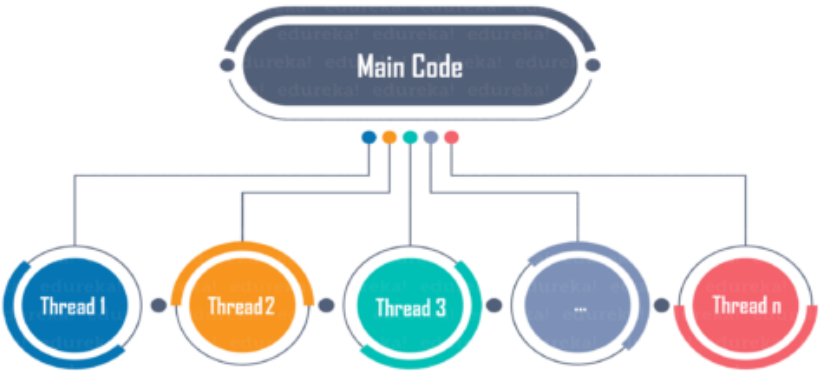
École Polytechnique de Louvain

UCLouvain

Système informatique

LINFO1252



Membres du groupe 10.3:
Charles Lohest 17201800,
Guang Li 54211800

07-11-2020

1 Introduction

Dans ce rapport nous analyserons les performances de plusieurs problèmes utilisant des threads.Philosophes, producteurs consommateurs,lecteur écrivain et enfin spinlock.Ensuite nous implémenterons notre propre interface sémaphore que nous remplacerons sur nos programmes et comparerons les performances. Les mesures sont effectués sur une machine virtuelle comprennant 4 coeurs.Nous effectuons donc 5 essais sur des threads allant de 1 à 8.

2 Partie 1

2.1 Philosophes

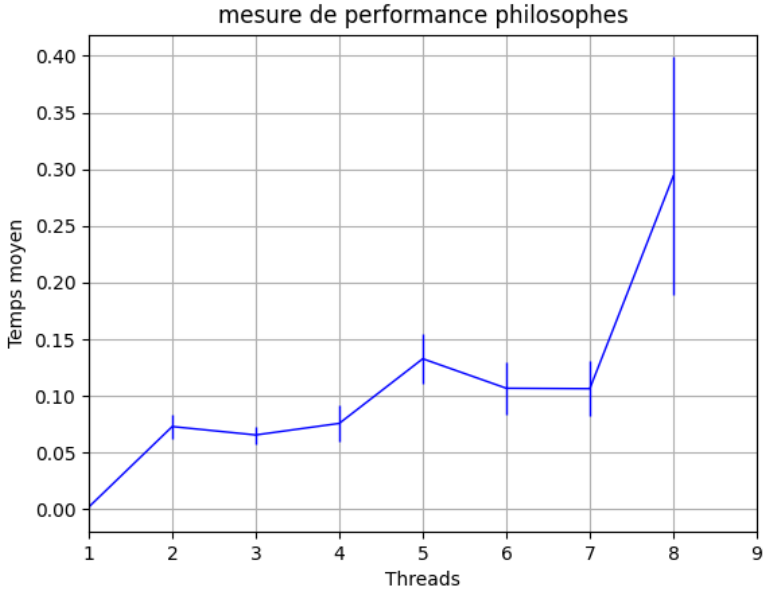
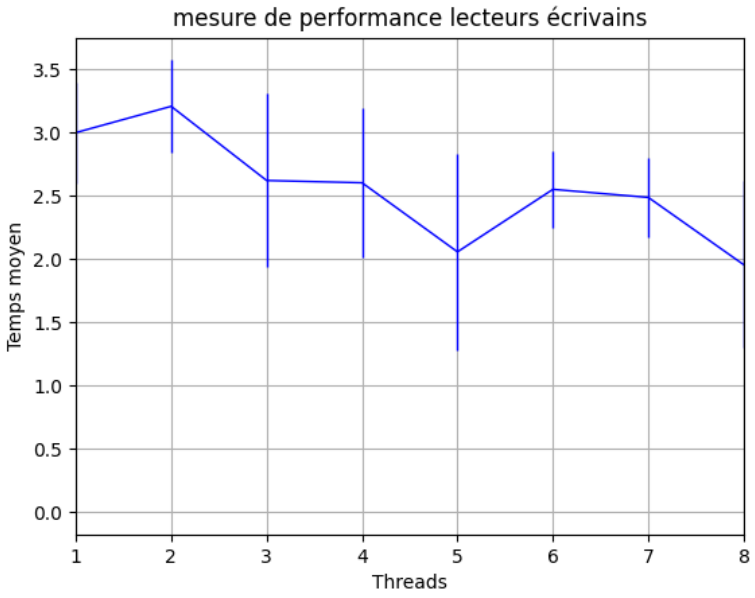


Figure 1: Philosophe with semaphore.h

Dans ce programme chaque threads effectuent le même nombre de cycle d'actions penser/manger il est donc logique que le temps de performance augmente lorsqu'on augmente le nombre de threads.Les threads se bloquent lorsqu'ils n'ont pas accès aux baguettes et se libère lorsqu'ils ont fini de manger.Le graphique doit être bien montant avec une pente inférieure a 1 car plusieurs philosophes peuvent manger en même temps et au plus les philosophes augmentent au plus la pente décroît jusqu'au nombre limite de threads ou le comportement des threads est plus aléatoire.

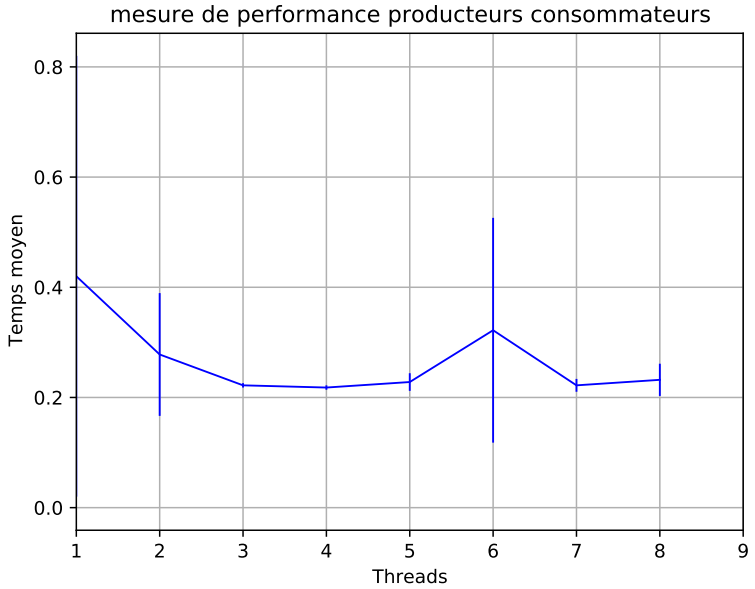
2.2 Lecteurs-Écrivains

Dans ce code du lecteur écrivain on voit clairement que le temps diminue lorsqu'on ajoute des threads. En effet plus de lecteurs écrivains peuvent effectuer la tache demandée et ainsi prendre moins de temps. On peut remarquer qu'au delà du nombre de threads disponibles c'est à dire 5 les performances ne diminuent plus mais fluctuent.Ce qui est du au fait que le nombre maximal de threads utilisable est pris.La diminution de temps lorsqu'on augmente le nombre de threads s'explique du fait que le programme s'attend a un nombre fixe d'exécution donc au plus il y a de threads au plus ce nombre diminuera rapidement contrairement au problème des philosophes ou chaque threads effectuent le même nombre d'actions



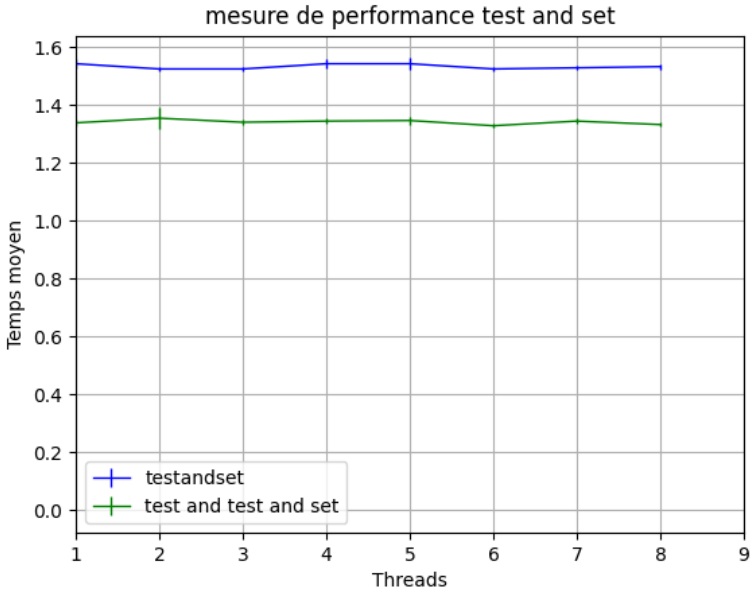
2.3 Producteurs consommateurs

Le problème des producteurs consommateurs implémenté ici mesure les performances de traitement de 1024 produits en fonction du nombre de threads. On peut observer que les temps sont proches les uns des autres peut importe le nombre de threads car il n'y que peu de calculs effectués hors de la section critique.

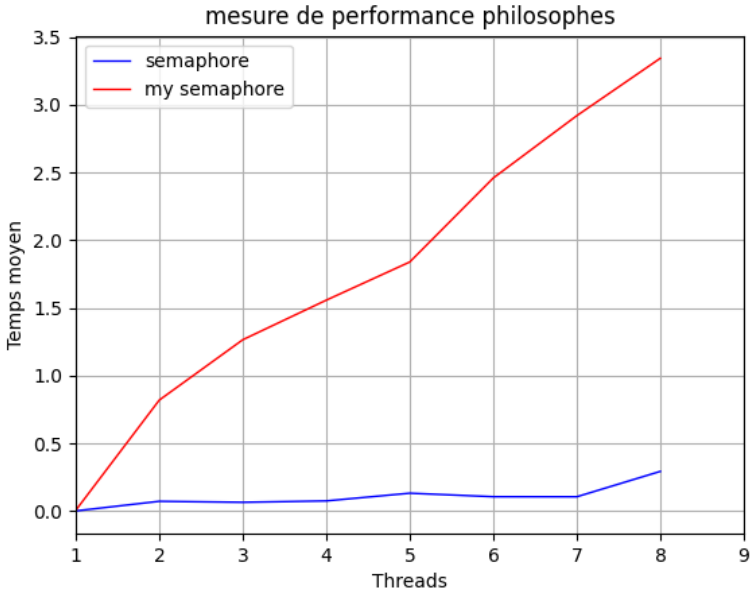


2.4 Spinlocks

On peut observer la différence entre le test and set et le test and test and set.On remarque que les deux spinlock ont la même allure ce qui est normale étant donné qu'ils reposent sur la même idée d'implémentation.Le test and test and set est logiquement plus rapide que le test and set car il s'agit d'une amélioration de celui ci ayant pour but d'effectuer moins d'opération et donc être plus performant



2.5 MyPhilo

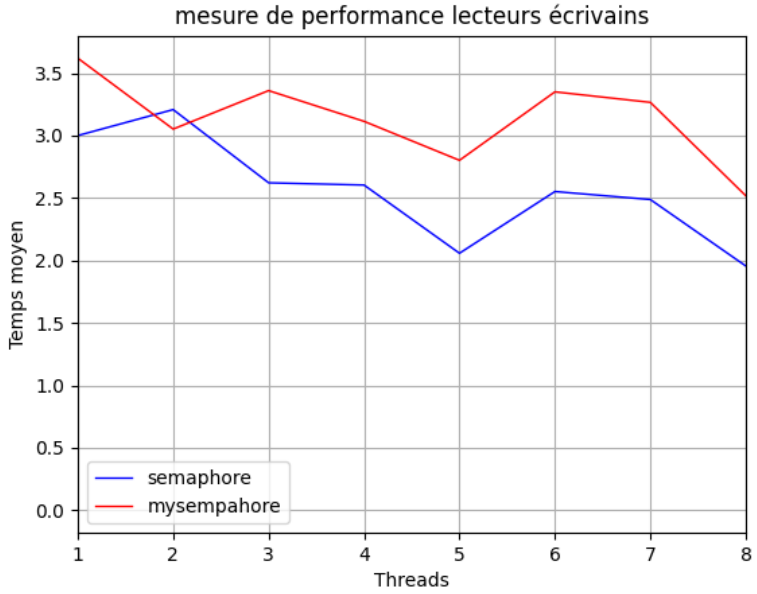


Dans cette mis à jour du premier plot on compare le philosophe utlisé au début avec les sémaphore de la librairie et le code du philosophe implémenté avec notre sémaphore.On observe directement que le code utilisant notre sémaphore est bien plus lent cependant il présente une allure similaire(bien sur amplifiée) que le code utilisant les vrais sémaphores.Cette lenteur s'explique par le fait que notre interface sémaphore ne soit pas des plus optimmisé car elle utilise un simple test and test and set comme spinlock qui fonctionne mais n'est pas le spinlock le plus rapide.Et les performances sont encore rallongées car on implémente notre interface de sémaphore (non binaire) avec une autre interface de sémaphore binaire.Ainsi l'optimisation des performances n'est pas la meilleure

2.6 My lecteur-ecrivain

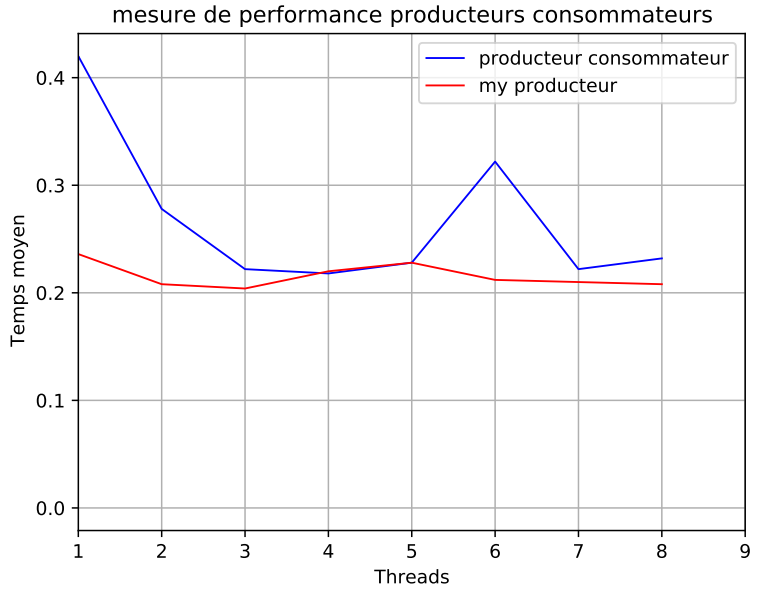
Nous observons la même chose que la section d'au dessus avec une courbe qui représente notre sémaphore,plus lente que l'implémentation initiale.De nouveau l'allure de la courbe est similaire ce qui signifie que le fonc-

tionnement est bien le même mais qu'il prend juste plus de temps.La différence de temps est moins observable que sur le philosophe car le nombre de sémaphore reste fixe et le programme prend déjà initialement plus de temps.De plus le programme effectue un nombre d'action total qui ne changera pas en fonction du nombre de threads attribués



2.7 My producteurs consommateurs

On observe que l'implementation des semaphores utilisant des spinlocks est plus performante que les sémaphores posix. Cela peut être du au temps d'exécution plus court qui ne permet pas de voir de différence de performance dans la file de priorité mais à l'initalisation plus rapide des semaphores implementées dans "semsem.h".



3 Conclusion

En conclusion après avoir codé différents problèmes utilisant les threads et sémaphores, avoir implémenté des méthodes d'attente active avec verrous en inline assembly et avoir implémenté notre propre interface

sémaphore pour adapter nos codes à celle ci. Nous avons constaté grâce aux différentes mesures de performances, que les sémaphores permettent de réguler les threads en leurs demandant d'attendre un signal. Ceci a pour effet de réguler les entrées de threads dans le système et ainsi d'éviter les précipitations(tous les threads qui rentrent en même temps dans une boucle) et les deadlocks qui mènent à des codes qui ne fonctionnent pas. De plus nous avons appris comment fonctionne une sémaphore grâce aux verrous (Un thread qui reste bloqué jusqu'a un changement de valeur).L'interface sémaphore de POSIX se base sur cette idée d'attente de signal. Cependant elle reste en général plus performante que la notre sûrement car elle se base sur un concept plus performant qu'un simple test and test and set. De plus nous avons décidé d'implémenter les sémaphores non binaires à l'aide de sémaphores binaires ce qui n'améliore pas les performances.Nous comprenons donc maintenant l'importance de sémaphores performantes dans les programmes "threadés" qui nous éviterons tous problèmes de deadlock tout en permettant des bonnes performances et l'utilisation du plein potentiel des processeurs multi-coeurs qui permettent d'effectuer plusieurs taches simultanément.