INFO-F-201 – Systèmes d'exploitation

Threads

Joël Goossens (N8.107) Olivier Markowitch (N8.106)
Arnaud Leponce (N8.213) Yannick Molinghen (N8.210)
Alexis Reynouard (N8.215)

1 Programmation multi-threads

Le concept de thread est proche de celui de processus, car tous deux sont un moyen de paralléliser un programme. Cependant, comme nous allons le voir au cours de ce TP, les threads diffèrent des processus dans plusieurs aspects que nous allons découvrir dans ce TP.

Là où chaque processus possède sa propre mémoire virtuelle (heap et stack), les threads d'un même processus se partagent le heap. Par contre, vu que les threads ont des exécutions différentes, ils ne peuvent de toute évidence pas partager le même stack.

Nous allons dans ce TP utiliser les threads conformes à la norme POSIX, ou `pthread`, que l'on peut créer à l'aide de la fonction `pthread_create`. La fonction `pthread_exit` permet, elle de mettre fin à un thread.

```
#include <pthread.h>
int pthread_create(pthread_t* thread, pthread_attr_t* attr,
                   void* (*start_routine)(void*), void* arg);
void pthread_exit(void *retval);
```

Note : *Pour compiler avec la librairie `pthread`, il faut ajouter le flag `-lpthread` à la fin de la commande de compilation.*

Le premier paramètre, `thread`, contiendra l'identifiant unique du thread créé. Il sert principalement comme paramètre pour les autres fonctions liées aux threads que nous verrons ci-après.

Le second paramètre, `attr`, est une structure de données utilisée lors de la création du thread pour déterminer les attributs de ce dernier. Dans le cadre de ce cours, ce paramètre prendra la valeur `NULL`.

Le troisième paramètre, `start_routine`, est un pointeur vers la fonction à exécuter pour le thread. En effet, un thread a pour vocation d'effectuer une unique fonction conforme à cette signature, come illustré ci-dessous.

Le dernier paramètre, `arg`, est un pointeur vers les paramètre à donner à la fonction `start_routine`.

Note : *Il peut parfois être difficile de déchiffrer le type certains paramètres en C. Ici,* `start_routine` *est de type* `void* (*start_routine)(void*)`*, c'est-à-dire un pointeur vers une fonction qui prend un paramètre de type* `void *` *et qui renvoie un pointeur vers* `void`*.*

```
// Une fonction compatible avec pthread_create
void* nom_de_la_fonction(void* param){ return NULL;  }
```

Une fois l'appel à `pthread_create` effectué, l'exécution reprend à l'instruction suivante dans le thread principal et dans la fonction dans le nouveau thread.

Note : *En C, on utilise le type* `void *` *pour indiquer un pointeur vers n'importe quel type de données. Il appartient au programmeur de caster le pointeur vers le type approprié.*

Tout comme il est utile d'attendre la fin d'un processus enfant, il est souvent utile, voire nécessaire, de synchroniser les threads entre eux et d'attendre la fin d'un thread en particulier. Cela se fait à l'aide de la fonction `pthread_join`.

```
int pthread_join(pthread_t thread, void **retval)
```

La fonction prend en paramètre l'identifiant du thread à attendre, et un pointeur sur un espace mémoire qui contiendra la valeur de retour du thread attendu.

Le thread se termine lorsque l'un des cas suivants se produit :

1. le thread fait un appel à l'appel système suivant :

   ```
   void pthread_exit(void *retval)
   ```

   Cette fonction permet de spécifier une valeur de retour. Comme énoncé précédemment, la valeur de retour peut être récupérée par le thread principal avec un appel à `pthread_join`,
2. le thread effectue un retour classique de la fonction pour laquelle il a été créé. Ce type de terminaison est équivalente à un appel à `pthread_exit`,
3. le thread principal effectue un appel à la fonction suivante :

   ```
   int pthread_cancel(pthread_t thread_id)
   ```

   Le paramètre `thread` est l'identifiant unique du thread à interrompre.
4. Le thread effectue un appel à `exit` provoquant la terminaison du thread appelant, du thread principal et de tous les autres threads lancés par le thread principal. De la même manière [1], exécuter un return depuis la fonction `main` termine tous les threads.

Exercice 1. *Écrivez un programme C créant un thread. Le thread principal écrira cent fois 1 et le second thread écrira cent fois 2.*

Exercice 2. *Écrivez un programme qui initialise une variable* `int count=0;` *et crée deux threads. Chaque thread incrémente la variable globale 5.000.000 fois puis se termine, tandis que le thread principal attend la fin de l'exécution des deux threads puis affiche la valeur de la variable globale.*

Qu'observez-vous ? Comment l'expliquez-vous ?

1. Le C standard indique que tout retour via return de la fonction main appelée initialement, si elle est définie comme retournant un int (comme nous la définissons usuellement), équivaut implicitement à appeler `exit`

Exercice 3. *Remplacez le type de la variable* `count` *par* `atomic_int` *et relancez votre programme. Veillez à utiliser l'opération d'incrémentation* `count++`*. Qu'observez-vous ?*

Exercice 4. *Mesurez et comparez le temps d'exécution des deux exercices précédents à l'aide de la commande* `time`*.*

Exercice 5. *Dans cet exercice, vous allez écrire un programme qui simule une course de chevaux. Chaque cheval doit courir et passer par les balises de numéro 1 au numéro 10. Votre programme principal va créer un thread par cheval et passer en paramètre une structure de type* `cheval_t`*.*

```
typedef struct {
    char* nom;
    int numero;
} cheval_t
```

Chaque thread doit

1. *Afficher le nom et le numéro du cheval*
2. *Pour chacune des 10 balises*
   (a) *S'endormir pour une durée aléatoire entre 1 et 4 secondes*
   (b) *Afficher le nom du cheval et de la balise atteinte*
3. *En arrivant à la dernière balise, afficher "<nom du cheval> a fini la course !"*

Ensuite, le programme se termine en écrivant "La course est terminée !".

Exercice 6. *Écrivez un programme qui calcule la moyenne d'un tableau de nombres aléatoirement générés de manière distribuée à l'aide de threads.*

2 Les programmes de gestion des threads

Normalement, les threads d'un même processus peuvent communiquer entre eux comme les processus d'une même machine peuvent communiquer entre eux. Ceci signifie que l'interaction avec un processus se fait normalement en considérant le processus comme un tout : on ne va pas communiquer directement avec ses threads. Mais même si c'est la façon commune de faire, on peut avoir des raisons de vouloir interagir directement avec les threads, ne serait-ce que pour les monitorer.

Pour ce faire, il faut utiliser l'ID du thread. Celui-ci peut être obtenu à l'aide de la fonction `gettid` :

```
#include <unistd.h>
pid_t gettid(void);
```

Une autre solution est d'utiliser les commandes du terminal :

ps -fL L'option `-L` sert à afficher les threads, l'option `-f` sert à afficher plus de colonnes. Les deux combinées ajoutent les colonnes NLWP avec le nombre de threads et LWP avec les *thread ID*. LWP signifie *lightweight process*, processus léger, un synonyme de thread. Voyez le TP précédent pous plus d'options utiles de `ps`.

htop a aussi une colonne NLWP qui peut être activée dans les options.

pgrep <pattern> liste normalement les PID des processus correspondant à la recherche `<pattern>`. Avec `-w`, il liste tous les thread ID aussi. Essayez avec et sans `-w` pour un pattern comme `gnome`, `mate`, `cinnamon`, `firefox`... suivant ce que vous utilisez en ce moment. Il y a bien sûr de nombreuses autres options possibles telle que `-P <PPID>` qui permet de lister tous les processus dont le père est `PPID`.