Système d'Exploitation Unix
TP
Les PThreads

Tous les programmes devront être développés avec passage de leurs éventuels paramètres à la fonction `main (int argc, char *argv [])`. Les valeurs de retour des appels aux primitives devront être testées et les messages d'erreurs affiches avec `perror`. Les messages d'erreurs à destination de l'utilisateur se feront sur le fichier standard des erreurs `stderr`.
Les programmes seront compilés avec l'option `–pthread` :
`gcc nom.c  -pthread –o nom`

Question 1 Création de threads

Ecrivez un programme qui crée n threads. Chaque thread exécute une boucle suffisamment longue pour permettre l'observation et dans laquelle il écrit à chaque itération:

`Bonjour, je suis le thread` *`tid`*

Le thread 1 affiche son message au debut de la ligne, le thread 2 à une tabulation du début de la ligne, le thread 3 à deux tabulations du début de la ligne, etc

Fonctions utiles:
- *pthread_create, pthread_exit (): cf. cours.*

Question 2 Synchronisation des threads

a) Ecrire un programme qui crée :
- n threads (n<=26) qui écrivent indéfiniment un caractère dans toutes les entrées d'un même tableau. Le thread écrivain 1 écrit le caractère A, le thread écrivain 2 écrit le caractère B, etc.
  Chaque thread écrivain est créé "détaché". Il reçoit en paramètre son numéro d'ordre de création
- un thread lecteur qui affiche le contenu complet du tableau et qui totalise le nombre de fois où un caractère apparaît dans le tableau. Ce thread lecteur du tableau est créé "joignable" et sera attendu par le thread `main`.
  Il reçoit en paramètre le nombre de fois où il doit afficher le contenu du tableau.

  Lorsque le thread lecteur est terminé, le thread `main` affiche le nombre d'apparitions de chaque caractère constatées par le thread lecteur.

Appel: `./nom 3 5`
Résultat pour un tableau de 75 caractères:

```
DEBUT MAIN
```

```
CBBCAAAACCAACBBBBAAACBCCCCCBBBBBCCAABBBBBACCCCAAAAABBBCCCCCBBAAAACCCCCCCBCC
CCBAACCAAAAACCBBBBBBBACBBBBCAAACCBACCCCCACCCCAAAABBBBAAAAACCAAAAAAAABABBBCC
CAAAAAAAACBBBBAAAAAACCCCAAAACCCABBBBBAAAABBBBACCCCCABCCCAAAABBAAAAAAAAACBBB
CBCAAAAABBBACCCCCBBBCCCAAAAABBCCCCCABBBBBCCAABBBBAACCCCAACCCCCCAAAACAAAAAAA
CCCAAAACCCCCCAAACCCCCCCCCCCCCCCCCCCAACCCCAACCCCAAACCCCCCCCAAAAACAAAAAAAAACC
A: 147
B: 81
C: 147
FIN MAIN
```

b) Procédez à plusieurs exécutions de : `./nom 3 5` . Expliquez ce que vous constatez.

c) Modifiez le programme précédent afin que

1. le thread lecteur
   - affiche chaque caractère lu du tableau et le remplace par un espace
   - ne s'exécute que lorsque le tableau ne contient que des caractères différents de l'espace
2. chaque thread écrivain :
   - soit le seul à écrire son caractère dans le tableau **au moment où il écrit ce caractère**.
   - n'écrive son caractère que dans les cases du tableau contenant un espace
   - ne s'exécute que lorsque le tableau a été rempli d'espaces par le thread lecteur.
   - Modifiez le programme précédent afin que

d) Expliquez pourquoi ce programme ne permet pas tout le parallélisme possible entre les threads écrivains.

e) Proposez et codez une solution permettant tout le parallélisme possible entre les threads écrivains.

Fonctions utiles:

pthread_create(), pthread_exit(), pthread_attr_init(), pthread_attr_setdetachstate(), pthread_join(), pthread_mutex_lock(), pthread_mutex_unlock(),pthread_cond_wait(), pthread_cond_signal(),pthread_cond_broadcast()

Question 3 Tri rapide avec les threads (question d'examen de a à f)

La méthode du tri rapide d'un tableau consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, gauche et droite, on définit un nouveau pivot et on recommence l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments du tableau soit trié.

La fonction `quicksort` ci-dessous implémente une version séquentielle du trie rapide d'un tableau `tab` s'étendant de l'indice `premier` à l'indice `dernier` inclus. La fonction `partition`

réalise le partitionnement du tableau `tab` entre de l'indice `premier` et l'indice `dernier` inclus. Elle retourne l'indice de du pivot dans la zone du tableau partitionnée. Vous devez modifiez le code ci-dessous pour en faire en une version parallèle **`pquicksort`** fondée sur les threads. Dans cette version parallèle, **chaque appel à la fonction `quicksort` sera exécuté par un thread**.

a) Ecrivez le code de la ligne 2 définissant la structure contenant les paramètres passés à chaque thread exécutant la fonction `quicksort`.
b) Ecrivez le code de la ligne 4 redéfinissant le prototype de la fonction `quicksort` afin qu'elle soit exécutable par un thread.
c) Ecrivez le code de la ligne 6 déclarant les variables locales nécessaires à la fonction `quicksort` pour son exécution par un thread.
d) Ecrivez le code des lignes 9, 10 et 20 remplaçant chaque appel à la fonction `quicksort` par la création d'un thread. Précisez à chaque fois le n° de la ligne concernée (9, 10 ou 20).
e) Ecrivez le code de la ligne 11 permettant de joindre (attendre la fin) les des deux threads créés en remplacement des lignes 9 et 10. Vous considérerez que les threads sont joignables par défaut.
f) Le thread créé en replacement de la ligne 20 doit-il ou pas être joint afin que le tableau trié soit correctement affiché ? Justifiez votre réponse.

Source du programme suivant : http://www.mi.parisdescartes.fr/~soto/PUnix/quicksort.c

```
1.   #include <stdio.h>
2.
//= Prototype de la fonction de partitionnement =
3.  int partition (int tab[], int premier,
                              int dernier);
// ============================================

// ============================================
4.   void quickSort (int tab[], int premier,
                                 int dernier){
// ============================================
5.   int j;
6.
7.   if (premier < dernier){
 // Partionnement du tableau
8.  j = partition (tab, premier, dernier);

// Tri de la partie à gauche du pivot
9.    quickSort (tab, premier, j-1);

// Tri de la partie à droite du pivot
10.   quickSort (tab, j+1, dernier);
11.
    }// if
 }// quickSort

// ============================================
12. void main () {
// ============================================
13. int i, tab[] = { 79, 17, 14, 65, 89, 4, 95};
14. int indice_max;

15. indice_max=(sizeof (tab)/sizeof (int)) - 1;

16. printf ("\nTableau non trié:\t");
17. for (i = 0; i <= indice_max; ++i)
18.      printf (" %d ", tab[i]);
19. printf ("\n");

20. quickSort (tab, 0, indice_max);

21. printf ("Tableau trié:\t\t");
22. for (i = 0; i <= indice_max; ++i)
23.      printf (" %d ", tab[i]);
24. printf ("\n");

 }// main

 /* ------------------------------------- */
    int partition (int tab[], int premier,
                              int dernier)
 /* ------------------------------------- */
 {
    int pivot, i, j, t;
    pivot = tab[premier];
    i = premier; j = dernier+1;

    while (1){
     do ++i; while (tab[i] <= pivot && i <=
                                    dernier);
     do --j; while (tab[j] > pivot);
     if (i >= j) break;
     t = tab[i]; tab[i] = tab[j]; tab[j] = t;
    }// while

    t = tab[premier]; tab[premier] = tab[j];
 tab[j] = t;

    return j;

 } // partition
```

g) Modifiez le programme **`quicksort`** ci-dessus afin qu'il trie un tableau d'entiers de n'importe quelle taille. Les valeurs de ce tableau seront générées avec la fonction `rand`.

   Ce programme se lance de la façon suivante : **`quicksort t max`**

   Où `t` est la taille du tableau à trier et [0-max] l'intervalle à l'intérieur duquel sont générées les valeurs aléatoires du tableau.

   En plus de trier le tableau, ce programme renvoie le nombre d'appel à la fonction **`quicksort`** qui ont été nécessaires à son tri

h) Modifiez de la même manière le programme résultant des questions a) à e) afin qu'il trie un tableau d'entiers de n'importe quelle taille. Les valeurs de ce tableau seront générées avec la fonction `rand`.

   Ce programme se lance de la façon suivante : **`pquicksort t max`**

   Où `t` est la taille du tableau à trier et [0-max] l'intervalle à l'intérieur duquel sont générées les valeurs aléatoires du tableau.

   En plus de trier le tableau, ce programme renvoie le nombre de thread qui ont été nécessaires à son tri

i) Lancez plusieurs fois votre programme avec des valeurs de `t` croissantes jusqu'à trouver ses limites d'exécution. Comparez ces limites avec la version séquentielle **`quicksort`**

j) Expliquez ce qui limite votre programme **`pquicksort`** et proposez des solutions pour repousser, un peu, ses limites