TP4 : Multitâche Linux et threads POSIX

Introduction

Dans ce TP, vous allez manipuler différentes possibilités de la bibliothèque `pthread`, qui permet de programmer en C avec des processus légers. Il s'agit d'illustrer l'une des motivations de la programmation concurrente, le gain de performances à l'exécution.

Remarques
- Lisez l'énoncé en entier avant de commencer à travailler. Ça ne coûte rien et ça vous fera probablement gagner du temps par la suite.
- Tous vos programmes devront être écrits uniquement en C, et utiliser uniquement les fonctions de la bibliothèque standard C (i.e. pas de C++, pas de bibliothèques tierce-partie, et pas de code copié-collé sur le web). Pour automatiser la construction des exécutables (compilation, édition de lien), vous écrirez un `Makefile`.
- À chaque fois que le sujet demande d'écrire une fonction, il vous faut non seulement implémenter l'algorithme demandé, mais aussi vous assurer que cette implémentation est correcte ! Vous aurez donc typiquement à écrire une fonction `main()` et/ou des fichiers de configuration, à exécuter votre programme, et à faire la mise au point (*debugging*), etc.

1 Décomposition en facteurs premiers

Question 1 Écrivez une fonction `int is_prime(uint64_t p)` qui prend comme paramètre un entier positif p sur 64 bits, et qui rend 1 lorsque p est un nombre premier et 0 lorsque p est non-premier.

Remarque Pour savoir si un nombre a est un multiple d'un autre nombre b, vous pouvez utiliser l'opérateur *modulo*. En d'autres termes, b divise a si et seulement si `a%b == 0`.

Question 2 Écrivez une fonction `void print_prime_factors(uint64_t n)` qui prend en paramètre un entier positif n et qui affiche la liste des facteurs premiers de la factorisation de n. Par exemple, un appel `print_prime_factors(84)` devra afficher :

```
84: 2 2 3 7
```

On suppose maintenant l'existence d'un fichier texte nommé `numbers.txt` et contenant un nombre sur chaque ligne, par exemple :

```
27166
1804289
168150
8469308
```

Dans ce TP, vous pouvez faire l'hypothèse simplificatrice que ce fichier existe toujours et qu'il a toujours le bon format : un nombre par ligne, et rien d'autre. Ce ne sera donc pas la peine de gérer les cas d'erreur, comme l'absence du fichier, les erreurs de syntaxe, etc. Vous pouvez aussi faire l'hypothèse que les nombres en question sont suffisamment petits pour être représentés par un `uint64_t` dans votre programme.

Question 3 Écrivez un programme qui lit ce fichier, et qui pour chaque ligne affiche la liste des facteurs premiers du nombre en question. Vous aurez besoin pour cela des fonctions `fopen()` et `fscanf()`.

Avec le fichier donné en exemple ci-dessus, votre programme devra afficher :

```
27166: 2 17 17 47
1804289: 127 14207
168150: 2 3 5 5 19 59
8469308: 2 2 389 5443
```

Dans la suite du TP, nous allons nous intéresser au temps nécessaire à votre programme pour calculer ces diviseurs. Pour cela, vous pouvez invoquer votre programme au travers de l'utilitaire `time`, en tapant à l'invite du shell la commande `time ./monProg` (pour plus d'information sur le mode d'emploi de `time`, tapez la commande `man time`). L'utilitaire `time` vous affiche plusieurs informations différentes sur l'exécution de votre programme. C'est la *durée totale d'exécution* qui nous intéresse aujourd'hui (i.e. la durée `real`, et non pas la durée `user` ni `sys`).

Avec les nombres donnés en exemple ci-dessus, votre programme n'aura pas beaucoup de calculs à faire, et s'exécutera typiquement en une fraction de seconde. Pour observer une durée d'exécution qui soit significative, vous pouvez augmenter la quantité de nombres à factoriser, et surtout augmenter la taille des nombres en question. Pour vous y aider, vous trouverez en annexe de ce sujet un petit programme qui génère des nombres aléatoires en quantité arbitraire. Il vous suffit de rediriger sa sortie dans votre fichier texte.

2 Boucle parallèle

Nous allons maintenant étudier différentes manières de paralléliser ce programme, dans le but d'accélerer son exécution.

Dans un contexte monoprocesseur, les threads sont typiquement utilisés comme des outils pour la conception des programmes complexes. Ils permettent de séparer la programmation de différentes «activités» identifiées par le concepteur comme indépendantes. L'utilisation des threads n'a alors pas de lien direct avec les performances du programme à l'exécution.

Sur une machine multi-coeur (ou multiprocesseur) au contraire, différents threads peuvent réellement être exécutés simultanément. Un découpage judicieux des activités du programme en threads permet donc d'exploiter tous les coeurs de la machine, et donc d'améliorer sensiblement les performances du programme. C'est dans cette optique que nous allons transformer le programme dans la suite du TP.

Question 4 Modifiez votre programme pour qu'il traite maintenant deux nombres à chaque itération au lieu d'un. Pour cela, plutôt que d'appeler `print_prime_factors()` directement, votre boucle principale créera deux threads à chaque tour, à l'aide de `pthread_create()`, et attendra que ces deux threads se soient terminés avant de passer à l'itération suivante (à l'aide de `pthread_join()`).

Remarque C'est l'ordonnanceur du système d'exploitation qui décide quand suspendre l'exécution du thread en cours pour exécuter un des autres threads prêts. Il ne vous garantit pas dans quel état est un thread (à quel point de son exécution il en est) au moment où il décide de l'interrompre. Du coup, les affichages simultanés de vos différents threads sortiront mélangés sur la sortie standard. Nous allons ignorer ce problème pour l'instant, et nous y reviendrons plus loin dans le TP.

Question 5 Comme vous l'avez fait pour la version séquentielle, mesurez les temps d'exécution à l'aide de l'utilitaire `time`.

3 Worker threads

Le programme que vous avez écrit à la partie 2 est logiquement plus rapide que le programme séquentiel, puisque deux threads peuvent travailler simultanément sur votre machine bi-coeur. Mais les deux coeurs ne sont pas toujours utilisés efficacement : même si l'un de vos deux threads termine très rapidement son calcul, votre fonction `main()` doit toujours attendre que les 2 threads se soient terminés avant d'en relancer 2 nouveaux.

Question 6 Modifiez votre programme pour déplacer la boucle principale à l'intérieur des threads parallèles. Chaque thread ressemblera beaucoup au programme séquentiel initial, sauf en ce qui concerne la lecture dans le fichier : maintenant, il s'agit d'une *section critique*. En effet, comme la fonction `printf()`, la fonction `fscanf()` n'est pas atomique, vos threads doivent donc utiliser un verrou (fonctions `pthread_mutex_init()` , etc.) pour s'assurer qu'ils l'invoquent en exclusion mutuelle.

Question 7 À nouveau, observez le temps d'exécution de ce nouveau programme et comparez-le aux versions précédentes.

4 Mémorisation des résultats intermédiaires

Pour accélérer encore notre programme, nous allons utiliser une technique appelée *mémoisation*[1], qui consiste tout simplement à garder en mémoire les résulats intermédiaires, pour éviter d'avoir à les recalculer plus tard.

Par exemple, une fois qu'on a factorisé 42 en $2 \cdot 3 \cdot 7$, on peut mémoriser ce résultat quelque part dans une structure de données. Si dans le futur, on doit factoriser le nombre 84, alors après la première division par 2 on se retrouve avec le nombre 42, pour lequel on connait déjà la réponse ! Algorithmiquement parlant, il s'agit de troquer une augmentation de l'occupation mémoire pour une réduction du temps d'exécution. Bien sûr, cette technique n'est rentable que s'il y a des répétitions dans les nombres à traiter, et que rechercher un résultat précédent est plus rapide que le recalculer. Le générateur fourni en annexe vous permet de choisir le niveau de redondance voulu.

Pour pouvoir enregistrer nos résultats intermédiaires, il faut d'abord qu'ils soient stockés quelque part dans la mémoire. Pour l'instant ce n'était pas le cas, puisque votre programme affiche les facteurs premiers au fur et à mesure qu'il les trouve.

Question 8 Écrivez une fonction `int get_prime_factors(uint64_t n, uint64_t* dest)` qui, au lieu d'afficher les facteurs premiers sur la console, les place dans un tableau `dest`, et renvoie le nombre de facteurs trouvés. La fonction `print_prime_factors()` peut maintenant s'écrire comme :

```
void print_prime_factors(uint64_t n)
{
    uint64_t factors[MAX_FACTORS];

    int j,k;

    k=get_prime_factors(n,factors);

    printf("%llu: ",n);
    for(j=0; j<k; j++)
    {
        printf("%llu ",factors[j]);
    }
    printf("\n");
}
```

Remarque Vous pouvez vous simplifier la vie et ne manipuler que des tableaux de taille fixe (il s'agit de la constante `MAX_FACTORS` dans le code ci-dessus). Il vous suffit pour cela de calculer cette constante une fois pour toutes, comme le nombre maximal de facteurs premiers que peut comporter un entier 64 bits.

Question 9 Implémentez une structure de données partagée entre vos différents threads, dans lesquels vous mémoriserez les résultats obtenus pour ne pas avoir à les recalculer plus tard. Vous devrez bien sûr synchroniser les accès concurrents à cette structure à l'aide de verrous, de façon à vous assurer qu'elle ne soit pas corrompue.

5 Facultatif : course de vitesse

Dans ce TP, on a mis en pratique quelques techniques de l'algorithmique parallèle, et on a observé que les résultats en termes de performance sont spectaculaires. Mais ce sujet est loin de faire le tour de toutes les optimisations possibles. Si vous avez terminé la partie 4 et qu'il vous reste du temps, n'hésitez pas à chercher d'autres facons d'accélérer votre programme, et à organiser une course de vitesse entre les différents binômes !

1. voir `http://en.wikipedia.org/wiki/Memoization`

Annexes

Voilà un programme qui génère une quantité arbitraire de nombres aléatoires, de taille arbitraire. Vous pouvez rediriger sa sortie dans un fichier, que vous utiliserez ensuite comme entrée de vos programmes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main(int argc, char *argv[])
{
    uint64_t number ;
    uint32_t  * word = (void*) & number ;
    uint64_t *previous_numbers;

    // how many numbers to generate
    int quantity = 20;
    if( argc > 1)
        quantity=atoi(argv[1]);

    // maximum magnitude of numbers, in bits (0..64)
    int magnitude= 64;
    if( argc > 2)
        magnitude=atoi(argv[2]);

    // percentage of redundancy (0..100)
    // 30% means each number only has 2/3 chance to be a brand new one
    int redundancy=50;
    if( argc > 3)
        redundancy=atoi(argv[3]);

    // we seed the the generator with a constant value so as to get
    // reproducible results.
    srandom(0);

    previous_numbers=malloc(quantity*sizeof(uint64_t));

    int i;
    for(i=0; i<quantity; i++)
    {
        if( i==0 || random() % 100 > redundancy)
        {
            // let's generate a new number
            word[0] = random();
            word[1] = random();

            // shift right to reduce magnitude
            number >>= 64-magnitude ;
        }
        else
        {
            // let's pick from previously generated numbers
            number = previous_numbers[ random() % i ];
        }

        previous_numbers[i] = number;
        printf("%llu\n",(long long)number);
    }

    return 0;
}
```