LINFO1252 – Systèmes informatiques

Projet 1 : Programmation multi-threadée et évaluation de performances

Au cours de ce projet vous apprendrez à évaluer et expliquer la performance d'applications utilisant plusieurs threads et des primitives de synchronisation : mutex, sémaphores (première partie) ainsi que les verrous avec attente active, que vous implémenterez vous même (deuxième partie).

Les applications considérées sont celles vues en cours et en TD : philosophes, producteurs/consommateurs avec un *buffer* de taille fixe, et lecteurs/écrivains avec la priorité aux écrivains. La métrique d'évaluation que nous considèrerons sera le temps d'exécution total, en variant le nombre de threads d'exécution, et pour un nombre d'opérations fixe (ne dépendant pas du nombre de threads).

Le projet doit être réalisé en binôme. Vous devez entrer votre binôme en utilisant l'activité correspondante sur Moodle. Si vous préférez réaliser le projet seul·e, il faut informer votre tuteur ou assistant et vous enregistrer seul·e dans un groupe. Seuls les étudiants enregistrés dans un groupe pourront rendre le projet sur Moodle.

Accès à une machine multi-cœurs

Pour effectuer vos mesures, une machine équipée de 16 coeurs sera mise à votre disposition. Il est impératif dans un premier temps de vous assurer du bon fonctionnement de vos codes/scripts sur votre propre machine/VM idéalement dotée d'au moins 2 cœurs, car le nombre de soumissions par jour sera limité.

L'accès à la machine se fait via une tâche INGInious. Lors de l'exécution de cette tâche, cette machine lui sera entièrement réservé. Ceci réduit la variance des résultats lors de mesures de performance. En contrepartie, il se peut que votre soumission passe un certain temps dans la file d'attente avant de pouvoir bénéficier de l'accès au serveur. La sortie standard de la compilation et de l'exécution du script vous sera retournée dans le feedback

Échéance et livrable

La première partie du sujet est disponible le vendredi 8 novembre 2024. La deuxième partie sera disponible le vendredi 15 novembre 2024. Le projet devra être rendu sur Moodle au plus tard le **dimanche 8 décembre 2024 à 18:00**. Il n'y aura pas d'extension.

Le livrable comprendra :

- L'ensemble du code réalisé (Python, bash et C), proprement structuré et commenté[1] ;
- Les instructions pour compiler et lancer les tests, idéalement sous la forme d'un Makefile (e.g. `make`, `make test`, `make clean`) ;
- Un rapport au format PDF d'au plus 5 pages avec les résultats des évaluations (graphiques au format PDF, avec une légende complète donnant les détails de l'expérience présentée) et avec des explications des performances observées (un paragraphe par graphique). Le rapport donnera enfin une conclusion sur les leçons apprises lors de la réalisation du projet.

[1] L'ensemble des codes sera passé en revue en utilisant des outils de détection de plagiat. En cas de plagiat, tant le groupe qui aura copié que celui qui aura partagé du code seront fermement sanctionnés.

Le projet sera noté selon les critères suivants :

- Respect des consignes de rendu (10%)
- Correction du code (30%)[2]
- Résultats et analyse dans le rapport (50%)
- Présentation du rapport (10%)

Un bonus de +5% sera donné si un Makefile correct et fonctionnel est fourni avec le code rendu.

Partie 1 : Utilisation des primitives de synchronisation POSIX

Dans cette première partie, vous mettrez en œuvre les synchronisations vues en cours et en TD et évaluerez leur performance en fonction du nombre de cœurs utilisés.

Tâche 1.1 – Coder le problème des philosophes :

- Le nombre de philosophes N est un paramètre obtenu à partir de la ligne de commande ;
- Chaque philosophe effectue **1,000,000** cycles penser/manger ;
- On n'utilise pas d'attente dans les phases manger et penser (ces actions sont immédiates) pour mettre en avant le coût des opérations de synchronisation.

Tâche 1.2 – Coder le problème des producteurs-consommateurs :

- Le nombre de threads consommateur et le nombre de threads producteurs sont deux paramètres obtenus à partir de la ligne de commande ;
- Le buffer est un tableau partagé de 8 places, contenant des entiers (`int`) :
  - Une donnée produite ne doit jamais 'écraser' une donnée non consommée !
  - À chaque insertion, chaque thread insère dans le buffer un `int` fixe, qui est son identifiant.
- Entre chaque consommation ou production, un thread « simule » un traitement utilisant de la ressource CPU, en utilisant : `for (int i=0; i<10000; i++);`
- **Le traitement se fait en dehors de la zone critique.**
- Le nombre d'éléments produits (et donc consommé) est toujours de **131072**.

Tâche 1.3 – Coder le problème des lecteurs et écrivains (code du TD) :

- Le nombre de threads écrivains et le nombre de threads lecteurs sont deux paramètres obtenus à partir de la ligne de commande ;
- Un écrivain ou un lecteur « simule » un accès en écriture ou en lecture à la base de données avec l'opération : *`for (int i=0; i<10000; i++);`* il n'y a pas d'attente entre deux tentatives d'accès.
- Le(s) écrivain(s) effectue(nt) **640** écritures et le(s) lecteur(s)effectue(nt) **2540** lectures **au total**.
- **Contrairement au problème des producteurs-consommateurs, le traitement se fait DANS la zone critique.**

Tâche 1.4 - Écrire un script d'évaluation des performances

- Sur le modèle du précédent TD, mesurer la performance de chacun des trois programmes et la sauver dans des fichiers .csv en faisant attention à :
  - Désactiver toute sortie sur STDOUT (car cela diminuerait les performances) ;
  - Prendre 5 mesures pour chaque configuration avec les nombres de threads **TOTAUX** suivants: [2, 4, 8, 16, 32]
    - Pour les problèmes avec deux types distincts de threads, vous devez donc séparer ce nombre en 2 ensembles de threads de taille égale.

3 graphes avec programmes de base

[2] Un code qui ne compile pas est par definition faux et obtiendra 0 pour ce critère.

Tâche 1.5 – Représenter graphiquement les résultats :

- En utilisant matplotlib et un script python, représenter graphiquement le temps d'exécution en fonction du nombre de threads, en respectant les consignes suivantes :
    - Les axes x et y doivent avoir des titres clairs ;
    - Chaque mesure doit présenter la moyenne et l'écart type[3] ;
    - L'axe des y doit systématiquement commencer à 0.

À la fin de cette première partie vous devez obtenir 3 plots, avec votre interprétation des résultats présentés. Vous referez le test avec la machine multicœurs pour la version qui sera ajoutée dans votre rapport.

Partie 2 : Mise en œuvre des primitives de synchronisation par attente active

Dans cette deuxième partie, vous implémenterez la synchronisation par attente active en utilisant des instructions atomiques. Vous analyserez leur performance et la comparerez avec celle des primitives de synchronisation POSIX pour les programmes réalisés en première partie.

Tâche 2.1 – Mettre en œuvre un verrou par attente active utilisant l'opération atomique `xchg`, sur le modèle de l'algorithme *test-and-set* vu en cours :

- Votre verrou présentera une interface `lock()` et `unlock()` permettant de protéger l'accès par les threads à leurs sections critiques ;
- La mise en œuvre doit utiliser de l'assembleur « inline » dans votre programme C. Vous étudierez la documentation suivante pour savoir comment faire, en complément des exemples présentés dans le syllabus : http://cristal.univ-lille.fr/~marquet/ens/ctx/doc/l-ia.html

Tâche 2.2 – Mesurez la performance du verrou test-and-set :

1 Graphe TAS de base

- Écrire un programme de test qui lance un nombre de threads N fourni en ligne de commande. Chaque thread effectue **32768/N** sections critiques.
- Chaque section critique a une durée qui est émulée avec une boucle « for » comme dans la Tâche 1.2. Il n'y a pas d'attente entre deux tentatives d'accéder à une section critique[3].
- Vous mesurez et représentez graphiquement le temps total pour les nombre de threads suivants: [1, 2, 4, 8, 16, 32], en respectant les mêmes consignes que pour la tâche 1.4.

modif le graphe de 2.2 pour ajouter une seconde courbe

Tâche 2.3 – Implémentez l'algorithme test-and-test-and-set.

- Ajoutez les résultats avec cet algorithme à la courbe de la tâche précédente.

Tâche 2.4 – Concevez une interface sémaphore sur la base de vos primitives d'attente active.

- Bien entendu, votre interface ne peut pas utiliser les appels `sem_wait()` et `sem_post()`

Tâche 2.5 – Adaptez vos 3 programmes de la partie 1 pour utiliser vos primitives d'attente active.

- Mesurez la performance et intégrez là aux 3 plots produits en partie 1[4] avec les mêmes contraintes qu'énoncé en Tâche 1.4.

modif les 3 plots de 1.4 (ajouter les courbes) en ajoutant TAS sem et TATAS sem

Vous obtiendrez à l'issue de cette partie 1 nouveau plot (Tâche 2.2) et aurez mis à jour les trois plots de la partie 1. Votre rapport discutera les résultats obtenus et expliquera les performances observées.

Si vous le souhaitez, vous pouvez réaliser la tâche optionnelle suivante (+10% de bonus maximum, mais votre note ne peut pas dépasser 100%).

Tâche optionnelle 2.6 – Implémentez le verrou *backoff-test-and-test-and-set* en utilisant une boucle d'attente similaire à celle décrite dans la Tâche 1.2, et testez sa performance avec différentes valeurs pour la durée d'attente minimale (vous réaliserez l'attente sur le modèle de la tâche 1.2, en adaptant la borne de la boucle `for`). Cette tâche nécessite de réaliser quelques tests pour déterminer les bonnes valeurs pour les intervalles de temps minimal et maximal sur votre machine… Intégrez les résultats au plot produit dans la tâche 2.2.

---

[3] Comme les sections critiques s'exécutent en exclusion mutuelle et que leur nombre est toujours le même, le temps total d'exécution avec un algorithme d'exclusion mutuelle « parfait » devrait être constant quel que soit le nombre de threads. Ce n'est bien sûr pas le cas ici, et ce test mesure donc le surcoût de cet algorithme d'exclusion mutuelle.

[4] Si la performance est trop différente, vous pouvez aussi produire deux plots séparés.