Pablo Rauzy <pr@up8.edu>
https://pablo.rauzy.name/teaching/sr

Systèmes et réseaux

TP 2 : La rue et le temps

Exercice 0.

La rue.

1. À l'aide du chapitre 2 du cours et des pages de manuel, écrivez un programme qui affiche sur sa sortie standard le pid du processus qui adopte les processus orphelins.

Exercice 1.

Temps.

1. L'objectif de cet exercice est de produire une version grossière de la commande `time`.

   Quelques questions préparatoires :

   (a) Quel appel système vous permet d'attendre qu'un processus enfant ait terminé ?

   (b) Quels sont les champs présents dans la structure `timespec` ?

   (c) Quelle fonction de la bibliothèque C standard permet de remplir une structure `timespec` ?

   (d) Avec quel horloge faut-il appeler cette fonction pour obtenir le temps écoulé depuis le démarrage du système ?

   (e) Quelles variantes de `exec*` vont automatiquement chercher un exécutable dans le `PATH` ?

2. Écrivez un programme qui prend une commande en argument, et qui mesure, en microsecondes, le temps écoulé entre le lancement de cette commande et sa complétion.

Exercice 2.

Collision.

1. Le fichier `collision.c` qui vous est fourni cherche une collision partielle sur un condensat (*hash*). Un condensat est le résultat d'une fonction de hachage qui est déterministe, facile à calculer dans un sens, mais très difficile à inverser. Trouver une collision ne peut donc se faire que par force brute.

   La fonction de hachage qu'on utilise renvoie un condensat sur 32 bits. Notre programme cherche un condensat qui tiendrait sur une seul chiffre hexadécimal, qui sera donc en collision partielle (seulement sur les 24 premiers bits) avec le condensat à zéro.

   Pour cela le programme `collision.c` utilise un *nonce* (contraction de *number* et *once*) qu'il place avec les données à hacher et le modifie (en l'incrémentant) jusqu'à tomber sur un condensat qui correspond à ce qui est recherché.

   → Où est stocké le *nonce* dans le programme ?

2. Compilez et exécutez le programme, puis vérifiez que la sortie est bien :

   ```
   Data: 56857ea837960df006a3dfb23ae546b135de3d0d190038fa11141224
   Partial collision found with nonce 29b81762
   Hash is 0000000e
   ```

   → Combien de calculs de condensat ont été effectués ?

3. Ce problème est ce qu'on appelle *embarrassingly parallel*, c'est-à-dire qu'il est très facile à paralléliser parce que les calculs ne dépendent pas du tout les uns des autres. Ici par exemple on pourrait faire en parallèle dans deux processus séparés d'un côté les nonce pair et de l'autre les nonce impairs. Si les deux processus peuvent effectivement s'exécuter en même temps (sur une machine avec plusieurs cœurs typiquement), on devrait trouver la solution environ deux fois plus vite !

   On pourrait faire cela avec un `fork`, mais il n'y a aucune raison de s'arrêter à 2, et gérer le lancement de 8 ou même 16 ou 64 processus en parallèle avec `fork` est non seulement compliqué au niveau logistique dans le code, mais aussi coûteux en ressources.

   On décide donc d'utiliser des *processus légers* (**threads**) pour paralléliser ce calcul.

   (a) → Quelle fonction permet de créer un thread de calcul depuis le thread principal ?

(b) Quand l'un des threads de calcul trouve une solution, il lui faut un moyen de se synchroniser avec le thread principal.

→ Que peut-on utiliser pour cette synchronisation ?

(c) Une fois le thread principal notifié par le premier qui aura trouvé, il doit dire à tout les autres de s'arrêter, comment peut-il s'y prendre ?

(d) Il faut d'une part que les threads de calculs puissent être annulés brutalement en plein calcul, et d'autre part pouvoir empêcher celui qui a trouvé d'être annulé en même temps que les autres pour qu'il puisse renvoyer (avec `pthread_exit`) la valeur du nonce qui fonctionne.

→ Quels réglages d'annulation utiliser pour cela ?

(e) → Comment récupérer seulement la valeur du thread qui a trouvé quand on fait les `pthread_join` depuis le thread principal pour terminer les threads de calcul, c'est-à-dire, comment différencier les threads annulés de celui qui a trouvé ?

4. → Implémentez une version parallèle avec un nombre de threads qui dépend d'une constante `NB_TRHEADS` qui sera définit avec un `#define` dans votre code source.
5. À l'aide de votre outils `temps` créé à l'exercice précédent, observez l'évolution du temps de calcul sur votre machine en fonction du nombre de threads.

   → Qu'en déduisez-vous ?