Programmation des systèmes

Devoir surveillé final

13 décembre 2016

Durée : 2h30
Documents autorisés

Vous expliquerez votre code de sorte que votre intention (et pas seulement votre implémentation) soit claire. La gestion des erreurs pourra rester très simple, par exemple en interrompant le programme si nécessaire.

Lorsque vos réponses sont de simples modifications de fonctions précédentes, vous vous contenterez d'indiquer les modifications (en particulier en précisant clairement où ces modifications sont placées).

Vos fonctions utiliseront exclusivement les entrées/sorties de bas niveau (c'est-à-dire `open`, `close`, `read`, `write`, etc.).

Afin de paralléliser la correction, vous rendrez 2 copies, 1 copie par section.

1 Première copie

Vous composerez les deux exercices suivants sur la première copie.

1.1 Détection de répertoires vides

L'objectif de cet exercice est de découvrir tous les répertoires vides dans une arborescence[1].

Considérons par exemple l'arborescence suivante :

```
.
├── rép0
│   └── fic0
├── rép1
│   └── rép2
│       └── rép3
└── rép4
    ├── rép5
    └── rép6
```

où les `répX` sont des répertoires et `fic0` est un fichier ordinaire. Les répertoires `rép3`, `rép5` et `rép6` sont *vides*.

Nous voulons définir une fonction `void liste_repertoires_vides(char *chemin)` qui :

- si `chemin` est un répertoire contenant des entrées (fichiers ordinaires, sous-répertoires, etc.), teste récursivement si ces entrées sont, ou contiennent, des répertoires vides,
- si `chemin` est un répertoire ne contenant aucune entrée, affiche ce chemin sur la sortie standard,
- si `chemin` n'est pas un répertoire, ne fait rien.

`liste_repertoires_vides(".")` devra donc afficher pour l'arborescence donnée ci-dessus :

```
./rép1/rép2/rép3
./rép4/rép5
./rép4/rép6
```

1. Un répertoire vide n'est en général pas très intéressant. Par exemple, l'outil `git` ignore de tels répertoires.

Dans un premier temps, nous décidons de ne pas suivre les liens symboliques.

Q 1. Définissez la fonction `liste_repertoires_vides`.

Q 2. Indiquez comment modifier votre fonction pour suivre les liens symboliques.

Nous décidons d'élargir la définition d'un répertoire vide : si un répertoire ne contient que des sous-répertoires vides, il sera considéré comme vide lui aussi. Par exemple, dans l'arborescence donnée ci-dessus, `rép4` sera aussi considéré *vide*, puisqu'il ne contient que des répertoires vides ; `rép2` aussi, et par conséquent `rép1` aussi. Seul `rép0` ne sera pas affiché.

En plus de l'affichage des répertoires vides trouvés, la nouvelle fonction que nous voulons définir, `int liste_repertoires_vides_bis(char *chemin)`, retournera 1 si `chemin` est un répertoire vide et 0 dans les autres cas.

Q 3. Définissez la fonction `liste_repertoires_vides_bis`.

1.2 Redirection(s) depuis un fichier

Redirection simple

La commande `from file cmd` exécute la commande `cmd` en ayant préalablement pris le soin de rediriger son entrée standard depuis le fichier `file`.

Q 4. Réfléchissons à deux cas particuliers. Quel résultat voudrait-on avoir pour la commande `from` si le fichier `file` n'existe pas ? Et si le fichier `file` est vide ? Justifiez.

Q 5. Donnez une implémentation de la commande `from`. *Pour simplifier, nous supposerons que `cmd` ne prend pas d'arguments.*

Redirections multiples

La commande `froms file cmd1 cmd2 cmd3 ...` exécute les commandes `cmd1`, `cmd2`, `cmd3`, etc. en prenant soin de préalablement rediriger leurs entrées standards depuis le fichier `file`. Chacune des commandes sera exécutée en parallèle par un processus fils indépendant.

Q 6. Nous prendrons soin d'ouvrir un unique descripteur de fichier avant d'exécuter les différentes commandes. Est-il important de déplacer la tête de lecture du fichier au début du fichier à chaque nouveau fils ? Pourquoi ?

Q 7. Donnez une implémentation de la commande `froms` en supposant que les commandes `cmd1`, `cmd2`, etc. ne prennent pas d'arguments.

Q 8. Peut-on donner l'ordre de fin d'exécution des différentes commandes ? Justifiez.

2 Deuxième copie

Vous composerez les deux exercices suivants sur la deuxième copie.

2.1 Lecture avec tampon

L'objectif de cet exercice est d'écrire quelques fonctions nécessaires pour un outil de type `diff` permettant de comparer deux fichiers texte, ligne par ligne.

Notre programme `mdiff` prend en arguments sur la ligne de commande les noms de deux fichiers à comparer. Il lit les deux fichiers ligne par ligne jusqu'à trouver deux lignes différentes. Soient `l1` et `l2` les premières deux lignes différentes, `l1` dans le premier fichier et `l2` dans le deuxième fichier :

— `mdiff` compare la ligne `l1` avec toutes les lignes du deuxième fichier après `l2`,
— s'il trouve une ligne `l22` égale à `l1`, il imprime toutes les lignes du deuxième fichier entre `l2` (incluse) et `l22` (exclue) puis reprend la comparaison ligne à ligne,
— s'il ne trouve pas de ligne égale à `l1`, il imprime `l1`, il lit la ligne suivante dans le premier fichier, et il recommence.

Par exemple, pour les deux fichiers suivants :

a.txt	b.txt
abc	abc
def	def
ghi	xxx
jkl	yyy
mno	jkl
pqr	mno
stu	

la commande `mdiff a.txt b.txt` imprime :

```
[a.txt]
ghi
[b.txt]
xxx
yyy
[a.txt]
pqr
stu
```

Pour réaliser ce programme de manière efficace, on souhaite réaliser une bibliothèque de fonctions pour :

— lire un fichier ligne par ligne, en utilisant un tampon (comme dans la bibliothèque entrée/sortie standard du C) ;
— sauvegarder la position (*offset*) à laquelle se trouve une ligne dans le fichier ;
— revenir à la position sauvegardée précédemment ;
— imprimer sur la sortie standard toutes les lignes comprises entre la position sauvegardée et la position courante.

En suivant une approche similaire à celle vue en TD utilisant des tampons mémoire pour améliorer les performances des lectures, vous compléterez au fur et à mesure de vos besoins une structure `DIFF_FILE` à associer à un fichier ouvert :

```
struct diff_file {
    int fd;
    char line[MAX_LINE_SIZE];
```

```
    char buf[BUF_SIZE];
    ...
} DIFF_FILE;
```

Attention : la bibliothèque permet juste de lire un fichier, mais pas d'en écrire.

Indice : vous définirez les fonctions auxiliaires qui vous paraîtront pertinentes pour simplifier la programmation. Vous pourrez vous inspirez de celles définies en TD.

Q 9. Écrivez la fonction pour lire une ligne de texte :

```
char *diff_getline(DIFF_FILE *f);
```

La fonction retourne `NULL` si on est à la fin du fichier. Sinon, elle remplit la chaîne `f->line` et retourne un pointeur vers cette chaîne.

Q 10. Écrivez la fonction pour sauvegarder la position (*offset*) de la dernière ligne déjà lue :

```
void diff_savepos(DIFF_FILE *f);
```

Q 11. Écrivez la fonction pour ramener le fichier à la dernière position sauvegardée :

```
void diff_resetpos(DIFF_FILE *f);
```

Q 12. Écrivez la fonction pour imprimer sur la sortie standard toutes les lignes comprises entre la position sauvegardée et la position courante :

```
void diff_printlines(DIFF_FILE *f);
```

Q 13. Donnez la définition complète de la structure `DIFF_FILE` et écrivez le code des fonctions pour ouvrir et fermer un fichier :

```
DIFF_FILE *diff_open(char *name);
void diff_close(DIFF_FILE *f);
```

2.2 Éclaircissement d'image

Dans cet exercice on réalise un programme permettant d'éclaircir une image. Le problème est résolu en multipliant chaque composante de la couleur (rouge, vert, bleu) par un coefficient réel plus grand que `1.0`.

Nous considérerons un codage très courant des images, où chaque composante est codée par un entier compris entre 0 et 255 (soit sur 1 octet).

Une image est simplement une matrice de pixels (un tableau de tableaux, voir en annexe le code de création d'une image) et chaque pixel est un `int`. En supposant qu'un `int` est codé sur 4 octets, la composante rouge est codée par le deuxième octet, la composante verte par le troisième octet, et la composante bleue par le dernier octet.

On donne ci-dessous le code d'une version non multithread de l'algorithme.

```
#define ROUGE 0
#define VERT  1
#define BLEU  2

typedef int pixel;

void eclaircir_image(pixel **image, int largeur, int hauteur, float coeff) {
    for (short composante = 0; composante <= 2; composante++)
        eclaircir_composante(image, largeur, hauteur, composante, coeff);
}
```

Cet algorithme procède en trois passes : il éclaircit d'abord la composante rouge puis la composante verte et enfin la composante bleue.

La fonction qui éclaircit une composante est la suivante :

```
void eclaircir_composante(pixel **image, int largeur, int hauteur, short composante, float coeff) {
    for (int i = 0; i < largeur; i++)
        for (int j = 0; j < hauteur; j++)
            image[i][j] = eclaircir_pixel(image[i][j], composante, coeff);
}
```

La fonction qui éclaircit un pixel n'est pas donnée ici mais son fonctionnement se résume à aller modifier l'octet correspondant à la composante dans l'entier qui représente un pixel.

Q 14. Donnez la structure de données qui sera nécessaire pour passer les arguments à la version multithread de `eclaircir_image`.

Q 15. Écrivez une version multithread nommée `eclaircir_image_mt` de la fonction `eclaircir_image` qui lancera la fonction `eclaircir_composante` sur chacune des composantes dans des threads indépendants. Vous modifierez la fonction `eclaircir_composante` si nécessaire.

La comparaison entre les versions séquentielle et multithread donne les résultats expérimentaux suivants (sur un processeur possédant au moins 3 cœurs, temps exprimés en secondes) :

hauteur	largeur	temps séquentiel	temps multithread
10	10	0.000	0.002
20	20	0.000	0.009
40	40	0.000	0.020
80	80	0.000	0.074
160	160	0.002	0.256
320	320	0.008	1.042
640	640	0.035	4.193

Alors que le nombre de cœurs est suffisant pour exécuter en parallèle les 3 threads, le temps de calcul n'est pas celui escompté.

Q 16. Pourquoi le temps de la version multithread est-il si mauvais ? Les résultats sont-ils en accord avec le code que vous avez écrit ?

Un observateur extérieur propose alors de coder un pixel par un tableau de trois `short` plutôt qu'un `int`.

```
typedef short *pixel;
```

La création d'une image avec ce type de pixel est donnée en annexe. La fonction `eclaircir_pixel` s'écrit alors :

```
short eclaircir_pixel(short valeur_composante, float coeff) {
    return min(255, (short)(valeur_composante * coeff));
}
```

et la fonction `eclaircir_composante` :

```
void eclaircir_composante(pixel **image, int largeur, int hauteur, short composante, float coeff) {
    for (int i = 0; i < largeur; i++)
        for (int j = 0; j < hauteur; j++)
            image[i][j][composante] = eclaircir_pixel(image[i][j][composante], coeff);
}
```

La fonction `eclaircir_image` ne change pas.

Q 17. La fonction `eclaircir_image_mt` écrite précédemment doit-elle être modifiée ? Si oui, donnez la nouvelle version. Sinon justifiez. Vous modifierez la fonction `eclaircir_composante` si nécessaire.

La comparaison entre les versions séquentielles et multithread dans les mêmes conditions expérimentales donne les résultats suivants :

hauteur	largeur	temps séquentiel	temps multithread
10	10	0.000	0.000
20	20	0.000	0.000
40	40	0.000	0.000
80	80	0.000	0.000
160	160	0.000	0.000
320	320	0.002	0.001
640	640	0.008	0.004

Q 18. Expliquez pourquoi la version multithread est cette fois plus rapide que la version séquentielle.

On décide une dernière expérimentation où la transformation est effectuée de manière récursive :

```
void eclaircir_image(pixel **image, int largeur, int hauteur, float coeff) {
    if ((largeur) <= 10) {
        for (short composante = 0; composante <= 2; composante++) {
            eclaircir_composante(image, largeur, hauteur, composante, coeff);
        }
    } else {
        eclaircir_image(image, largeur / 2, hauteur, coeff);
        eclaircir_image(image + (largeur / 2), largeur - (largeur / 2), hauteur, coeff);
    }
}
```

Q 19. Donnez une version multithread de la fonction `eclaircir_image`.

La comparaison entre les versions séquentielle et multithread donne les résultats expérimentaux suivants :

hauteur	largeur	temps séquentiel	temps multithread
10	10	0.000	0.000
20	20	0.000	0.000
40	40	0.000	0.000
80	80	0.000	0.000
160	160	0.000	0.000
320	320	0.004	0.002
640	640	0.024	0.007

Q 20. Commentez ces résultats.

Annexe

```
typedef int pixel;

image = (pixel **) malloc(sizeof(pixel *) * largeur);
for (int i = 0; i < largeur; i++)
    image[i] = (pixel *) malloc(sizeof(pixel) * hauteur);

for (int i = 0; i < largeur; i++) {
    for (int j = 0; j < hauteur; j++) {
        pixel p = 0;
        p = p | (128 << 16);
        p = p | (128 << 8);
        p = p | (128 << 0);
        image[i][j] = p;
    }
}
```

(a) Utilisation d'un `int` par pixel

```
typedef short *pixel;

image = (pixel **) malloc(sizeof(pixel *) * largeur);
for (int i = 0; i < largeur; i++)
    image[i] = (pixel *) malloc(sizeof(pixel) * hauteur);

for (int i = 0; i < largeur; i++) {
    for (int j = 0; j < hauteur; j++) {
        pixel p = (pixel) malloc(3 * sizeof(short));
        p[0] = 128;
        p[1] = 128;
        p[2] = 128;
        image[i][j] = p;
    }
}
```

(b) Utilisation de 3 `short` par pixel

FIG. 1: Code de création d'une image suivant la représentation choisie pour un pixel