Programmation multitâche sous Linux avec les 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), et leur exécution, vous pouvez écrire 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.
- En particulier, deux astuces :
  - n'attendez pas d'avoir «fini» de programmer avant de tester votre programme. Au contraire, dès que vous ajoutez ou modifiez une dizaine de lignes de code, il faut compiler et exécuter le programme pour faire apparaître les bugs.
  - Compilez toujours avec `-Wall` et corrigez votre programme jusqu'à éliminer tous les messages d'erreur et de warning !
- Gardez tout le code (correct) que vous écrivez : faites un fichier pour la question 1, un pour la question 2, etc. Cela vous sera utile pour comparer différentes versions de votre code.

1 Décomposition en facteurs premiers

Question 1 Écrivez une fonction `void print_prime_factors(uint64_t n)` qui prend en paramètre un entier positif *n* sur 64 bits 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
```

Remarques

- Si le paramètre `n` est lui-même un nombre premier, alors il se décomplose en un unique facteur. On s'attend donc à ce qu'un appel `print_prime_factors(101)` affiche le résultat suivant :
  `101: 101`
- 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`.
- Il est probable que certains nombres que vous manipulez (le nombre `n`, mais aussi les indices de boucle, etc) prennent des valeurs trop grandes pour le type `int`. Vous pouvez par contre utiliser le type `uint64_t` de `stdint.h`, que la fonction `printf()` sait afficher correctement si on lui précise le bon format (`%ju` ou `%llu` en fonction de votre chaîne de compilation)

Plusieurs nombres On suppose maintenant l'existence d'un fichier texte contenant un nombre sur chaque ligne, par exemple :

```
77
27166
1804289
168150
8469308
123456789123456789
```

Question 2 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 :

```
77: 7 11
27166: 2 17 17 47
1804289: 127 14207
168150: 2 3 5 5 19 59
8469308: 2 2 389 5443
123456789123456789: 3 3 7 11 13 19 3607 3803 52579
```

- 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 toujours représentables par un `uint64_t` dans votre programme.
- Quelques fonctions de la bibliothèque standard qui peuvent vous être utiles : `fopen()` pour ouvrir un fichier, `fgets()` pour lire une ligne, et `atoll()` pour convertir la chaîne lue en un nombre. Lisez la documentation de ces fonctions si vous n'êtes pas déjà familier avec !

Mesurer le temps d'exécution 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 (à l'exception du dernier), 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. Écrivez (à la main, ou avec le générateur, ou les deux) une liste de nombre que votre programme met entre 10s et 30s à traiter.

2 Boucle parallèle

Nous allons maintenant étudier différentes manières de paralléliser notre 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.

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

Question 3 Modifiez votre programme pour qu'il traite maintenant deux nombres simultanément 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()`).

- Pour utiliser les pthreads dans votre programme, vous devez invoquer `gcc` avec l'option spéciale `-pthread`. Vérifiez que votre Makefile est correct !
- Comme deux nombres sont maintenant décomposés simultanément, les facteurs sont affichés mélangés sur le terminal. Nous allons ignorer ce problème pour l'instant, et y revenir vers la fin du TP.

Question 4 Comme vous l'avez fait pour la version séquentielle, mesurez les temps d'exécution à l'aide de l'utilitaire `time`. Écrivez (et gardez pour plus tard) plusieurs listes de nombres, pour illustrer différents comportements. En particulier :

- Écrivez une liste de nombre que les deux versions de votre programme mettent le même temps à décomposer. Comment expliquez-vous ce comportement ?
- Écrivez une autre liste de nombre pour laquelle l'une des versions va significativement plus vite. Quel accélération peut-on observer au meilleur cas ?

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 5 (dans un nouveau fichier) 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 les opérations liées à la lecture dans le fichier. En effet, votre `FILE*` sera maintenant partagé entre ves deux worker threads, qui y feront des accès concurrents. On veut s'assurer que tous ces accès sont bien mutuellement exclusifs (il s'agit d'une *section critique*). En pratique vous devrez donc utiliser un verrou (fonctions `pthread_mutex_init()` , etc.) pour assurer l'exclusion mutuelle.

Question 6 À nouveau, observez le temps d'exécution de ce nouveau programme et comparez-le aux versions précédentes sur différents jeux de données. Essayez vos différentes listes de nombres précédentes pour comprendre quels aspects influencent les performances de cette nouvelle version.

On va maintenant s'occuper de rendre lisible à nouveau les informations affichées par votre programme. Pour ça, on va devoir distinguer deux choses : la décomposition des nombres proprement dites (qu'on veut continuer à faire en parallèle) et l'affichage des résultats (qu'on veut faire un nombre à la fois, c'est à dire, en exclusion mutuelle).

Question 7 Écrivez une fonction `int get_prime_factors(uint64_t n,uint64_t* dest)` qui n'affiche rien, mais stocke les facteurs premiers de `n` dans le tableau `dest`, et qui retourne le nombre de facteurs trouvés. Du coup, notre fonction `print_prime_factors()` peut maintenant être ré-écrite ainsi :

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

    int j,k;

    k=get_prime_factors(n,factors);

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

— Vous pouvez (pour vous simplifier) manipuler uniquement 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 8 Modifiez cette fonction `print_prime_factors()` pour que la décomposition des nombres soit faite en parallèle, mais l'affichage en exclusion mutuelle.

Question 9 Mesurez les temps d'exécution sur vos différents jeux de données. Que constatez-vous ?

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 rencontre de nouveau ce nombre, pas besoin de refaire le calcul. Ou même, si on doit plus tard 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 vous permet de choisir le niveau de redondance voulu.

Question 10 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. En tous cas, 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 !

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.

```
#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)
```

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

```
        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("%ju\n",number);
    }

    return 0;
}
```