IG2I
Institut de Génie Informatique et Industriel

LA1 - Année 2016-2017

TP n°2 - Processus, threads, parallélisme et concurrence

Ce second TP aborde l'utilisation de threads pour faciliter la mise en place du parallélisme dans le développement d'applications. Il se termine en exhibant les problèmes de concurrence induits par ce parallélisme.

Ce TP fera l'objet d'un compte-rendu contenant :

- les réponses aux diverses questions présentes dans cet énoncé,
- les codes sources commentés des programmes réalisés,
- les jeux d'essais obtenus,
- une conclusion sur le travail réalisé, les difficultés rencontrées, les connaissances acquises ou restant à acquérir, …

Il sera tenu compte de la qualité de vos pratiques de production de code :

- Lisibilité du code : emploi de commentaires, indentation, constantes symboliques
- Automatisation : Fichiers makefile avec macros …
- Robustesse : Vérification des valeurs de retours des appels systèmes

IG2I
Institut de Génie Informatique et Industriel

LA1 - Année 2016-2017

1 Processus/threads

Vous allez dans ce paragraphe comparer les performances de deux applications effectuant la même tâche : l'une nommée `test_fork` fait appel à des processus fils, l'autre `test_thread` fait appel à des threads.

Le code source de `test_fork` est présenté en annexe : le processus père crée 50000 processus fils qui ne réalisent, chacun, qu'une opération élémentaire avant de se terminer.

Dans l'application `test_thread` à réaliser, le thread principal sera chargé de créer 50000 threads enfants qui n'effectueront, chacun, qu'une opération élémentaire (la même que celle de l'application `test_fork`) avant de se terminer.

1. Générer l'application `test_fork` à partir du code source fourni et utiliser la commande `time` pour déterminer sa durée d'exécution.
2. Rappeler la définition et les principales caractéristiques des threads ainsi que le rôle des trois appels systèmes permettant de mettre en place un programme manipulant des threads :
   - `pthread_create()`
   - `pthread_exit()`
   - `pthread_join()`

   Avec quelle(s) librairie(s) un programme manipulant des threads doit-il être lié ?
3. En utilisant les appels systèmes précédents, développer l'application `test_thread` d'une manière similaire à `test_fork` en remarquant que :
   - `fork()<=>  pthread_create()`
   - `exit()<=>  pthread_exit()`
   - `waitpid()  <=>  pthread_join()`
4. Tester cette application, comparer sa durée d'exécution à celle de `test_fork` et conclure.

2 Parallélisme

1. Expliquer le mécanisme de passage de paramètre à un thread. Donner un exemple pour le passage d'une valeur entière.
2. Rédiger un programme permettant de lancer *n* threads parallèles ; chaque thread affiche son numéro toutes les 100 ms, et compte le nombre d'affichages produits.

Mettre en évidence la problématique de l'équité en affichant au fur et à mesure les statistiques d'utilisation de chaque thread.

3. Afficher l'arborescence de processus. Que remarquez-vous ? Comment sont ordonnancés les différents threads d'un même processus ?

3 Concurrence

1 Un exemple de « scénario »

Deux applications tout à fait indépendantes, ont besoin à la fin de leur exécution, d'imprimer leurs résultats ; l'objectif étant que, dès que l'impression des résultats d'une application commence, celle-ci va jusqu'au bout…

2 Mise en évidence d'un problème

Dans l'exercice qui suit, chaque application sera traitée sous forme de *thread*, et l'impression de résultats se résumera à l'affichage d'une chaîne de caractères.

1. Développer un exemple de programme comportant deux threads affichant caractère par caractère le contenu d'une phrase donnée ; l'un affiche les caractères en majuscules, l'autre en minuscules. Exécutez à plusieurs reprises cette application. Que remarquez-vous ?
2. Vous allez maintenant accentuer le phénomène observé en insérant après l'affichage d'un caractère, une pause d'une durée aléatoire comprise entre 0 et 1 seconde (cf. `nanosleep()`) pour le thread qui affiche le texte en majuscules, et d'une durée aléatoire comprise entre 0 et 2 secondes pour l'autre thread.

3 Mise en place d'un mécanisme de protection

Il s'agit maintenant de mettre en place un mécanisme de protection à l'aide d'une ***variable globale*** dénotant la disponibilité de la ressource écran : l'objectif étant de garantir que, dès qu'un thread commence son affichage, il puisse le terminer avant que l'autre thread ne commence le sien. L'un des threads va donc attendre, dans une boucle, tant que la ressource est indisponible.

1. Comment s'appelle ce mécanisme d'attente et pourquoi faut-il l'éviter ?
2. Est-ce que cela permet de résoudre parfaitement le problème identifié à la question précédemment ? Donner un scénario expliquant la cause du problème.

Le système Linux propose un mécanisme permettant de résoudre ces deux problèmes, en utilisant des primitives *atomiques* (ie non interruptibles) de manipulation de variables entières : les ***sémaphores***. Ce sera l'objet du TP3.

4 Annexe

```c
/* ------------------------------------------------------------------- */
/*    test_fork.c                                                      */
/*    Test de création de processus à l'aide de l'appel système fork()  */
/* ------------------------------------------------------------------- */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define NB_FORKS 50000

void do_nothing()
{
    int i;

    i = 0;
    exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
    int pid, j, status;

    for (j = 0; j < NB_FORKS; j++)
    {
        switch (pid = fork())
        {
        case -1 :
            perror ("fork()");
            exit(EXIT_FAILURE);

        case  0 :       /** le code du processus fils est ici */
            do_nothing();

        default:  /** Suite du processus père */
            waitpid(pid, &status,0);
            break;
        }
    }
    return EXIT_SUCCESS;
}
```