Matière : PRS (Programmation Système)
Année : LA1
Année scolaire : 2019-2020
Enseignant : Isabelle Le Glaz

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 mettant en évidence 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

1 Processus/threads

Vous allez comparer les performances de deux applications effectuant la même tâche :

- L'une nommée **test_fork** fait appel à des processus fils, (comme vu dans le TP N°1)
- L'autre **test_thread** fait appel à des threads.

Le code source de **test_fork** est joint 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. Ecrire le code du programme **test_thread2_1.c** réalisant le passage d'une valeur entière depuis le thread principal (main) vers le thread secondaire créé
2. Rédiger un programme **test_thread2_2.c** permettant de lancer ***n*** threads parallèles (n étant défini par un argument passé en ligne de commande) ; chaque thread affiche son numéro de thread (n) toutes les 100 ms, et incrémente un compteur.
3. Mettre en évidence la problématique de l'équité en affichant au fur et à mesure les statistiques d'utilisation de chaque thread.
4. Afficher l'arborescence de processus. Que remarquez-vous ? Comment sont ordonnancés les différents threads d'un même processus ?

3 Concurrence

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...

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 **test_thread3_1.c** 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 (programme **test_thread3_2.c)** 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.

4 Mise en place d'un mécanisme de protection

L'objectif de cette question est de mettre en évidence qu'un ensemble d'opérations dans un thread peuvent être interrompues librement par le système.

Si on souhaite qu'une opération ou un ensemble d'opérations ne soi(en)t pas interrompue(s), on peut choisir de protéger ce traitement en utilisant une variable globale permettant de signaler que le traitement est entré dans une phase à ne pas interrompre. Tant que ce « drapeau » est levé, aucun autre traitement ne soit être lancé.

1. Comment s'appelle ce mécanisme d'attente et pourquoi faut-il l'éviter ?
2. Ecrire un programme **test_caractere.c** permettant de réaliser cette protection.

Pour cela, vous allez écrire un programme qui :

- ✔ Exécute deux threads, exécutant chacun une fonction différente (fonctionA et fonctionB).
  - La fonctionA
    - affiche, caractère par caractère, le contenu d'une chaine de caractères de longueur 20 et ne contenant que des A.
    - Attend entre chaque affichage de caractère en utilisant deux boucles imbriquées suffisamment longues pour y passer du temps (ne pas utiliser sleep)
  - La fonctionB
    - affiche, caractère par caractère, le contenu d'une chaîne de caractères de longueur 20 et ne contenant que des B.
    - Attend entre chaque affichage de caractère en utilisant deux boucles imbriquées suffisamment longues pour y passer du temps (ne pas utiliser sleep)
- ✔ Attend la fin des deux threads pour se terminer

Que constatez vous ?

3. Modifier votre programme pour introduire une variable globale qui permet de vous assurer que vous aurez écrit tous les caractères d'une des chaînes de caractères avant de passer à l'affichage de l'autre chaîne.
4. Quel est l'inconvénient de cette solution ?

5 Les mutex

Lorsqu'un nouveau processus est créé par un fork, toutes les données (variables globales, variables locales, mémoire allouée dynamiquement), sont dupliquées et copiées, et le processus père et le processus fils travaillent ensuite sur des variables différentes.
Dans le cas de threads, la mémoire est partagée, c'est à dire que les variables globales sont partagées entre les différents threads qui s'exécutent en parallèle. Cela pose des problèmes lorsque deux threads différents essaient d'écrire et de lire une même donnée.

Deux types de problèmes peuvent se poser :

• Deux threads concurrents essaient en même temps de modifier une variable globale ;
• Un thread modifie une structure de donnée tandis qu'un autre thread essaie de la lire.

Il est alors possible que le thread lecteur lise la structure alors que le thread écrivain a écrit la donnée à moitié. La donnée est alors incohérente.

Pour accéder à des données globales, il faut donc avoir recours à un mécanisme d'exclusion-mutuelle, qui fait que les threads ne peuvent pas accéder en même temps à une donnée. Pour cela, on introduit des données appelés mutex, de type pthread_mutex_t.

Un thread peut verrouiller un mutex, avec la fonction pthread_mutex_lock(), pour pouvoir accéder à une donnée globale ou à un flot (par exemple pour écrire sur la sortie stdout). Une fois l'accès terminé, le thread déverrouille le mutex, avec la fonction pthread_mutex_unlock().

Si un thread A essaie de verrouiller un mutex alors qu'il est déjà verrouillé par un autre thread B, le thread A reste bloqué sur l'appel de pthread_mutex_lock() jusqu'à ce que le thread B déverrouille le mutex.
Une fois le mutex déverrouillé par B, le thread A verrouille immédiatement le mutex et son exécution se poursuit. Cela permet au thread B d'accéder tranquillement à des variables globales pendant que le thread A attend pour accéder aux mêmes variables.
Pour déclarer et initialiser un mutex, on le déclare en variable globale (pour qu'il soit accessible à tous les threads) :

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER ;
```

La fonction pthread_mutex_lock(), qui permet de verrouiller un mutex, a pour prototype :

```
int pthread_mutex_lock ( pthread_mutex_t *mutex ) ;
```

NB : Il faut éviter de verrouiller deux fois un même mutex dans le même thread sans le déverrouiller entre temps. Il y a un risque de blocage définitif du thread.
Certaines versions du système gèrent ce problème mais leur comportement n'est pas portable.

La fonction pthread_mutex_unlock(), qui permet de déverrouiller un mutex, a pour prototype:

```
int pthread_mutex_unlock ( pthread_mutex_t *mutex ) ;
```

1 : Ecrire un programme **test_mutex1.c** qui permet de créer 10 threads de la même fonction.

La fonction de thread réalisera un calcul peu complexe, basé sur un nombre aléatoire, une opération simple et des boucles imbriquées pour prendre du temps.
La fonction doit écrire sur la sortie standard son N° de thread (de 1 à 10), réaliser le calcul, puis écrire sur la sortie standard quand le calcul est terminé. Son calcul ne doit pas être interrompu par les autres threads.

2 : Comment pouvez-vous vérifier facilement que le calcul d'un thread n'a pas été interrompu ?

3 : Que se passe-t-il si vous commentez les parties de votre code relatives aux mutex ?

Lorsqu'un thread doit patienter jusqu'à ce qu'un événement survienne dans un autre thread, on emploie une technique appelée la condition.
Quand un thread est en attente d'une condition, il reste bloqué tant que celle-ci n'est pas réalisée par un autre thread.
Comme avec les mutex, on déclare la condition en variable globale, de cette manière :

```
pthread_cond_t nomCondition = PTHREAD_COND_INITIALIZER;
```

Pour attendre une condition, il faut utiliser un mutex :

```
int pthread_cond_wait(pthread_cond_t *nomCondition, pthread_mutex_t *nomMutex);
```

Pour réveiller un thread en attente d'une condition, on utilise la fonction :

```
int pthread_cond_signal(pthread_cond_t *nomCondition);
```

4: Créez un programme (test_mutex2.c) qui crée deux threads : un qui incrémente une variable compteur par un nombre tiré au hasard entre 0 et 10, et l'autre qui affiche un message lorsque la variable compteur dépasse 20.

5 : Reprenez le programme de la question 1 (test_mutex3.c).
Ajouter un thread d'affichage qui se réveille quand un thread a terminé toutes ses itérations.

6 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;
}
```