Pratique des systèmes — L3 MIAGE

Première session

décembre 2017

1 Quizz

Exercice 1 – *Quizz*

1. Plutôt que d'être monolithique, certains systèmes d'exploitations utilisent une architecture client-serveur. De quoi s'agit-il ? Citez un avantage et un désavantage.
2. On suppose qu'il n'y a pas d'erreur lors de l'exécution du code suivant :

```
#include <unistd.h>                                    1
#include <sys/wait.h>                                  2
int main(void) {                                       3
    write(1, "A", sizeof "A") ;                        4
    int child = fork();                                5
    write(1, "B", sizeof "B") ;                        6
    if (child)                                         7
       wait(NULL);                                     8
    write(1, "C", sizeof "C") ;                        9
    return 0 ;                                         10
}                                                      11
```

Donnez tous les affichages possibles produits par ce code.

2 Écriture de code

Exercice 2 – *Processus léger*

La suite de Syracuse $(u_i)_{i\in\mathbb{N}}$ d'un nombre entier n strictement positif est définie par récurrence de la manière suivante :

$$u_0 = n, \quad u_{i+1} = \begin{cases} \frac{u_i}{2} \text{ si } n \text{ est pair;} \\ 3u_i + 1 \text{ si } n \text{ est impair.} \end{cases}$$

Par exemple, pour les 5 premiers entiers, on obtient les suites :

1.
2, 1.
3, 10, 5, 16, 8, 4, 2, 1.
4, 2, 1.
5, 16, 8, 4, 2, 1.

Si le nombre 1 est atteint, on arrête la suite car elle devient cyclique 4, 2, 1, 4, 2, 1, etc.

La conjecture de Syracuse est l'hypothèse selon laquelle la suite de Syracuse de n'importe quel entier strictement positif atteint 1.

Cette conjecture popularisée par Lothar Collatz aux environs de 1937 reste l'un des plus célébre mystère mathématique dont l'énoncé est simple.

Pour chaque entier, on appelle :

— vol de Syracuse de cet entier, la suite u_i produite ;
— l'altitude maximale du vol est la valeur maximale du vol.

Ainsi, pour le vol de 3, l'altitude maximale est 16.

Remarquons que si au cours d'un vol, on atteint une altitude maximale déjà atteinte par un autre vol se terminant par 1, il n'est pas nécessaire de continuer puisque le vol en cours se terminera aussi par 1.

Cette observation peut inspirer le code suivant pour effectuer des vols :

```
int main(void){                                                                    1
    graine:=6:                                                                     2
    globalaltitudemax:=16:                                                         3
    while(1==1){                                                                   4
        position=graine ; /* d\'ebut du vol */                                     5
        localaltitudemax=graine ;                                                  6
        while(1==1){                                                               7
          if(position % 2 ==0) position = position/2;                              8
          else position = 3*position+1 ;                                           9
          if(position>localaltitudemax) localaltitudemax = position ;              10
          if(position== globalaltitudemax) break ; /* on interrompt le vol */     11
          if(position==1) break ;                                                  12
        }                                                                          13
        graine++;                                                                  14
        if(globalaltitudemax<localaltitutedmax) globalaltitudemax=localaltitudemax ; 15
        /* fin du vol */                                                           16
    }                                                                              17
    return 0 ;                                                                     18
}                                                                                  19
```

Questions. Ce code ne tire pas partie d'une architecture multicore car une seule unité de calcul est sollicitée pour calculer l'ensemble des vols. Nous allons adapter ce code en utilisant 8 processus légers pour effectuer ces vols.

1. Quelles sont les ressources partagées et les sections critiques à prendre en compte ?
2. Combien de sémaphores allez vous utiliser pour gérer les accès concurrents ? De quel type sont ces sémaphores ?
3. Modifiez ce code afin de le paralléliser et de faire 8 vols simultanés.

Indications culturelle : comme tous les entiers plus petit que $5 \cdot 2^{60}$ on déjà été vérifiés, il faut être plus malin que l'implantation ci-dessus pour apprendre quoi que ce soit sur la conjecture de Syracuse.

3 Synchronisation

Exercice 3 – *Sémaphore*

Une piscine dispose de 10 cabines, de 200 casiers. Il ne peut y avoir que 150 baigneurs dans le bassin de la piscine.

Chaque personne voulant accéder à la piscine doit dérouler le scénario suivant :
— prendre un panier dans un casier,
— aller dans une cabine,
— se déshabiller,
— libérer la cabine,
— se baigner,
— retrouver une cabine,
— s'habiller,

— libérer sa cabine et
— remettre son panier dans un casier.

Le programme des processus `baigneur` utilisera les fonctions `seDeshabille()`, `seBaigne()` et `seRhabille()` :

```
programme baigneur;
  seDeshabille();
  seBaigne();
  seRhabille();
```

3.1 Synchronisation

Utilisez des sémaphores pour synchroniser les processus baigneur afin qu'ils respectent le scénario décrit ci-dessus.

3.2 Nouveaux intervenants

De nouveaux acteurs sont introduits dans notre application : le personnel de nettoyage qui s'occupe de bassin en cas de problème d'hygiène. Remarquez qu'eux aussi passe par les cabines pour se changer.

```
programme nettoyage;
   seDeshabille();
   nettoye_si_necessaire();
   seRhabille();
```

Pour que tout fonctionne au mieux, les baigneurs ne peuvent occuper le bassin pendant le nettoyage et ce dernier ne peut pas commencer tant qu'il y a des baigneurs dans le bassin.

Modifiez les programmes `baigneur` et `nettoyage` en conséquence.

Indications

— L'appel système de prototype :

```
#include <fcntl.h> /* Pour les constantes O_* */
#include <sys/stat.h> /* Pour les constantes des modes */
#include <semaphore.h>
  sem_t *sem_open( const char *name, int oflag,
                   mode_t mode, unsigned int value);
```

permet de créer un sémaphore nommé et de l'initialiser à la valeur `value`.

— L'appel système de prototype :

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

permet de verrouiller un sémaphore.

— L'appel système de prototype :

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

permet de déverrouiller un sémaphore.

— L'appel système de prototype :

```
#include <semaphore.h>
int sem_close( sem_t *) ;
```

permet de se dissocier d'un sémaphore nommé.

— L'appel système de prototype :

```
#include <semaphore.h>
int sem_close( const char *) ;
```

permet de détruire un sémaphore nommé.

— L'appel système `pthread_create` dont le prototype est :

```
#include <pthread.h>                                                                    1
int pthread_create(pthread_t *, pthread_attr_t *, void * (*start_routine)(void *), void *arg);   2
```

`pthread_create` crée un nouveau thread s'exécutant simultanément avec le thread appelant. Le nouveau thread exécute la fonction `start_routine` en lui passant arg comme premier argument. Le nouveau thread s'achève soit explicitement en appelant `pthread_exit`, ou implicitement lorsque la fonction `start_routine` s'achève. Ce dernier cas est équivalent à appeler `pthread_exit` avec la valeur renvoyée par `start_routine` comme code de sortie.
Le second indique les attributs du nouveau thread. L'argument `attr` peut être NULL, auquel cas, les attributs par défaut sont utilisés : le thread créé est joignable (non détaché) et utilise la politique d'ordonnancement normale (pas temps-réel).