IC2

TP 1 Fils d'exécution

CR à rédiger : un texte en format `.txt` avec les réponses aux questions. La réponse de toute question ne doit pas dépasser 1-2 lignes (sur une feuille A4, taile de police : 12, marges classiques). À la fin des 3 TPs, vous imprimez ce fichier et il sera noté.

1 Les signaux et les processus

Exercice 1.1 Utiliser la commande `ps -e` dans un terminal pour visualiser tous les processus qui tournent sur le système d'exploitation. Ensuite, utiliser l'opérateur `pipe` (|) et la commande `grep` *MOTIF* afin de lister uniquement les processus avec un nom qui contient le mots *MOTIF* (ex., essayer *MOTIF*="gnome"). Utiliser encore une fois le `pipe` et la commande `cut` (voir `man cut`) afin d'afficher uniquement les PIDs (process IDs) des ces processus. Pour compter le nombre de processus afficher, vous pouvez utiliser `|wc -l`.

Question 1 Quelle information indique la troisième colonne de la commande `ps -ea` ? Décrire le résultat de la commande `ps f -ea`. □

Notions de base sur les signaux Les signaux sont des événements externes qui changent le déroulement d'un programme de manière asynchrone, c'est-à-dire à n'importe quel instant lors de l'exécution du programme. Les signaux s'opposent aux autres formes de communications où les programmes doivent explicitement demander à recevoir les messages externes en attente, par exemple en utilisant les sockets (qui nécessitent la construction d'un serveur).

Il y a plusieurs types de signaux, indiquant chacun une condition particulière. En voici quelques-uns, avec le comportement par défaut associé entre parenthèses : `sigint` envoyé par `CTRL-C` (terminaison), `sigterm` envoyé par la commande `kill` ou par `shutdown now` (terminaison), `SIGUSR1` et `SIGUSR1` (sans action par défaut, il s'agit de signaux définis par l'utilisateur), `SIGSEGV`, ou *segmentation fault*, envoyé par le noyaux du système d'exploitation lorsqu'il y a un problème d'accès à la mémoire (terminaison).

Voici un simple exemple de *handler* qui permet de traiter un signal en Ruby.

```
1 Signal.trap("TERM") do
2     puts "Ce signal ne peux pas me tuer!"
3 end
```

Exercice 1.2 Ajouter une ligne au programme ci-dessus pour exécuter un `sleep` de 100 secondes. Dans une console, tapez la commande `ps f` (à étudier) pour trouver le PID du processus `ruby`. Ensuite, taper la commande `kill` PID afin d'essayer de l'arrêter. Si tout fonctionne correctement, vous n'allez pas réussir à arrêter le processus de cette manière, car la commande `kill` envoie en fait le signal `SIGTERM`.

Question 2 Donner une autre commande `kill` qui permet d'arrêter le processus `ruby` sans lui laisser d'alternative. □

Exercice 1.3 Ajouter au programme précédent un *handler* pour traiter le signal `SIGINT`. Vérifier le fait qu'il devient impossible d'arrêter le programme avec `CTRL-C`.

Exercice 1.4 Soit le programme `jruby` ci-dessous. Ajouter un traitement de signal afin de pouvoir inverser la direction de rotation de manière asynchrone en utilisant le signal `SIGUSR1`. Dans un autre terminal, utiliser la commande `kill` pour faire changer la direction de rotation.

```
1 require "java"
2 import javax.swing.JFrame;
3 $fenetre = JFrame.new("Bonjour");
4 $fenetre.setSize(200,200)
```

```
$fenetre.setVisible(true)
$fenetre.setDefaultCloseOperation(JFrame::EXIT_ON_CLOSE)
i=1
while (true) do
    $fenetre.setLocation(100 + 50*Math.sin(i*Math::PI/180),100 + 50*Math.cos(i*
        Math::PI/180))
    sleep(0.02)
    i=i+1
end
```

Faites valider le programme par l'enseignant

Exercice 1.5 L'instruction `Process.kill("TERM",12345)` permet d'envoyer le signal `TERM` au processus de PID `12345`. L'instruction `a=‘ps‘` permet de récupérer dans la variable `a` le résultat texte de la commande `ps`. **Attention :** il faut bien utiliser le caractère ‘ (disponible via `<ALT>-<7>`) et **non pas l'apostrophe classique** ’. Écrire un programme `ruby` qui permet de tuer tous les processus dont le nom contient le motif `sleep`. En bref, ce programme devrait pouvoir tuer tous les processus lancés par des commandes comme `sleep 3`. `Indication` La liste des processus qui contient un certain motif peut être récupérée en lançant une commande externe de type `ps ... | grep ... | cut ...` comme dans un des exercices ci-dessus.

2 Manipulations de matrices avec des fils d'exécutions

Un des principaux objectifs des fils d'exécutions est la programmation parallèle. Les systèmes d'exploitations modernes peuvent gérer un grand nombre de processus parallèles. Les cartes graphiques utilisent couramment des centaines de processeurs pour faire des calculs sur des matrices.

Soit le code ci-dessous.

```
$n = 10
$a = Array.new($n)
$a2 = Array.new($n)
for i in 0..($n-1) do
    $a[i]  = Array.new($n)
    $a2[i] = Array.new($n)
    for j in 0..($n-1) do
        $a[i][j] = i+j
        $a2[i][j] = 0
    end
end

def calc(i,j)
    $a2[i][j] = 0
    for k in 0..$n-1 do
        $a2[i][j] = $a2[i][j]+$a[i][k]*$a[k][j]
        sleep(0.01)    #multiplication=operation lente
    end
end
def calcLigne(i)
    for j in 0..$n-1 do
       calc(i,j)
```

```
23      end
24 end
25 for i in 0..$n-1 do
26      calcLigne(i)
27 end
```

Question 3 Quel est l'objectif du code ci-dessus, donner une formule qui exprime $A2$ en fonction de A. □

Exercice 2.1 Ajouter une routine qui permet d'afficher $A2$. Afficher ainsi la matrice $A2$, ainsi que le temps de calcul. (utiliser `Time.now` avant et après l'opération).

Question 4 Indiquer le temps de calcul pour $n = 10, 15, 20, 25$ et $n = 30$. □

Après chaque multiplication, notre programme exécute `sleep(0.01)`. Cela simule le fait que la multiplication est une opération beaucoup plus couteuse que l'addition. En `Ruby`, l'instruction `sleep` est (presque) la seule instruction qui peut être réellement exécutée en parallèle par plusieurs fils d'exécution en même temps. Concernant les autres instructions, `Ruby` transforme le programme en un programme séquentiel (le processeur ne dépassera jamais une utilisation de `100`%).

Exercice 2.2 Ajouter un fil d'exécution qui permet de calculer les premières $n/2$ lignes de la matrice $A2$. Le programme principale pourrait calculer les autres lignes (de $n/2 + 1$ à $n - 1$). Afficher la matrice $A2$, ainsi que le temps de calcul.

Après avoir construit le fil d'exécution (e.g., avec `fil = Thread.new do...`), le programme principal doit exécuter `fil.join()` pour attendre la terminaison de ce fil.

Question 5 Indiquer le temps de calcul pour $n = 10, 15, 20, 25$ et $n = 30$. □

Exercice 2.3 Ajouter un fil d'exécution qui permet de calculer chaque élément de la matrice $A2$ d'une manière indépendante.

Question 6 Indiquer le temps de calcul pour $n = 10, 15, 20, 25$ et $n = 30$. □

3 Serveur TCP *multi*-utilisateur à base de fils d'exécutions

Les serveurs à base de sockets qu'on a réalisé dans le passe ne permettaient la connexion de plusieurs clients.

Exercice 3.1 Réaliser un serveur TCP qui permet la connexion d'un nombre indéfini de clients. Tout client qui se connecte peut envoyer une chaîne de caractères et recevoir une réponse. En fait, le serveur devrait répondre avec la même chaine de caractères et fermer la connexion.

Pour capter des éventuelles erreurs dans les fils d'exécutions, lancez les programmes avec `ruby -d`!!!

Exercice 3.2 Modifier le programme précédent afin de faire le serveur construire un tableau de de sessions TCP. Après toute connexion d'un chaque client (`sessionLocale=serv.accept`), le tableau de sessions devrait ajouter la nouvelle session avec une instruction comme `$sessions<<sessionLocale`. Chaque message reçu d'un client devrait être retransmis à tous les autres clients (à toutes les sessions du tableau `$sessions`). L'objectif est de faire une messagerie multi-utilisateur. Chaque message envoyé par le serveur devrait être précédé par `IP:PORT-->` ou IP et PORT représentent l'ip et le port du client qui a envoyé le message. Pour récupérer ces informations, utilisez, par exemple, `addrtype, port, nom, ip = sessionLocale.peeraddr()`..

Faites valider le programme par l'enseignant

Comment peut-on limiter le nombre de fils d'exécutions qui tournent en même temps ? Une solution serait d'utiliser au début de chaque fil d'exécution un code comme, par exemple,

```
1 Thread.new do
2     $filsDispo = $filsDispo -1
3     while($filsDispo<=0) do
4         #ne rien faire, attendre
5     end
6 end
```

Question 7 Donner deux inconvénients de la solution présentée ci-dessous. □

Les systèmes d'exploitation utilisent la notion de sémaphore pour limiter le nombre de fils d'exécutions qui tournent en même temps. Télécharger le fichier `sem.rb` et ajoutez au début de vos programme multi-thread l'instruction `require sem.rb`. Vous pouvez utiliser les instructions ci-dessous :

- `monSem = Semaphore.new(maxNoFils)` : le sémaphore est initialisé avec une valeur `maxNoFils`
- `monSem.down()` : si la valeur du sémaphore est supérieure à 0, alors celle ci est décrémentée et le fil d'exécution continue de tourner. Si la valeur du sémaphore est 0, alors cette instruction bloque le fil d'exécution en attente jusqu'au moment où la valeur est incrémenté par un autre fil
- `monSem.up()` : à la fin d'un fil d'exécution, on incrémente la valeur du sémaphore. Ainsi, un autre fil d'exécution bloqué sur `monSem.down()` pourrait commencer son activité.

Exercice 3.3 Modifier le programme précédent afin de permettre la connexion de maximum 3 client en même temps. Une éventuelle 4ème connexion devrait attendre jusqu'à ce qu'un autre client se déconnecte.

4 Console à distance et transfert simplifié de fichiers texte

Exercice 4.1 Écrire un serveur multi-thread qui exécute les commandes `BASH` reçues sur le port 2222 et renvoie le résultat. Écrire un client qui permet d'envoyer au serveur les commandes tapées au clavier par l'utilisateur. Le client ferme la connexion si l'utilisateur tape `stop`. Le serveur doit pouvoir gérer au maximum 3 clients.

Indications :

- Pour exécuter une commande `BASH` et récupérer sa sortie standard (habituellement affiché à l'écran), le serveur pourrait utiliser une instruction comme `sortie=‘#{ligne}‘`.
- L'instruction `‘#{ligne}‘` peut générer plusieurs lignes de sortie. Par exemple, si **ligne="ls -l"**, alors la sortie contient plusieurs lignes. Ainsi, pour indiquer la fin de la sortie, le serveur devrait envoyer un mot clé "END" supplémentaire. Le client arrête la réception de la sortie uniquement lorsqu'il reçoit ce mot clé (sans l'afficher).

Exercice 4.2 Ajouter la commande `GET` au programme client. Plus exactement, si l'utilisateur tape `GET fic.txt`, le client devrait ouvrir (pour l'écriture) le fichier `fic.txt` et fonctionner d'une manière spéciale, *i.e.,* en mode GET. En effet, le client devrait en fait envoyer au serveur la commande `cat fic.txt`. Le serveur n'est pas modifié et il répond avec le contenu du fichier indiqué. Cependant, dans ce cas, le client est en mode `GET` et ne devrait pas afficher ce contenu, mais juste l'écrire dans le fichier `fic.txt`.

5 Simulation des flux avec fils d'exécutions et sémaphores

Un passager à l'aéroport doit passer par plusieurs points : achat de billet sur place (optionnel), enregistrement et contrôle de sécurité. On suppose que l'intervalle de temps entre deux clients est un numéro aléatoire entre 0.1 et 0.5 secondes. Il existe un nombre a de guichets pour acheter le billets (probabilité de devoir acheter un billet :

50%), b guichets d'embarquement (durée aléatoire entre a_1 et a_2 secondes) et c points de sécurité. Trouver le point qui génère des bouchons.

6 Question Bonus

Exercice 6.1 Utiliser la commande `ulimit -v 100000` pour limiter la mémoire virtuelle totale disponible pour les commandes lancées dans le même terminal. Lancer un serveur web Ruby en utilisant le programme présenté en cours, par exemple. Utiliser ensuite une autre machine pour envoyer des requêtes à ce serveur. Lancer un programme Ruby client qui envoie des requêtes de grande taille afin de faire planter le serveur.

Deuxième étape : installer Apache sur la machine serveur et essayer de répéter l'opération.