Multi-threading sous LINUX

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

Les programmeurs LINUX / UNIX sont habitués aux fonctionnalités *multi-taches*. La *programmation système* permet de créer des *processus* fils à partir d'un processus existant en utilisant l'appel système *fork*, comme le montre le petit exemple de code ci-dessous :

fork.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4
 5 int i;
 6
 7 int main(int ac, char **av)
 8 {
 9     int pid;
10
11     i = 1;
12
13     if ((pid = fork()) == 0) /* Processus le fils */
14     {
15         printf("Je suis le fils, pid = %d\n", getpid());
16         sleep(2);
17         printf("Fin du fils, i = %d !\n", i);
18         exit(0);
19     }
20     else if (pid > 0) /* Processus le pere */
21     {
22         printf("Je suis le pere, pid = %d\n", getpid());
23         sleep(1);
24
25         /* Modifie la variable */
26         i = 2;
27         printf("le pere a modifie la variable a %d\n", i);
28
29         sleep(3);
30         printf("Fin du pere, i = %d !\n", i);
31         exit(0);
32     }
33     else /* Erreur */
34     {
35         perror("fork");
36         exit(1);
37     }
38
39     return 0;
40 }
```

qui donne à l'exécution:

```
$ ./fork &
[1] 367
$ 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 lorsqu'il s'agit de partager des variables entre un processus père et son fils. Dans l'exemple ci-dessus, la variable globale *i*, modifiée par le père 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'empêcher le partage de variables, la création d'un nouveau contexte est pénalisante au niveau performances, création et 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)

2. Les bibliothèques de threads

De nombreux systèmes d'exploitation permettent aujourd'hui la programmation par threads : Solaris 5.x de SUN, Windows95/98/NT, et 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 système. Dans le cas des systèmes 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 entièrement faite dans l'espace utilisateur.
- Au niveau noyau (kernel-level). Dans ce cas, les threads sont directement gérés par le noyau.

3. Créer des threads sous LINUX

Exemple de programme utilisant deux threads d'affichage:

thread1.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <pthread.h>
 5
 6 void *my_thread_process(void *arg)
 7 {
 8     int i;
 9
10     for (i = 0; i < 5; i++)
11     {
12         printf("Thread %s: %d\n", (char *)arg, i);
13         sleep(1);
14     }
15     pthread_exit(0);
16 }
17
18 int main(int ac, char **av)
19 {
20     pthread_t th1, th2;
21     void *ret;
22
23     if (pthread_create(&th1, NULL, my_thread_process, "1") < 0)
24     {
25         fprintf(stderr, "pthread_create error for thread 1\n");
26         exit(1);
27     }
28
29     if (pthread_create(&th2, NULL, my_thread_process, "2") < 0)
30     {
```

```
31          fprintf(stderr, "pthread_create error for thread 2\n");
32          exit(1);
33      }
34
35      (void)pthread_join(th1, &ret);
36      (void)pthread_join(th2, &ret);
37
38      return 0;
39  }
```

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

```
gcc -D_REENTRANT -o thread1 thread1.c -lpthread -Wall
```

Il donne à l'exécution:

```
$ ./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 d'utiliser des techniques permettant de protéger à un instant donné une variable partagée par plusieurs threads. Comme exemple, un 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 utilise le système des *MUTEX* (MUTual EXclusion) afin de protéger le tableau pendant le temps de son remplissage:

thread2.c

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <unistd.h>
 4  #include <pthread.h>
 5
 6  static pthread_mutex_t my_mutex;
 7  static int tab[5];
 8
 9  void *read_tab_process(void *arg)
10  {
11      int i;
12
13      pthread_mutex_lock(&my_mutex);
14      for (i = 0; i != 5; i++)
15          printf("read_process, tab[%d] vaut %d\n", i, tab[i]);
16      pthread_mutex_unlock(&my_mutex);
17      pthread_exit(0);
18  }
```

```
19
20 void *write_tab_process(void *arg)
21 {
22     int i;
23
24     pthread_mutex_lock(&my_mutex);
25     for (i = 0; i != 5; i++)
26     {
27         tab[i] = 2 * i;
28         printf("write_process, tab[%d] vaut %d\n", i, tab[i]);
29         sleep(1);               /* Relentit le thread d'ecriture... */
30     }
31     pthread_mutex_unlock(&my_mutex);
32     pthread_exit(0);
33 }
34
35 int main(int ac, char **av)
36 {
37     pthread_t th1, th2;
38     void *ret;
39
40     pthread_mutex_init(&my_mutex, NULL);
41
42     if (pthread_create(&th1, NULL, write_tab_process, NULL) < 0)
43     {
44         fprintf(stderr, "pthread_create error for thread 1\n");
45         exit(1);
46     }
47
48     if (pthread_create(&th2, NULL, read_tab_process, NULL) < 0)
49     {
50         fprintf(stderr, "pthread_create error for thread 2\n");
51         exit(1);
52     }
53
54     (void)pthread_join(th1, &ret);
55     (void)pthread_join(th2, &ret);
56
57     return 0;
58 }
```

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 à *pthread_mutex_unlock* pour verrouiller à son tour le MUTEX et lire le tableau correct. A l'exécution on obtient :

```
$ ./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
```

Sans MUTEX, on obtiendrait par contre:

```
$ ./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 :

thread3.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <pthread.h>
 5 #include <semaphore.h>
 6
 7 static sem_t my_sem;
 8 int the_end;
 9
10 void *thread1_process(void *arg)
11 {
12     while (!the_end)
13     {
14         printf("Je t'attend !\n");
15         sem_wait(&my_sem);
16     }
17
18     printf("OK, je sors !\n");
19     pthread_exit(0);
20 }
21
22 void *thread2_process(void *arg)
23 {
24     register int i;
25
26     for (i = 0; i < 5; i++)
27     {
28         printf("J'arrive %d !\n", i);
29         sem_post(&my_sem);
30         sleep(1);
31     }
32
33     the_end = 1;
34     sem_post(&my_sem);          /* Pour debloquer le dernier sem_wait */
35     pthread_exit(0);
36 }
37
38 int main(int ac, char **av)
39 {
40     pthread_t th1, th2;
41     void   *ret;
42
43     sem_init(&my_sem, 0, 0);
44
45     if (pthread_create(&th1, NULL, thread1_process, NULL) < 0)
46     {
47         fprintf(stderr, "pthread_create error for thread 1\n");
48         exit(1);
```

```
49      }
50
51      if (pthread_create(&th2, NULL, thread2_process, NULL) < 0)
52      {
53          fprintf(stderr, "pthread_create error for thread 2\n");
54          exit(1);
55      }
56
57      (void)pthread_join(th1, &ret);
58      (void)pthread_join(th2, &ret);
59
60      return 0;
61  }
```

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

```
$ ./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 ;* 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:

```
1   pthread_attr_t thread_attr;
2
3   if (pthread_attr_init (&thread_attr) != 0) {
4     fprintf (stderr, "pthread_attr_init error");
5     exit (1);
6   }
7
8   if (pthread_attr_setdetachstate (&thread_attr, PTHREAD_CREATE_DETACHED) != 0) {
9     fprintf (stderr, "pthread_attr_setdetachstate error");
10
11    exit (1);
12  }
```

puis de créer les threads avec des appels du type:

```
1   if (pthread_create (&th1, &thread_attr, thread1_process, NULL) < 0) {
2     fprintf (stderr, "pthread_create error for thread 1\n");
3     exit (1);
4   }
```

6. Destruction de thread: cancellation

Les exemples ci-dessus utilise 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 envoit une requète 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:

thread4.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <pthread.h>
 5
 6 void   *my_thread_process(void *arg)
 7 {
 8     int     i;
 9
10     pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
11
12     for (i = 0; i < 5; i++)
13     {
14         printf("Thread %s: %d\n", (char *)arg, i);
15         sleep(1);
16         pthread_testcancel();
17     }
18     return NULL;
19 }
20
21 int main(int ac, char **av)
22 {
23     pthread_t th1;
24     void   *ret;
25
26     if (pthread_create(&th1, NULL, my_thread_process, "1") < 0)
27     {
28         fprintf(stderr, "pthread_create error for thread 1\n");
29         exit(1);
30     }
31
32     sleep(2);
33     if (pthread_cancel(th1) != 0)
34     {
35         fprintf(stderr, "pthread_cancel error for thread 1\n");
36         exit(1);
37     }
38
39     (void)pthread_join(th1, &ret);
40
41     return 0;
42 }
```

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:

```
$ ./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 manière suivante:

```
 1 void *my_thread_process (void * arg)
 2 {
 3   int i;
 4
 5   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
 6   pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
 7
 8   for (i = 0 ; i < 5 ; i++) {
 9     printf ("Thread %s: %d\n", (char*)arg, i);
10     sleep (1);
11   }
12 }
```