DU ISN 2019-2020
TP Systèmes - Threads

Ce TP a pour but de vous faire découvrir les processus légers, les *threads*. Il est construit autour des transparents de cours fournis de manière séparée. Les points traités sont :

- Motivation des *threads* (pourquoi la programmation multi-threadée est importante?)
- Compréhension de la nature des *threads* et différence entre *threads* et processus
- Programmation avec *threads*
- Découverte des problèmes de synchronisation dans les programmes multi-threadés

I. Motivation des *threads*

Les processus que nous avons traités lors des deux séances précédentes sont appelés des *processus lourds*. Cette appellation vient du fait que les processus sont des entités qui demandent une quantité de ressources non négligeables (mémoire) et dont les fonctions de gestion prennent du temps. En effet, toute opération sur les processus (création, terminaison, changement d'état, ordonnancement, ...) est effectuée par le système, implique un changement de contexte/de mode et est coûteuse en temps. De plus, les processus, par définition, sont des entités isolées i.e. ne partagent pas de mémoire. Si cette caractéristique les rend sécurisés (un processus ne peut perturber l'exécution d'un autre processus), elle rend difficile la coopération entre processus.

Or, avec le développement des architectures multi-coeur, les architectures matérielles standard utilisées sont des architectures parallèles. Les applications deviennent naturellement elles aussi parallèles. En effet, la majorité des services utilisés (p.ex. tous les services WEB) sont capables de gérer plusieurs activités en même temps. Un exemple typique est de pouvoir gérer plusieurs requêtes provenant de différents clients en même temps. Dans ce cas, toutes les requêtes exécutent le même code et probablement partagent des données (travaillent sur des données ou avec des ressources partagées).

II. Processus versus threads

Les *threads* viennent donc répondre à ces besoins applicatifs et proposer une alternative aux limités des processus. Ainsi, les threads sont définis comme des entités plus légères, demandant moins de ressources et qui peuvent travailler sur des données partagées. En réalité, les threads sont des fils d'exécution concurrents s'exécutant au sein de l'espace mémoire d'un processus. En d'autres termes, les threads n'existent pas en dehors d'un processus. Ils partagent donc le code mais ont chacun leur propre pile et compteur de programme.

Les slides 4,5,6,7,8 parlent un peu plus de l'implémentation des threads.

Exercice 1

Lire les slides 9,10 et 11 qui donnent un premier exemple (`0_threads.py`) de programmation multi-threadée en Python. Comprendre la structure du programme, exécuter, observer. Que pouvez-vous dire en ce qui concerne l'ordonnancement des threads?

Exercice 2

Etudier l'exemple 2 (`1_threads.py`) qui crée plus de threads qui s'exécutent plus longtemps. Quid de l'ordonnancement? Combien de threads s'exécutent au total?

Exercice 3

Etudier l'exemple 3 (`join_threads.py`) qui vous montre comment un thread peut attendre la terminaison d'un autre. Lire le programme, comprendre|exécuter. Cela vous fait-il penser à des fonctions similaires au niveau des processus?

Exercice 4

Avec les processus nous avons vu la fonction `fork()` et avons parlé de la relation père-fils. Nous avons également vu qu'un père est obligé de se préoccuper de la terminaison d'un fils, sinon des *zombies* apparaissent. La situation est-elle la même dans le cas des threads? En particulier, a-t-on une relation père-fils? La terminaison du thread main a-t-elle une repercussion "spéciale" sur l'exécution des autres threads?

Exercice 5

Etudier l'exemple `pb_file_threads.py` (slides 17,18,19). Quel est le problème? Comment peut-on le résoudre?

Exercice 6

Etudier l'exemple `pb_threads.py` (slides 20,21,22). Exécuter. Arrivez-vous à reproduire le problème? Est-ce déterministe? Quelle est l'explication?

Exercice 7

Pour résoudre le problème précédent, les threads ont besoin de synchronisation. Les *verrous* sont une structure de synchronisation simple. Découvrez leur utilisation avec `lock_threads.py/py3.`

Exercice 8

Non seulement nous sommes obligés à faire attention au partage de données entre threads et donc à faire de la synchronisation mais aussi nous devons faire attention à la manière dont on fait (de la synchronisation)! Découvrez le problème d'interblocage illustré avec `deadlock_threads.py.`¶

1. Pour démarrer

Lancer un terminal. Dans ce terminal, vous devez voir un prompt qui vous invite à taper des commandes.

2. Où est-vous ? Chemins dans l'arborescence de fichiers.

En lançant un terminal, si vous ne faites rien de spécial, le répertoire dans lequel vous vous trouvez (répertoire courant) correspond à votre "maison" (typiquement quelque chose comme `/home/users/nom-de-login`).

Pour vérifier où est votre "maison", vous pouvez vérifier la valeur de la variable d'environnement `HOME` avec
`echo $HOME`

Pour voir où vous vous trouvez dans l'arborescence de fichier, taper
pwd

Cette commande donnera l'emplacement du répertoire courant à partir de la racine de l'arborescence (la racine est marquée /). Ce chemin est appelé *chemin absolu*. Un *chemin relatif* est un chemin qui considère un emplacement à partir du répertoire courant.

Le répertoire courant est désigné par `.`
Le répertoire père est désigné par `..`
Le répertoire de votre "maison" est désigné par `~`

3. Changer de répertoire courant (cd)

Pour changer de répertoire courant, la commande à utiliser est cd (change directory)
Essayer par exemple (ce sont des commandes qui utilisent des chemins relatifs)

```
cd .
cd ..
cd ~
```

Pour tester les chemins absolus

```
cd /usr/bin
cd le_chemin_absolu_que vous_avez_vu_avec_pwd_au_debut
```

4. Lister le contenu d'un répertoire (ls)

Tester dans le répertoire courant.

```
ls
ls .
ls -l
ls -la
```

Pour voir toutes les options de ls, faire `man ls`.

5. Créer un répertoire (mkdir)

Tester (dans votre répertoire racine (= maison))

```
mkdir DU_SR
ls
mkdir DU_SR/TP1
ls DU_SR
cd DU_SR/TP1
ls -la
```

6. Copier un fichier (cp)

Créer le fichier dans `~/DU SR/TP1/fichier.txt` (avec votre éditeur préféré).
Copier le fichier avec (si vous êtes dans `~/DU SR/TP1`)

```
cp fichier.txt copie.txt
```

Vérifier le contenu du répertoire `~/DU SR/TP1` (en ligne de commande) Vérifier le contenu de copie.txt (en l'ouvrant avec votre éditeur ou tout simplement avec cat copie.txt.

7. Déplacer un fichier (mv)

Essayer

```
mv copie.txt ..
```

Regarder le contenu du répertoire courant et du répertoire père.

8. Effacer un fichier (rm)

Essayer et vérifier les effets de la commande suivante

```
rm ~/DU SR/copie.txt
```

9. Effacer un répertoire (rmdir)

`rmdir` ne marche que si le répertoire est vide. Pour effacer un répertoire non vide, utiliser `rm -rf`. Effacer le répertoire DU SR.