Introduction

Ce TP a pour but de vous servir de prise en main pour le développement de programmes parallèles multithreadés en C.

Rendus attendus

Il est attendu que vous preniez des notes pour répondre aux questions posées, et que vous interagissiez avec votre encadrant·e de TP en cas de doute.

Votre prise de note et les codes que vous produirez pendant ce TP ne seront pas évalués directement sur cette séance. Il est cependant attendu que vous réalisiez complètement les **3 exercices** de ce TP. Les évaluations suivantes (prochain TP et contrôle sur table) supposera que les compétences manipulées pendant ce TP sont maîtrisées.

Prérequis matériels, logiciels et de données

- Un CPU en `x86-64` (Intel/AMD 64 bits) ou en `aarch64` (ARMv8).
  Si besoin, `uname -a` affiche l'architecture matérielle de votre CPU.
- Un Linux, un terminal, un shell.
- Une chaîne de compilation C : `gcc`, `clang`…
- Ou outil qui permet de tracer les appels système : `strace`, `lurk`…
- Avoir téléchargé les fichiers du TP (ce sujet PDF + codes source).

Ressources supplémentaires

- Supports de cours / TD.
- Pages de manuel des différentes fonctions à manipuler.

Compétences

Attendues (prérequis)

- Programmation en C.
- Compilation et exécution de commandes dans un shell.

Travaillées pendant le TP

- Tracer les appels système réalisés par un programme monothread ou multithread.
- Concevoir des applications multithread simples (parallélisme de données).
- Implémenter une exclusion mutuelle simple.
- Mesurer et analyser la performance de différentes implémentations du même programme.

Exercice 1 : Éxécution de programmes et appels système

1. Compilez et exécutez chacun des programmes suivants pour observer leur comportement.
   Ne compilez les exemples en assembleur que pour l'architecture de votre CPU.

Fichier source	Instruction de compilation
`hello-x64.s`	`gcc -nostartfiles -nostdlib -no-pie -o hello-x64 hello-x64.s`
`hello-aarch64.s`	`gcc -nostartfiles -nostdlib -no-pie -o hello-aarch64 hello-aarch64.s`
`hello.c`	`gcc -o hello-c hello.c`
`fork.c`	`gcc -o fork fork.c`

`thread-sem-testbed.c`	`man pthread_create` pour voir comment activer `pthread` selon votre `libc`…
	`gcc -lpthread -o thread-sem-testbed thread-sem-testbed.c`
	`gcc -pthread -o thread-sem-testbed thread-sem-testbed.c`

2. Exécutez les versions assembleur et C du programme *hello world* via `strace`.
   1. Quelles sont les différences d'appels système réalisés par ces deux versions ?
   2. D'où viennent ces différences ?

3. Complétez le tableau suivant en déterminant quels appels système sont réalisés dans chacun des différents cas, en appelant les différents programmes fournis.
   - Différents arguments d'appels devront être donnés à `thread-sem-testbed`.
   - Quand il y a plusieurs threads, jetez un œil aux options `-f`, `-ff` et `--output` de `strace`.
   - L'option `-e` de `strace` permet de paramétrer quels appels système doivent être tracés ou non. Un script `strace-wrap.sh` est fourni avec les fichiers du TP et permet de *wrapper* `strace` de telle sorte que la plupart des des appels systèmes inutiles pour cet exercice soient cachés. Il est recommandé d'utiliser ce script plutôt que `strace` directement.

Cas	Appel système
Terminaison du thread principal	
Appel de `sleep()`	
Appel de `fork()`	
Appel de `wait(NULL)`	
Appel de `pthread_create()`	
Appel bloquant à `pthread_join()`	
Appel non bloquant à `pthread_join()`	
Appel bloquant à `sem_wait()`	
Appel non bloquant à `sem_wait()`	
Appel réveillant à `sem_post()`	
Appel non réveillant à `sem_post()`	

Exercice 2 : Programmation multithread

Le programme `mat-gen-sum.c` fait les opérations suivantes.

- Génération d'une matrice carrée $n * n$ de nombres entiers à valeur pseudo-aléatoire entre 0 et 10.
- Calcul de la somme des valeurs de la matrice.
- Affichage de la somme des valeurs de la matrice.

1. Compilez le programme fourni.
2. Executez le programme et vérifiez son déterminisme avec l'entrée `0 10 1`
3. Imaginons que vous vouliez paralléliser le calcul de la somme de la matrice, de telle sorte que chaque thread calcul la somme d'une ligne.
   1. Les données de chaque thread sont-elles indépendantes ?
   2. En déduire le besoin de synchronisation entre vos threads.
4. Faites une version multithread de ce programme dans le fichier `mat-gen-sum-mt-v1.c`
   - Parallélisez **uniquement** le calcul de la somme de la matrice.
   - En supposant que le nombre de threads vaut n, attribuez le calcul de la ligne i au thread i.
   - Vérifiez que votre code renvoie le même résultat que la version séquentielle.
5. Améliorez la version précédente dans un fichier `mat-gen-sum-mt-v2.c`
   On suppose maintenant que n est un **multiple** du nombre de threads.
   - Chaque thread reçoit en entrée deux entiers i et j
   - Chaque thread calcule la somme des lignes x telles que $i \leq x < j$
   - Le `main` distribue les lignes de manière équilibrée aux threads.
6. Améliorez la version précédente dans un fichier `mat-gen-sum-mt-v3.c`
   On veut maintenant aussi paralléliser la génération de la matrice.
   1. Quelle dépendance y a-t-il entre les données de la matrice dans le code initial ?
   2. Proposez une méthode déterministe quel que soit le nombre de threads pour générer pseudo-aléatoirement le contenu de la matrice.
      Faites valider votre méthode par votre encadrant·e de TP.
   3. Implémentez votre méthode en utilisant un `rand` **réentrant** comme `rand_r`.

Exercice 3 : Accès partagé non synchronisé

1. Lisez le code du fichier fourni `para-write.c`. Que fait ce programme ?
2. Le résultat de ce programme devrait-il être déterministe ? Pourquoi ?
   1. Compilez le programme puis exécutez-le plusieurs fois sur de grandes valeurs de paramètres. Faites varier ces valeurs jusqu'à voir régulièrement un problème de résultat.
   2. Mesurez le temps moyen d'exécution de votre programme, par exemple en le lançant via `time`
3. Implémentez différentes versions synchronisées de ce programme dans des fichiers différents. Vérifiez que vos implémentations sont déterministes en les lançant quelques fois, et mesurez le temps moyen d'exécution de chaque programme.
   1. Dans `para-write-sync-v1.c`
      - **Chaque incrémentation** de la variable partagée doit être réalisée en **exclusion mutuelle**.
      - Vous devez utiliser un `phtread_mutex_t` pour réaliser la synchronisation.
   2. Dans `para-write-sync-v2.c`
      - **Chaque incrémentation** de la variable partagée doit être réalisée en **exclusion mutuelle**.
      - Vous devez utiliser un `sem_t` pour réaliser la synchronisation.
   3. Dans `para-write-sync-v3.c`
      - Chaque thread doit faire ses incrémentation sur une variable locale initialisée à 0.
      - Après tous ses calculs, chaque thread doit ajouter sa somme locale à une somme globale, qui est une variable partagée. **La mise à jour de la somme globale doit être synchronisée en exclusion mutuelle.**
4. Comparez et expliquez les différents temps d'exécution obtenus.