Multi-threading sous LINUX

Pierre Ficheux (pficheux@com1.fr)

Avril 1999

---

0. Résumé

Cet article est une introduction à la programmation *multi-threads* sous LINUX. Les exemples de programmation utilisent la bibliothèque LinuxThreads disponible en standard sur la majorité des distributions LINUX récentes. ces exemples sont disponibles en téléchargement sur http://www.com1.fr/~pficheux/articles/lmf/threads/examples.tar.gz

La lecture de cet article nécessite une assez bonne compréhension de la syntaxe du langage C...

1. Qu'est-ce que le multi-threading ?

Les programmeurs LINUX et plus généralement UNIX sont depuis longtemps habitués aux fonctionnalités *multi-taches* de leur systéme préféré. Tous ceux qui se sont frottés un tant soit peu à la programmation *systéme* savent qu'il est aisé sous UNIX de créer des *processus* fils à partir d'un processus existant en utilisant l'appel systéme *fork*, comme le montre le petit exemple de code ci-dessous:

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

int i;

main (int ac, char **av)
{
  int pid;

  i = 1;

  if ((pid = fork()) == 0) {
```

```
    /* Dans le fils */
    printf ("Je suis le fils, pid = %d\n", getpid());
    sleep (2);
    printf ("Fin du fils, i = %d !\n", i);
    exit (0);
  }
  else if (pid > 0) {
    /* Dans le pere */
    printf ("Je suis le pere, pid = %d\n", getpid());
    sleep (1);

    /* Modifie la variable */
    i = 2;
    printf ("le pere a modifie la variable a %d\n", i);

    sleep (3);
    printf ("Fin du pere, i = %d !\n", i);
    exit (0);
  }
  else {
    /* Erreur */
    perror ("fork");
    exit (1);
  }
}
```

qui donne à l'exécution:

```
pierre@mmxpf % ./fork&
[1] 367
pierre@mmxpf % Je suis le pere, pid = 367
Je suis le fils, pid = 368
le pere a modifie la variable a 2
Fin du fils, i = 1 !
Fin du pere, i = 2 !
```

Les limites du *fork* apparaissent d'ores et déjà lorsqu'il s'agit de partager des variables entre un processus pére et son fils. Comme on le voit dans le petit exemple ci-dessus, la variable globale *i*, modifiée par le pére a toujours l'ancienne valeur dans le fils. Ceci est le comportement normal du *fork* qui duplique le contexte courant lors de la création d'un processus fils.
En plus d'empécher le partage de variables, la création d'un nouveau contexte est pénalisante au niveau performances. Il en est de méme pour le changement de contexte (context switch), lors du passage d'un processus à un autre.

Un *thread* ressemble fortement à un processus fils classique à la différence qu'il partage beaucoup plus de données avec le processus qui l'a créé:

- Les variables globales
- Les variables statiques locales
- Les descripteurs de fichiers (file descriptors)

Le *multi-threading* est donc une technique de programmation permettant de profiter des avantages (et aussi de certaines contraintes) de l'utilisation des *threads*.

2. Les bibliothèques de threads

De nombreux systémes d'exploitation permettent aujourd'hui la programmation par threads: Solaris 5.x de SUN, Windows95/98/NT et bien d'autres (dont LINUX). Dans le cas de Solaris, la bibliothèque de threads disponible est conforme à la norme *POSIX 1003.1c* ce qui assure une certaine portabilité de l'applicatif en cas de portage vers un autre systéme. Dans le cas des systémes Microsoft, la bibliothèque utilisée est bien entendu non conforme à cette norme POSIX !

Il existe aujourd'hui diverses bibliothèques permettant de manipuler des threads sous LINUX. On dénombre deux principaux types d'implémentations de threads:

- Au niveau utilisateur (user-level). A ce moment la, la gestion des threads est entiérement faite dans l'espace utilisateur.
- Au niveau noyau (kernel-level). Dans ce cas, les threads sont directement gérés par le noyau.

Dans ce dernier cas, la base de l'implémentation est entre-autres l'appel systéme *clone*, également utilisé pour la création de processus classiques:

```
NAME
       clone - create a child process

SYNOPSIS
       #include <linux/sched.h>
       #include <linux/unistd.h>

       pid_t clone(void *sp, unsigned long flags)

DESCRIPTION
       clone  is  an  alternate  interface  to  fork,  with  more
       options.  fork is equivalent to clone(0, SIGCLD|COPYVM).
```

La bibliothèque LinuxThreads développée par Xavier Leroy (Xavier.Leroy@inria.fr) est une excellente implémentation de la norme *POSIX 1003.1c*. Cette bibliothèque est basée sur l'appel systéme *clone*. Je ne saurais trop vous conseiller d'utiliser ce produit, ce que nous ferons dans la suite des exemples présentés dans cet article.

Pour information, cette bibliothèque est livrée en standard sur les distributions RedHat 5.

3. Comment créer des threads sous LINUX ?

Voici un petit exemple de programme utilisant deux threads d'affichage:

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

void *my_thread_process (void * arg)
{
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
}

main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;

  if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, my_thread_process, "2") < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }
```

```
  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

La fonction *pthread_create* permet de créer le thread et de l'associer é la fonction *my_thread_process*. On notera que le paramétre *void *arg* est passé au thread lors de sa création. Aprés création des deux threads, le programme principal attend la fin des threads en utilisant la fonction *pthread_join*. Aprés compilation de ce programme par la commande:

```
cc -D_REENTRANT -o thread1 thread1.c -lpthread
```

Il donne à l'exécution:

```
pierre@mmxpf % ./thread1
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
```

4. Partages des données et synchronisation

4.1 Les MUTEX

Le partage de données nécessite parfois (méme souvent) d'utiliser des techniques permettant de protéger à un instant donné une variable partagée par plusieurs threads. Imaginons un simple tableau d'entier rempli par un thread (lent) et lu par un autre (plus rapide). Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher sont contenu. Pour cela, on peut utiliser le systéme des *MUTEX* (MUTual EXclusion) afin de protéger le tableau pendant le temps de son remplissage:

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

static pthread_mutex_t my_mutex;
static int tab[5];
```

```
void *read_tab_process (void * arg)
{
  int i;

  pthread_mutex_lock (&my_mutex);
  for (i = 0 ; i != 5 ; i++)
    printf ("read_process, tab[%d] vaut %d\n", i, tab[i]);
  pthread_mutex_unlock (&my_mutex);
  pthread_exit (0);
}

void *write_tab_process (void * arg)
{
  int i;

  pthread_mutex_lock (&my_mutex);
  for (i = 0 ; i != 5 ; i++) {
    tab[i] = 2 * i;
    printf ("write_process, tab[%d] vaut %d\n", i, tab[i]);
    sleep (1); /* Relentit le thread d'ecriture... */
  }
  pthread_mutex_unlock (&my_mutex);
  pthread_exit (0);
}

main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;

  pthread_mutex_init (&my_mutex, NULL);

  if (pthread_create (&th1, NULL, write_tab_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, read_tab_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }

  (void)pthread_join (th1, &ret);
```

```
  (void)pthread_join (th2, &ret);
}
```

La fonction *pthread_mutex_lock* verrouille le MUTEX pendant la durée du remplissage du tableau. Le thread de lecture est contraint d'attendre l'appel é *pthread_mutex_unlock* pour verrouiller à son tour le MUTEX et lire le tableau correct. A l'exécution on obtient:

```
pierre@mmxpf % ./thread2
write_process, tab[0] vaut 0
write_process, tab[1] vaut 2
write_process, tab[2] vaut 4
write_process, tab[3] vaut 6
write_process, tab[4] vaut 8
read_process, tab[0] vaut 0
read_process, tab[1] vaut 2
read_process, tab[2] vaut 4
read_process, tab[3] vaut 6
read_process, tab[4] vaut 8
```

Si par malheur on n'utilisait par le MUTEX, on obtiendrait par contre:

```
pierre@mmxpf % ./thread2
write_process, tab[0] vaut 0
read_process, tab[0] vaut 0
read_process, tab[1] vaut 0
read_process, tab[2] vaut 0
read_process, tab[3] vaut 0
read_process, tab[4] vaut 0
write_process, tab[1] vaut 2
write_process, tab[2] vaut 4
write_process, tab[3] vaut 6
write_process, tab[4] vaut 8
```

4.2 Les sémaphores POSIX

La bibliothèque LinuxThreads fournit également une implémentation des sémaphores *POSIX 1003.1b*. L'utilisation de sémaphores permet aussi la synchronisation entre plusieurs threads. Voici un exemple simple:

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

```
static sem_t my_sem;
int the_end;

void *thread1_process (void * arg)
{
  while (!the_end) {
    printf ("Je t'attend !\n");
    sem_wait (&my_sem);
  }

  printf ("OK, je sors !\n");
  pthread_exit (0);
}

void *thread2_process (void * arg)
{
  register int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("J'arrive %d !\n", i);
    sem_post (&my_sem);
    sleep (1);
  }

  the_end = 1;
  sem_post (&my_sem); /* Pour debloquer le dernier sem_wait */
  pthread_exit (0);
}

main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;

  sem_init (&my_sem, 0, 0);

  if (pthread_create (&th1, NULL, thread1_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, thread2_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
```

```
  }

  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

Dans cet exemple, le thread numéro 1 attend le thread 2 par l'intermédiaire d'un sémaphore. Aprés compilation on obtient la sortie suivante:

```
pierre@mmxpf % ./thread3
Je t'attend !
J'arrive 0 !
Je t'attend !
J'arrive 1 !
Je t'attend !
J'arrive 2 !
Je t'attend !
J'arrive 3 !
Je t'attend !
J'arrive 4 !
Je t'attend !
OK, je sors !
```

5. Mode de création des threads: JOINABLE ou DETACHED

Dans les exemples précédents, les threads sont créés en mode *JOINABLE*, c'est à dire que le processus qui a créé le thread attend la fin de celui-ci en restant bloqué sur l'appel à *pthread_join*. Lorsque le thread se termine, les ressources mémoire du thread sont libérées grâce à l'appel à *pthread_join*. Si cet appel n'est pas effectué, la mémoire n'est pas libérée et il s'en suit une *fuite de mémoire*. Pour éviter un appel systématique à *pthread_join* (qui peut parfois être contraignant dans certaines applications), on peut créer le thread en mode *DETACHED*. Dans ce cas la, la mémoire sera correctement libérée à la fin du thread.

Pour cela il suffit d'ajouter le code suivant:

```
  pthread_attr_t thread_attr;

  if (pthread_attr_init (&thread_attr) != 0) {
    fprintf (stderr, "pthread_attr_init error");
    exit (1);
  }

  if (pthread_attr_setdetachstate (&thread_attr, PTHREAD_CREATE_DETACHED) != 0) {
```

```
    fprintf (stderr, "pthread_attr_setdetachstate error");

    exit (1);
  }
```

puis de créer les threads avec des appels du type:

```
  if (pthread_create (&th1, &thread_attr, thread1_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
```

6. Destruction de thread: cancellation

Les exemples ci-dessus utilisait la fonction *pthread_exit* pour la destruction d'un thread (en fait le thread se détruisait tout seul). Il existe un mécanisme dans lequel un thread peut en détruire en autre à condition que ce dernier ait validé cette possibilité. Le comportement par défaut est de type *décalé* (deferred). Lorsqu'on envoie une requête de destruction d'un thread, celle-ci n'est exécutée que lorsque ce thread passe par un *cancellation point* comme par exemple l'appel à la fonction *pthread_testcancel*. Voici un petit exemple:

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

void *my_thread_process (void * arg)
{
  int i;

  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
    pthread_testcancel ();
  }
}

main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;
```

```
  if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  sleep (2);
  if (pthread_cancel (th1) != 0) {
    fprintf (stderr, "pthread_cancel error for thread 1\n");
    exit (1);
  }

  (void)pthread_join (th1, &ret);
}
```

Le thread un accepte les destruction par l'appel à la fonction *pthread_setcancelstate* au début du thread puis teste les demandes de destruction par *pthread_testcancel*. Au bout de deux secondes, le thread est détruit par un appel à *pthread_cancel* dans le programme principal. Le résultat à l'éxécution est le suivant:

```
pierre@mmxpf % ./thread4
Thread 1: 0
Thread 1: 1
```

Pour éviter l'utilisation des *cancellation points*, on peut indiquer que la destruction est en mode *asynchrone* en modifiant le code du thread de la maniére suivante:

```
void *my_thread_process (void * arg)
{
  int i;

  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
}
```

7. Debug d'un programme multi-thread sous LINUX

Les derniéres versions de *gdb* et de la *glibc* (packages distribués avec la RedHat 5.2) permettent de debugger un programme LINUX utilisant du multi-threading. Plus d'infos sont disponibles sur la page WWW de la bibliothèque LinuxThreads (voir bibliographie).

Voici un petit exemple de session *gdb* sur le programme d'exemple *thread1*.

```
(gdb) b main
Breakpoint 1 at 0x8048622: file thread1.c, line 22.
```

On a posé un point d'arrêt dans le programme principal avant la création des threads.

```
(gdb) n
[New Thread 25572]
[New Thread 25571]
[New Thread 25573]
Thread 1: 0
Thread 1: 1
Thread 1: 2
Thread 1: 3
Thread 1: 4
(gdb) info threads
  3 Thread 25573  0x4007a921 in __libc_nanosleep ()
* 2 Thread 25571  main (ac=1, av=0xbffffca0) at thread1.c:27
  1 Thread 25572  0x4008b2de in __select ()
```

L'action sur *next* exécute la fonction *pthread_create* qui provoque la création du thread 1. La commande *info threads* permet de connaitre la liste des tous les threads associés à l'exécution du programme. Le thread courant est indiqué par l'étoile, le numéro du thread est indiqué en deuxiéme colonne (ici 1, 2, 3).

```
(gdb) thread 1
[Switching to Thread 25654]
#0  0x4008b2de in __select ()
```

On peut passer d'un thread à l'autre en utilisant la commande *thread numéro-du-thread.*

8. Conclusion et bibliographie

L'utilisation du multi-threading permet de faciliter la programmation d'un grand nombres d'applications de type serveur ou multimédia, tout en améliorant les fonctionnalités du programme par rapport à une solution classique basée sur l'utilisation des créations de processus (fork). Les pointeurs

suivants vous seront utiles si vous vous lancez dans le multi-threading:

- La bibliothèque LinuxThreads sur http://pauillac.inria.fr/~xleroy/linuxthreads
- Le site "Programming POSIX threads" sur http://www.humanfactor.com/pthreads
- Si vous voulez porter vos applicatifs sur Win32, la bibliothèque "POSIX Threads (pthreads) for Win32" sur http://sourceware.cygnus.com/pthreads-win32
- La FAQ du groupe de discussion comp.programming.threads sur http://www.serpentine.com/~bos/threads-faq