TP 3– Pointeurs de fonction, threads et mutexs

Sovannara Hak et Vincent Padois
hak@isir.upmc.fr

Concepts abordés

- Pointeurs de fonction
- Création de processus légers avec la bibliotèque pthread
- Utilisation de mutexs
- Manipulation d'images

Fonctions utiles

- atoi , strcmp
- malloc, free
- pthread_create, pthread_exit, pthread_join, pthread_kill
- rand, srand, rand_r
- ceil

N'hésitez pas à utiliser la commande man 2 ou man 3 pour obtenir la documentation de chaque fonction.

Contraintes et remise des TPs

- Le code de chaque exercice doit être dans un répertoire séparé (exercice1, exercice2, etc.).
- Chaque répertoire d'exercice doit contenir un fichier Makefile permettant de compiler les réponses à chaque question.
- Lorsque c'est nécessaire, ajoutez dans le repertoire de l'exercice des fichiers texte ou image pour répondre aux questions (avec un nom explicite).
- N'oubliez pas de rendre une archive « propre » (pas de .o, pas binaires, pas de fichier temporaires *.swp ou *~). La cible « clean » de vos makefiles est faite pour cela !
- Indentez vos fichiers (commande `indent` ou éditeur de texte civilisé).
- N'oubliez pas les "gardes" (#ifndef ...) dans vos fichier `.h`.
- La correction tiendra compte de la briéveté des fonctions que vous écrivez (évitez les fonctions de plus de 25 lignes) ; n'hésitez pas à découper une fonction en plusieurs sous-fonctions plus courtes.
- Votre enseigant vérifiera avec Valgrind l'absence d'erreurs (fuites mémoires, lectures invalides, ...).
- La convention de nommage des fonctions est la suivante : une fonction doit avoir un nom explicite et si son nom se compose de plusieurs mots, ces derniers sont écrits en minuscule et séparés d'un tiret bas _. Ainsi une fonction qui affiche un tableau sera appelée `affiche_tableau`.

- Le code source doit être envoyé forme d'une seule archive tar.gz [1]
- Votre archive doit être propre : pas d'exécutables, pas de .o, pas de fichiers temporaires. **au maximum 30 minutes après la fin du TP**
- Envoyez votre archive par e-mail à `hak@isir.upmc.fr`
- Les fichiers utiles et les transparents de cours sont sur `http://pages.isir.upmc.fr/~hak`

Exercice 1 – Pointeurs de fonction

Soit le programme suivant (disponible sur le site du cours) :
`#### question0.c ###`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void usage(){
5    printf("Usage : ./question0 nombre1 nombre2 operateur\n");
6    printf("\t operateur : +, -, x, /\n");
7  }
8
9  double plus(double a, double b){ return a+b;}
10 double minus(double a, double b) { return a-b;}
11 double multiply(double a, double b) { return a*b;}
12 double divide(double a, double b) {
13   if(b != 0.0){
14     return a / b;
15   } else {
16     return 0;
17   }
18 }
19
20 void switch_op(double a, double b, char op_code){
21   double result;
22   int ok = 1;
23
24   switch(op_code){
25   case '+' : result = plus(a, b); break;
26   case '-' : result = minus(a, b); break;
27   case 'x' : result = multiply(a, b); break;
28   case '/' : result = divide(a, b); break;
29   default : ok = 0;
30   }
31
32   if(ok == 1)
33     printf("%lf\t%c\t%lf\t=\t%lf\n",a,op_code,b,result);
34 }
35
36 int main(int argc, char* argv[]){
```

1. Pour faire une archive tar.gz : `tar zcvf mon_archive.tar.gz mon_repertoire`

```
37
38    double nb1, nb2;
39    char op;
40
41    if(argc != 4){
42      usage();
43      return -1;
44    }
45
46    nb1 = atof(argv[1]);
47    nb2 = atof(argv[2]);
48    op = argv[3][0];
49
50    switch_op(nb1,nb2,op);
51
52    return 0;
53  }
```

1. Compilez et exécutez ce programme. Que fait-il ? (Ajoutez dans un commentaire C votre réponse directement dans le fichiers)
   Pour la suite de cet exercice, il est inutile de séparer les questions dans le Makefile comme demandé dans les consignes. Modifiez tout le temps de le même fichier.
2. En utilisant typedef, définissez le type décrivant un pointeur sur une fonction prenant deux arguments de type **double** et retournant un double.
3. Ecrivez un fonction `choix_op` qui prend comme paramètre d'entrée l'opérateur et retourne un pointeur sur la fonction correspondante. Vous utiliserez le type défini à la question précédente.
4. Ecrivez un fonction `switch_op_pf` remplaçant `switch_op` qui prend comme paramètres d'entrée les opérandes et un pointeur sur la fonction correspondant à l'opération à effectuer.
5. Modifiez votre programme principal en conséquence et testez le.

Exercice 2 – Threads : exemples simples

1. Ecrivez un programme lançant deux *threads* appelant la même fonction de façon à ce que le premier thread écrive "hello" 8 fois et le second "world" 4 fois.
2. On souhaite maintenant réaliser un programme composé d'un *thread* principal dans lequel l'utilisateur saisit des chaînes de cactères et un *thread* observateur qui affiche « FOUND » si l'utilisateur tape la chaîne « groovy » et continue de l'afficher toutes les secondes si rien d'autre n'est tapé. (On pourra éventuellement quitter le progamme en tapant « quit »). *Attention, l'utilisation de variables globales est illégal dans cet exercice.*
   Exemple :

```
1  ?ue
2  ?ueo
3  ?groovy
```

```
4  ?
5  FOUND
6
7  FOUND
8
9  FOUND
10 x
11 ?2
12 ?
```

Exercice 3 – Traitement d'images multi-threadé

Cet exercice a pour but le développement d'un programme multi-threadé (ie. utilisant des threads) afin de mettre en oeuvre une fonction simple de traitement d'images. Afin de ne pas alourdir le code et pour des raisons évidentes de temps, nous nous limitons au traitement d'images en niveaux de gris. Le format retenu est le format PGM (Portable Grey Map). Ce n'est pas le plus connu mais il a l'avantage d'être facile à manipuler. Vous pouvez très facilement obtenir des fichiers *pgm* à partir d'autres formats en utilisant l'utilitaire en ligne de commande `convert` ou des outils plus sophistiqués de traitement d'images :
`convert fichier.jpg fichier.pgm`
Pour visualiser une image au format *pgm*, vous pouvez utiliser le programme `eog` (`eog fichier.pgm &` en ligne de commande).

Si on ouvre une image au format *pgm* avec son éditeur de texte préféré, on constate que ce format de fichier contient un en-tête de type ASCII composé de trois lignes contenant respectivement :

1. un code de format **P5** correspondant au format *pgm* ;
2. deux entiers séparés par un espace et correspondant à la largeur et la hauteur de l'image ;
3. un entier représentant le nombre de niveaux de gris utilisé pour l'image (si ce nombre est, par exemple, 255, cela signifie que les niveaux de gris sont codés dans l'intervalle $[0, 255]$).

A la suite de cet en-tête, on trouve le buffer binaire correspondant aux pixels de l'image rangés par lignes.

Code fourni

Les fichiers suivants vous sont fournis :

- `image.h` : il contient la définition du type structuré `image_t` ainsi que les prototypes des fonctions utilisés pour manipuler les images : création, allocation, copie, destruction, chargement, sauvegarde ;
- `image.c` : il contient l'implémentation des fonctions déclarées dans le `.h` ;
- `transf_image.h` : **A compléter**, il contient le prototype des fonctions de transformation d'images ;
- `transf_image.c` : **A compléter**, il contient l'implémentation des fonctions déclarées dans le `.h` ;

– `Makefile` : un fichier permettant de compiler les différentes questions séparément en utilisant make ;
– `images` : ce répertoire contient un ensemble d'images au format *pgm* qui pourront être utilisées pour vos tests.

Les images étant rangées en mémoire sous la forme d'un vecteur colonne, la macro VAL(img,i,j) définie dans le fichier image.h permet d'accéder simplement au pixel (i,j) de l'image pointée par img .

1. **Bruitage d'une image**
   Cette question a pour but d'écrire une fonction permettant de bruiter une image.
   Ajoutez dans `transf_image.c` la fonction bruiter_image() dont le prototype vous est donné et qui créé une image bruitée à partir d'une image source. Le principe du bruitage est simple : pour chaque pixel de l'image, une valeur aléatoire est tirée dans l'intervalle $[0, 100]$. Si cette valeur est inférieure au pourcentage fournit comme argument d'entrée de la fonction, le pixel prend une valeur aléatoire dans l'intervalle de niveau de gris de l'image.
2. Ajoutez l'instruction nécessaire au fichier `question2.c` afin de pouvoir tester la fonction de bruitage.

Bruitage : version multi-threadée

Cette série de questions a pour but d'obtenir une version multi-threadé de la fonction bruit_image() .

3. Dans un premier temps, créez une fonction bruit_image_nm() prenant comme paramètres d'entrée un pointeur sur l'image destination, un entier non signé représentant le pourcentage de bruit, quatre entiers non signés représentant respectivement un numéro de colonne de départ, un numéro de ligne de départ, le nombre de colonnes de pixels à bruiter à partir de la colonne de départ et le nombre de lignes de pixels à bruiter à partir de la ligne de départ. Cette fonction ne bruite que les pixels de la zone dont les paramètres sont passés à la fonction. Les paramètres de la fonction doivent être décrits sous une forme générique en vue de l'utilisation de la fonction dans un thread. La structure de donnée associée pourra être définie dans `transf_image.h`.
4. Créez une fonction principale dans le fichier `question4.c` afin de pouvoir tester la nouvelle fonction de bruitage en supposant qu'on commence le bruitage au pixel de 50ème colonne et de la 200ème ligne.
5. Créez une fonction principale dans le fichier `question5.c` permettant de lancer le bruitage d'une image en découpant l'image en $n \times m$ zones chaque zone étant associée à un thread.
6. Le bruitage d'image en version threadée nécessite-t-il un mécanisme d'exclusion mutuelle ? Pourquoi ? Qu'en est-il pour d'autres opérations de traitement d'images utilisant l'opérateur de convolution ? (réponse à inscrire dans un commentaire dans le fichier question5.c).