Mesure du temps processeur et de l'espace mémoire

Le développement efficace consiste, lors de la programmation, à choisir des algorithmes et des structures de données économes en temps de calcul ou en espace mémoire, selon les besoins de l'application.

Les exemples seront en langages `C`/`C++` et Python (version 3).

L'objectif de cette séance est de découvrir les outils de profilage pour mesurer l'utilisation du temps processeur (CPU) ou de l'espace mémoire, ainsi que les bonnes et mauvaises pratiques...

1 Premières approches pour mesurer le temps et l'espace

Tout d'abord, il existe par exemple des commandes UNIX :

- `top` affiche la liste des processus les plus gourmands en temps CPU (colonnes `%CPU` et `TIME`) et en espace mémoire (colonne `MEM`). Le temps est le temps d'occupation du processeur par le processus. Ce n'est pas le temps réel (par exemple, un processus qui passe son temps à dormir n'occupe pas le processeur, s'il est bien écrit...).
- `time` permet d'afficher le temps réel, le temps utilisateur (occupation du processeur), et même le temps passé dans les appels système. Elle s'utilise ainsi sur un exécutable `main` :

```
time ./main
```

1.1 Chronomètre en langage `C`

Vous connaissez la bibliothèque `chrono`. Il est très simple d'en ré-implémenter une variante, avec les fonctions UNIX de base. Le programme `time_chrono.c` suivant définit ainsi deux fonctions permettant d'attendre un certain nombre de secondes (passé en paramètre), avec deux approches différentes :

- `bad_sleep()` attend de manière active (en monopolisant le processeur) que l'intervalle de temps demandé soit écoulé (à l'aide de la fonction UNIX `clock_gettime()`) ;
- `good_sleep()` met le processus en sommeil (en économisant le processeur) durant l'intervalle de temps demandé (grâce à la fonction UNIX `sleep()`).

Voici le fichier `time_chrono.c` implémentant ces fonctions :

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include "time_chrono.h"

void bad_sleep(int s)
{
  struct timespec before, after;

  clock_gettime(CLOCK_REALTIME, &before);

  while (1)  /* ATTENTE ACTIVE... */
    {
      clock_gettime(CLOCK_REALTIME, &after);

      double elapsed_secs = (after.tv_sec - before.tv_sec) +
                            (after.tv_nsec - before.tv_nsec) * 0.000000001;

      if (elapsed_secs > s) break;
    }
}

void good_sleep(int s)
{
  sleep(s);
}
```

Question 1 : (TD et TP) – Complétez le programme précédent avec les fonction `start()` et `stop()`, et écrivez le fichier d'en-tête `time_chrono.h` correspondant.

Question 2 : (TD et TP) – Écrivez dans un fichier `time_example.c` une fonction `countdown()` qui effectue un compte-à-rebours sur 10 secondes, en affichant chacune des secondes, ainsi qu'une fonction `main()` qui chronomètre ce compte-à-rebours.

Lors du TP, pour obtenir un exécutable (main), vous compilerez directement dans le terminal (sans IDE, c'est plus efficace...) en utilisant la commande

```
make
```

qui nécessite un fichier `Makefile` fourni dans Moodle.

1.2 Chronomètre en langage Python

Question 3 : (TD et TP) – Écrivez un fichier `time_chrono.py`, avec deux fonctions :
- `start()` qui démarre le chronomètre ;
- `stop()` qui stoppe le chronomètre et affiche le temps écoulé ;

en utilisant la fonction `time()` dans le module `time`, qui retourne l'heure actuelle, exprimée en secondes.

2 Découverte des outils de profilage

Un `profiler` est un programme sur-couche permettant d'analyser la consommation en temps ou en mémoire d'un programme de base.

2.1 Profilage CPU pour le `C`

Pour le langage `C`, historiquement c'est la commande `gprof`, le *profiler* du projet GNU, qui était utilisée. Nous allons utiliser les `gperftools` (Google Performance Tools). En TP, vous trouverez des détails à l'adresse suivante :

`https://gperftools.github.io/gperftools/cpuprofile.html`

mais nous nous contenterons d'utiliser le fichier `time_profiler.sh` (script *shell* exécutable) suivant :

```
CPUPROFILE=main.prof LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libprofiler.so.0 ./main
google-pprof --pdf ./main main.prof > time-profile.pdf
rm -f main.prof
```

qui génère le profil temporel (fichier `time-profile.pdf`) de l'exécutable `main`.

Attention : ce code est pour Linux.
Pour MacOS X, il faut remplacer `LD_PRELOAD` par `DYLD_INSERT_LIBRARIES` et `.so` par `.dylib`.
De plus, `google-pprof` s'appelle tout simplement `pprof`,
et le chemin `/usr/lib/x86_64-linux-gnu/` devra être changé lui aussi,
a priori par `/usr/local/lib/gperftools/`.

2.2 Profilage mémoire pour le `C`

Avec les mêmes outils, il est possible de générer le profil mémoire. En TP, vous trouverez des détails à l'adresse suivante :

`https://gperftools.github.io/gperftools/heapprofile.html`

mais nous nous contenterons d'utiliser le fichier `memory_profiler.sh` (script `shell` exécutable) suivant :

```
HEAPPROFILE=main.prof LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libtcmalloc.so.4 ./main
google-pprof --pdf ./main main.prof.*.heap > memory-profile.pdf
rm -f main.prof.*.heap
```

qui génère le profil mémoire (fichier `memory-profile.pdf`) de l'exécutable `main`.

En fait, l'espace mémoire total est composé d'un tas *(heap)* et d'une pile *(stack)* où se trouvent les variables temporaires. Mais nous verrons cela plus tard...

2.3 Profilage CPU pour Python

Question 4 : (TD et TP) – Écrivez dans un fichier `time_example.py` une fonction `countdown()` qui effectue un compte-à-rebours sur 10 secondes, en affichant chacune des secondes.

Le fichier `time_profiler.py` (basé sur le module `cProfile`) est disponible dans Moodle.

Question 5 : (TP) – Utilisez le chronomètre et le *profiler* pour mesurer le temps pris par ce compte-à-rebours.

Pour exécuter un programme Python `main.py`, il suffit de taper :

```
python main.py
```

(attention, pour Python 3, la commande peut s'appeler par exemple `python3`).

2.4 Profilage mémoire pour Python

Pour installer un module Python, utiliser la commande `pip` (attention, pour Python 3, la commande peut s'appeler par exemple `pip3`). Par exemple, pour utiliser le module de profilage mémoire, faire :

```
pip install memory_profiler
```

ou

```
python -m pip install memory_profiler
```

On considère le fichier `memory_example.py` suivant :

```
from memory_profiler import profile

@profile
def example():
    a = [1] * (2 ** 20)
    b = [2] * (2 * 10 ** 7)
    del b  # try to comment this line...
    return a

example()
```

sachant que la syntaxe `[v] * n` permet de créer une liste contenant `n` fois l'élément `v`.

Question 6 : (TD, à vérifier en TP) – Combien d'octets nécessite la variable `a` ?

Question 7 : (TD, à vérifier en TP) – Combien d'octets nécessite la variable `b` ?

La commande `del` permet de détruire une variable, et de libérer immédiatement la mémoire qu'elle occupait. Elle est facultative, car par défaut Python utilise un ramasse-miettes *(garbage collector)* qui libère la mémoire non utilisée, mais pas immédiatement, car ce n'est pas forcément efficace...

3 Exemple de profilage CPU : attente active ou mise en sommeil ?

Question 8 : (TP) – Faites le profil temporel du programme `time_example.c`, en utilisant dans ce dernier soit la fonction `bad_sleep()` soit la fonction `good_sleep()`. Que remarquez-vous ?

4 Exemple de profilage mémoire : *delete* ou pas ?

Question 9 : (TP) – Faites le profil mémoire du programme `memory_example.py`, en commentant ou pas la ligne comportant la commande `del`. Que remarquez-vous ?