnanceur de GNU/Linux 2.6 est préemptif
- L'ordonnanceur de MS/Windows 98 est coopératif (blocage possible par une 'boucle blanche')
- L'ordonnanceur de MS/Windows NT/XP est préemptif (évènements + time slicing)

## VII - Ressources

Pour approfondir, je recommande cet excellent site **POSIX Threads Programming**.