Licence 3 Informatique

Programmation des Systèmes

Devoir surveillé final

18 décembre 2015

Durée : 2h
Documents (notes de cours, mini-man, TD et TP) autorisés

Vous expliquerez votre code de sorte que votre intention (et pas seulement votre implémentation) soit claire. La gestion des erreurs pourra rester très sommaire (par exemple par des `assert`).

Vous trouverez au début de chaque question une estimation approximative du temps nécessaire pour y répondre (réflexion et longueur de la réponse) : court *(c)*, moyen *(m)* ou long *(l)*. Le barème accordera plus de points aux questions demandant le plus de temps.

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

1 Sous-répertoires et bugs

Répondre aux questions de cette section sur la première copie.

1.1 Listes de sous-répertoires

L'objectif de cet exercice est d'afficher la liste des sous-répertoires (mais pas des sous-sous-répertoires) contenus dans un répertoire. Prenons l'exemple de l'arborescence suivante :

```
$ ls -F -R
.:
a  b@  c/  f@  g/

./c:
d/

./c/d:
e

./g:
h

$ ls -l b f
lrwxrwxrwx 1 ray mond 1 déc 18 10:30 b -> a
lrwxrwxrwx 1 ray mond 1 déc 18 10:30 f -> c
```

Tous les noms font une lettre. L'option `-F` de `ls` permet d'indiquer par un caractère leur type :

— `@` pour un lien symbolique,
— `/` pour un répertoire,
— rien pour un fichier normal.

L'option `-R` permet d'afficher récursivement le contenu des répertoires imbriqués.

Nous voulons écrire une fonction `liste_sous_reps` telle que `liste_sous_reps(".")` affiche :

```
./c
./g
```

Q 1. *(l)* Définissez une fonction `void liste_sous_reps(char * repertoire)` qui affiche la liste des sous-répertoires de `repertoire` (pas les sous-répertoires des sous-répertoires, uniquement les sous-répertoires directs).

Si nous décidions de suivre les liens symboliques, nous voudrions afficher en plus `./f`.

Q 2. *(c)* Sans récrire toute la fonction, indiquez quelle(s) modification(s) faire pour suivre les liens symboliques.

1.2 Débogage

Voilà le code source d'un programme (en omettant les `#include`) :

```c
int main(int argc, char *argv[]) {
    int status;

    switch (fork()) {
        case -1: exit(EXIT_FAILURE);

        case 1:
            printf("Je suis le fils\n");
            exit(123);

        case 0:
            printf("Je suis le père\n");
    }

    assert(wait(&status) != -1);
    printf("Mon fils s'est terminé en retournant %d\n", status);
    return 0;
}
```

Ce programme compile sans erreur mais il n'a pas le comportement espéré. Voilà une exécution :

```
$ ./fork-bug
Je suis le père
fork-bug: fork-bug.c:22: main: Assertion `wait(&status) != -1' failed.
Mon fils s'est terminé en retournant 6
```

alors que nous espérions :

```
$ ./fork-correct
Je suis le père
Je suis le fils
Mon fils s'est terminé en retournant 123
```

Q 3. *(l)* Corrigez le programme.

Nous voulons maintenant enrichir l'information que le processus fils retourne à son père. Au lieu d'un petit entier, le fils retournera une structure de type `struct resultat_s`.

Q 4. *(c)* Indiquez combien de bits un fils peut transmettre à son père en utilisant `exit`.

Nous allons utiliser un tube pour transmettre la structure `res` du fils au père. Voilà le schéma général du code :

```
int main(int argc, char *argv[]) {
    struct resultat_s res;

    /* (1) */

    switch (fork()) {
        ...
            /* Code du fils */
            /* ... */
            /* Le fils écrit le résultat dans res */

            /* (2) */
            exit(EXIT_SUCCESS);
            /*************************/

        ...
            /* Code du père */
            /* (3) */
            assert(wait(NULL) != -1);
    }

    return 0;
}
```

Q 5. *(l)* Indiquez le code à écrire aux positions `(1)`, `(2)` et `(3)`. Il faudra :
- créer un tube et fermer ses deux extrémités aux bons endroits,
- que le fils écrive la structure dans le tube,
- que le père lise une structure depuis le tube.

2 Calcul de π

Répondre aux questions de cette section sur la deuxième copie.

Nous allons calculer des valeurs approchées de π de plusieurs façons, en utilisant des *threads* pour paralléliser le calcul et exploiter ainsi toute la puissance de calcul de la machine. Nous ne nous intéresserons pas ici aux détails de l'algorithme de calcul, il suffit de savoir que nous disposerons d'une fonction

```
long double pi(long n);
```

qui retourne une valeur approchée de π. Plus `n` est grand, plus la valeur est proche de π.

2.1 Un *thread* par valeur de `n`

Nous allons commencer en créant un *thread* pour chaque `n`. Cependant `pi` n'a pas le prototype requis pour la fonction principale d'un *thread*. Nous allons donc définir une fonction intermédiaire ayant le bon prototype et chargée de déclencher `pi`.

Q 6. *(m)* Définissez une fonction `th_pi` ayant le prototype d'une fonction principale de *thread*. Cette fonction doit juste déclencher `pi` sur le bon `n` et récupérer le résultat.
Vous définirez les types de structures que vous jugerez nécessaires.

Nous allons utiliser cette fonction en créant une fonction principale

```
void pi_tous_n(long n_max)
```

qui :

- pour tout n compris entre 0 et `n_max`, crée un *thread* pour calculer la valeur approchée de π pour ce n,
- attend la fin du calcul de tous ces *threads* et affiche leur résultat.

Q 7. *(m)* Définissez la fonction `pi_tous_n`.

2.2 *Threads* synchronisés

Créer et supprimer des *threads* prend du temps et de la mémoire. Il est donc plus efficace de demander à un *thread* d'effectuer plusieurs calculs. Nous voulons alors procéder de la manière suivante :

- nous utiliserons une variable globale `prochain_n`, partagée entre tous les *threads*,
- chaque *thread*, avant de commencer son calcul, consultera cette variable partagée (et l'incrémentera pour le prochain *thread*).

La variable `prochain_n` sera donc lue et écrite par différents *threads*.

Q 8. *(l)* Expliquez quels problèmes posent l'accès à des variables partagées. Que devez-vous faire pour contourner ce problème ? Vous expliquerez les variables dont vous avez besoin, les fonctions que vous devez appeler et quand vous devez les appeler.

Plus `n` est grand, plus le résultat est proche de la valeur de π. Nous pouvons utiliser cette propriété pour arrêter le calcul : nous mémoriserons dans une variable globale `approx` la dernière approximation de π calculée. Si l'écart entre `approx` et le nouveau résultat est inférieur à `seuil`, le *thread* s'arrêtera.

Chaque *thread* exécutera les opérations suivantes en boucle :

(a) la copie de la valeur de `prochain_n` à calculer dans une variable locale,
(b) l'incrément de cette valeur pour le *thread* suivant,
(c) le calcul de la valeur approchée de π,
(d) l'affichage de cette valeur si sa différence avec `approx` est inférieure à `seuil`,
(e) l'écriture de la nouvelle valeur d'`approx`.

Chaque *thread* s'arrêtera quand il aura affiché son résultat (étape (d)).

Il est très important que tous les *threads* puissent exécuter en parallèle leur étape (c) !

Q 9. *(m)* Précisez où vous devez ajouter des sections critiques dans les différentes étapes de calcul des *threads*.

Q 10. *(l)* Définissez une fonction `void *th_multi_pi(void *)` qui effectue la boucle d'opérations d'un *thread*.

Q 11. *(l)* Définissez la fonction `void pi_threads(long nb_threads, long double seuil)` qui crée `nb_threads` *threads* exécutant la fonction `th_multi_pi` et attend qu'ils se terminent. Cette fonction initialisera au passage les variables globales qui doivent être initialisées.

2.3 Annexe

Voici une implémentation possible de la fonction `pi`, pour faire des tests chez vous :

```
long double pi(long n) {
    long double somme = 0, x;
    long i;

    for (i = 0; i < n; i++) {
        x = ((long double)i + 0.5) / (long double)n;
        somme += 4.0 / (1 + x * x);
    }

    return somme / n;
}
```