Sujet de TP 2 **Système** et réseau - Programmation système Licence 3

Programmation multithreadée

V. Felea & A. Hugeat

Les exercices sont orientés sur la manipulation de base des threads C norme POSIX (sans synchronisation). Exception faite pour le premier exercice, qui traite des pointeurs de fonction. La notion de pointeur de fonction est nécessaire pour la création des threads en C. Une présentation succincte de cette technique est donnée ci-après.

Pointeur de fonction - description de la technique

Un pointeur est une variable contenant une adresse mémoire. Les pointeurs peuvent également contenir l'adresse d'une fonction, c'est ce qui est appelé un *pointeur de fonction*. Cette dernière peut ainsi être passée en paramètre à une autre fonction et être appelée.

1. Déclaration d'un pointeur sur fonction

Syntaxe : `type_t (*pt_fct)(... paramètres ...);`
Pointeur sur une fonction

- renvoyant un résultat de type `type_t`
- de nom `pt_fct`
- ayant les paramètres de types donnés entre les parenthèses (`void` si absence de paramètres)

Exemples :

```
int (*fct)();
void (*fct_param)(int, char);
```

2. Affectation d'un pointeur sur fonction

```
pt_fct = nom_fonction; /* notation actuelle */
pt_fct = &nom_fonction; /* notation "cohérente" avec la notion d'adresse */
```

3. Appel (syntaxe) : `(*pt_fct)(... paramètres ...)`

Exemples :

```
(*fct)();
(*fct_param)(10,'a');
```

Exemple complet :

```
#include <stdio.h>
int somme(int i, int j) { return i+j;}

int main(int argc, char* argv[]) {
  int x, y;
  /* déclare un pointeur de fonction */
  int (*pSomme)(int, int);
  pSomme = somme; /* initialise pSomme avec l'adresse de la fonction Somme */
  /* pSomme = &somme; *//* autre notation moins utilisée */

  printf("Entrer deux entiers : "); scanf("%d%d",&x, &y);
  printf("Leur somme : %d\n", (*pSomme)(x,y));
  return 0;
}
```

4. Fonction comme paramètre d'une autre fonction

Le pointeur de fonction permet en particulier de généraliser un traitement (comme le tri croissant ou décroissant), en donnant le traitement à réailser comme paramètre d'une fonction.

Passage d'une fonction comme paramètre d'une autre fonction
Pour une fonction de prototype : `type fonction(type_1,...,type_n);`
le type de son pointeur de fonction est : `type (*)(type_1,...,type_n)`

Exercices

1. **Fonctions en paramètre**

   - Écrire une fonction `mult` qui multiplie par `fact` un entier `x` et renvoie la nouvelle valeur de `x`.
   - Écrire une fonction `map` qui applique une fonction binaire passée en paramètre à chaque élément d'un tableau unidimensionnel, d'entiers, de taille donnée `lg`. Le premier argument de la fonction à appliquer est un élément du tableau ; le deuxième argument correspond au deuxième opérande de l'opération binaire à appliquer. Le résultat de l'opération est la nouvelle valeur de l'élément du tableau.
   - Écrire le programme principal qui utilise `map` pour appliquer la fonction **mult** à un tableau unidimensionnel d'entiers.

2. **Affichage concurrent**

   Q1 (fonction sans paramètre) Écrire un programme qui crée deux threads. Chaque thread affiche son identifiant toutes les secondes ainsi que le nombre de fois que le thread a écrit. Après 10 affichages chacun, les threads devraient terminer. Le programme principal doit attendre la terminaison des deux threads avant de continuer.

   Indication La fonction `pthread_self` permet d'obtenir l'identificateur du thread courant.

   Rappel prototype `pthread_create`

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

Il convient d'utiliser ce type de déclaration de la fonction appelée :
`void* print_fct(void* ptr);`
L'appel de cette fonction dans un fil d'exécution indépendant est réalisé par l'intermédiaire de la fonction `pthread_create` (cas d'une fonction sans paramètres) :

```
ret = pthread_create(&thread1, NULL, print_fct, NULL);
```

Q2 (fonction avec paramètre) Modifier la solution précédente pour que le nombre d'affichages maximum soit donné par le programme principal.

3. **Générateur de matrice aléatoire**
Écrire un programme qui permet d'initialiser de manière aléatoire une matrice d'entiers, de manière concurrente, ligne par ligne. Pour implémenter ce mécanisme, un thread initialise tous les éléments d'une ligne. Chaque valeur aléatoire entière doit être comprise entre deux limites données par le programme principal, qui obtient ces valeurs depuis les arguments en ligne de commande.

4. **Somme des éléments d'une matrice**
Utiliser l'exercice précédent pour initialiser une matrice de manière aléatoire. Créer un thread pour chaque ligne de la matrice qui calcule la somme des éléments de la ligne correspondante et le programme principal calcule la somme finale, additionnant toutes les sommes partielles, des lignes, calculées par les threads.

5. **Commandes système**

Q1 Pour le programme de l'exercice 2, question 2, visualiser à l'aide des commandes système les identifiants des threads créés. Correspondent-ils aux identifiants affichés lors de l'exécution du programme ? Justifier.
Indication Le nombre d'affichages donné par le programme principal devrait être suffisamment grand pour que les commandes systèmes soient exécutées pendant la durée de vie du thread.

Q2 Rajouter dans ce programme, en précision de la valeur retournée par la fonction `pthread_self`, l'affichage du résultat retourné par l'instruction

```
syscall(SYS_gettid)
/* à inclure la bibliothèque <sys/syscall.h> */
```

Que pouvons-nous conclure ?
Attention Cette solution n'assure pas la portabilité.

Q3 Grâce aux commandes système, déterminer quels sont les processus multithreadés qui s'exécutent à un instant donné sur la machine courante (leur identifiant de processus, ainsi que la commande associée doivent être affichés) et leur nombre de threads. Donner le script shell ayant permis d'obtenir ces informations.