TP - ASR7 Programmation Concurrente

—

Synchronisation et Ordonnancement

Sylvain Brandel, Guillaume Damiand, Laurent Lefèvre, Matthieu Moy, Grégoire Pichon

Printemps 2022

I Introduction

Pour ce TP, nous allons jouer avec les politiques d'ordonnancement temps-réel du noyau Linux (`SCHED_FIFO`, `SCHED_RR`). Activer ces politiques est potentiellement dangereux pour le système (une boucle infinie en priorité temps-réel peut figer l'ensemble du système), donc interdit pour des simples utilisateurs : vous aurez besoin de passer root pour le faire. Vous ne pourrez donc pas faire ce TP sur les machines physiques de l'université.

Nous allons utiliser l'infrastructure de *cloud computing* du département financée par la région Rhône-Alpes. C'est un ensemble de machines pilotées par le logiciel `OpenStack` avec lequel vous allez avoir un premier contact. Vous allez l'utiliser afin de créer une machine virtuelle, que vous pourrez conserver, effacer (mais perdre ainsi toutes les configurations effectuées), re-générer... La machine virtuelle (VM) créée va ici nous servir de base d'expérimentation, et dans le TP suivant d'entraînement à des manipulations d'administration systèmes que vous pourrez ensuite effectuer sur vos machines personnelles.

Cette année, les machines virtuelles ont été créées pour vous. Référez-vous au TP 1 : "II Accès aux machines virtuelles" si vous ne vous rappelez plus comment y accéder. L'adresse IP et le mot de passe sont disponibles sur TOMUSS.

II Modification de l'ordonnanceur

Dans cet exercice, vous allez utiliser explicitement l'ordonnanceur du noyau Linux. C'est assez dangereux car un processus prioritaire qui ne s'arrête pas, par définition, bloque l'ordinateur. Vous devez donc utiliser les machines virtuelles.

Le code `sched.cpp` a été préparé, il lance un certains nombre de threads qui font des calculs en affichant de temps en temps des informations. Pour le moment, le code de la fonction `change_ordonnancement()` n'est pas fait et l'ordonnancement n'est pas modifié.

Q.II.1) - Complétez le code de cette fonction pour qu'elle change l'ordonnancement du thread qui l'appelle.

Il n'y a pas de fonctions C++ définies dans la bibliothèque standard pour manipuler l'ordonnanceur. Vous allez devoir utiliser les fonctions C suivantes pour faire ce travail :

— Retourne le numéro du thread qui l'appelle :

```
1     pthread_self();
```

— Récupérer les paramètres d'ordonnancement courant (elle vous permettra d'initialiser les variables) :

```
1     int pthread_getschedparam(pthread_t target_thread,
2                               int *politique,
3                               struct sched_param *param);
```

Le paramètre `politique` est passé par adresse (on n'a pas de passage par référence en C). Il n'est pas nécessaire (ou plus précisément : ce serait une aberration !) de faire une allocation dynamique pour autant. Une manière raisonnable d'appeler cette fonction est, tout simplement :

```
int anc_policy;
sched_param sched;
int r = pthread_getschedparam(pthread_self(), &anc_policy, &sched);
```

Si cela vous paraît compliqué, reprenez vos cours de L1/L2, ou lisez n'importe quel tuto sur le passage par adresse, c'est un concept fondamental en C et C++ !

— modifier la politique d'ordonnancement et ses paramètres :

```
1     int pthread_setschedparam(pthread_t target_thread,
2                               int politique,
3                               const struct sched_param *param);
```

Vérifiez la valeur de retour de la fonction (notamment pour vous assurer que vous avez bien les droits pour changer la politique et la priorité, il faut être root pour le faire). La valeur de la politique est définie dans des macros : `SCHED_FIFO`, `SCHED_RR` ou `SCHED_OTHER`.

Faites quelques expériences :

Q.II.2) - Lancez 3 threads RR, l'un de priorité 4 et deux autres de priorité 2.

Q.II.3) - Lancez 3 threads FIFO de priorité identique.

Q.II.4) - Lancez 1 threads FIFO et 2 RR de priorité identique.

Q.II.5) - Lancez 1 thread FIFO de priorité 99 et 2 autres FIFO de priorité plus faibles.

Q.II.6) - Pouvez-vous stopper le programme pendant que des threads très prioritaires tournent ? Pourquoi [1] ?

Les 4 premières questions ont un comportement qui semble conforme avec ce qui est attendu. La tâche de plus forte priorité gagne, à priorité égale, lorsque qu'une fifo commence elle se termine, les tâches RR s'alternent.

Mais on peut stopper le programme en cours ce qui est par contre anormal (il faut déjà pouvoir le contacter or cela nécessite d'utiliser la gestion du réseau, le programme ssh, le shell ... qui sont des tâches en temps partagé. Si on réfléchit bien, aucun des résultat n'est naturel car pour voir les affichages, il a bien fallu faire du réseau et cela aurait du être bloqué. En fait,

1. Pour vous inspirer, vous pouvez regarder les valeurs des 2 variables du noyau `/proc/sys/kernel/sched_rt_runtime_us` et `/proc/sys/kernel/sched_rt_period_us`

on peut s'apercevoir que quoi qu'on dise, les tâches `OTHER` sont parfois exécutées alors qu'il y a des tâches plus importantes en cours.

La raison de tout cela est que pour éviter les « freeze » du système, il y a une sécurité. Un certain pourcentage de temps CPU est réservé aux taches temps partagé. En fait, les tâches temps réelles ne peuvent utiliser que `sched_rt_runtime_us/sched_rt_period_us` du CPU. On peut avoir un comportement plus conforme en mettant -1 dans le fichier `/proc/sys/kernel/sched_rt_runtime_us`. Dans ce cas, en refaisant les expérience précédentes, le système est totalement bloqué tant que les processus `RR` ou `FIFO` ne sont pas terminés.

Q.II.7) - Si vous avez votre propre ordinateur, refaites les expériences sur ce dernier. Avez-vous les mêmes résultats ? Pourquoi ?

Il y a une bonne chance qu'il y ait une différence avec les FIFO qui s'alternent par exemple. Cela est du au fait que les ordinateurs sont multi-processeurs. Une tâche n'occupe donc pas tout le processeur.

Si on souhaite observer le même comportement, on peut forcer l'ensemble des threads à s'exécuter sur un seul cœur, avec par exemple :

```
1  cpu_set_t my_set;
2  /* Initialize it all to 0, i.e. no CPUs selected: */
3  CPU_ZERO(&my_set);
4  /* Set the bit that represents core 0: */
5  CPU_SET(0, &my_set);
6  /* Apply the setting: */
7  int s = sched_setaffinity(0, sizeof(cpu_set_t), &my_set);
8  handle_error_en(s, "sched_setaffinity");
```

On peut également forcer un processus à s'exécuter sur un cœur particulier en utilisant `taskset -c 0 ./mon_programme`.

III Si le temps le permet, Intégration par la méthode des rectangles

III.1 Théorie

Les sommes de Riemann sont des sommes approximant des intégrales. Ainsi, si on considère $f : [a, b] \to \mathbb{R}$ une fonction partout définie sur le segment $[a, b]$; un entier $n > 0$ et une subdivision *régulière* définie pour $0 \leq k \leq n$ par :

$$x_k = a + k\frac{b-a}{n}$$

Alors la somme de Riemann est définie comme

$$S_n = \frac{b-a}{n} \sum_{k=1}^{n} f(a + k\frac{b-a}{n})$$

c-à-d

$$S_n = \sum_{k=1}^{n} (x_k - x_{k-1}) f(x_k)$$

Plus n est grand, plus [2] cette méthode des rectangles pour le calcul des intégrales donne une estimation proche de la valeur cherchée de l'intégrale

III.2 Pratique

Nous avons accès à des machines multi-cœur, et voulons en faire bon usage pour calculer des résultats d'intégration.

Q.III.1) - Avec ce qui a été vu en cours, vous devez proposer un programme qui sera capable de s'adapter au nombre de cœurs fourni par l'utilisateur en ligne de commande (de sorte que chacun soit utilisé pour calculer une partie d'intégrale). Le programme codé en C++ retournera la valeur de la somme totale calculée.
Vous pourrez utiliser des fontions à intégrer plus ou moins complexes pour vous assurer de la validité de votre code, et pour les questions suivantes.

Pas de corrigé détaillé pour cette question, mais le principe est le même que celui que nous avons déjà appliqué plusieurs fois :

- Créer un tableau de N nombres (`float`) qui contiendra les résultats des calculs de chaque thread. N est le nombre de cœurs utilisés pour le calcul. En général on a $N \ll n$.
- Lancer N threads, chacun chargé du calcul d'une partie de l'intégrale. Passer à chaque thread une case du tableau précédent, par référence (ou par pointeur).
- Chaque thread calcule l'intégrale sur un segment.
- La fonction `main` fait un `join()` sur chaque thread, puis fait la somme des éléments du tableau.

Q.III.2) - Donnez 2 moyens de savoir combien de cœurs dispose la machine que vous utilisez actuellement.

En ligne de commande : `cat /proc/cpuinfo`

En C : `sysconf(_SC_NPROCESSORS_CONF)`

Et expérimentalement, calculer le meilleur facteur d'accélération de performance en parallélisant le calcul.

Q.III.3) - À l'aide de la fonction `gettimeofday()`, vous mesurerez la durée du programme pour un nombre de cœurs valant 1, 2, 4, 8 et 64.
Commentez les résultats obtenus !

Q.III.4) - Vous pouvez trouver une solution en C utilisant OpenMP à l'URL `http://graal.ens-lyon.fr/~ycaniou/Teaching/1415/L3/integrale_OpenMP.c`. Quelles observations pouvez-vous faire ?

2. moyennant la prise en compte des erreurs d'arrondi latentes...

OpenMP est un outil de programmation parallèle qui permet de s'abstraire des détails vus dans cette UE (lancement des threads, répartition des tâches à effectuer sur les threads, ...) avec une syntaxe à base d'annotations (`#pragma` ajoutés dans un programme séquentiel).