PROJET CS-PC

HADJ-HAMDRI Mohammed-Amine

EL-TAHIRI Youness

3 ETI GRP-D

2023

Exercices réalisés :

- Course-hippique (Arbitre)
- Faites des calculs : -V1 (1 demandeur, n calculateurs)

-V2.1 (m demandeurs, n calculateurs)

-V2.2 (lambda functions)

- Gestionnaires de billes
- Estimation de pi (Monte-Carlo)
- Estimation de pi (espérance)
- Multi-tâche restaurant
- Multi-tâche pression et température

Simulateur de Course Hippique en Python:

Description : Cet exercice propose une simulation de course hippique où plusieurs chevaux se déplacent simultanément sur une piste. Chaque cheval est représenté par un processus distinct, permettant ainsi une exécution parallèle réaliste de la course. L'exercice utilise la bibliothèque multiprocessing de Python pour gérer les processus et une liste partagée pour mettre à jour les positions des chevaux.
Méthode employée : Le code crée des processus pour chaque cheval en utilisant la bibliothèque multiprocessing. Chaque processus exécute une fonction qui gère le

déplacement du cheval sur la piste. Les positions des chevaux sont stockées dans une liste partagée, protégée par un sémaphore pour éviter les conflits d'accès concurrents.

Explication du code : Le code commence par définir des constantes et des codes d'échappement pour le formatage de l'affichage à l'écran. Ensuite, des fonctions utilitaires sont définies pour l'effacement de l'écran, le déplacement du curseur, etc.
Le code principal demande d'abord le pronostic de l'utilisateur, puis initialise les variables et les objets nécessaires à la course. Les processus pour chaque cheval sont créés à l'aide de la fonction mp.Process() et stockés dans une liste.
Chaque processus de cheval exécute une fonction qui met à jour la position du cheval dans la liste partagée et affiche le déplacement à l'écran. Un processus arbitre surveille la course en vérifiant périodiquement les positions des chevaux et affiche le cheval en tête et le cheval en queue.
Une fois que tous les chevaux ont terminé la course, le résultat est affiché, indiquant si le pronostic de l'utilisateur était correct ou non.
Résultats obtenus : L'exécution du code affiche en temps réel l'avancement des chevaux sur la piste. À la fin de la course, le résultat est affiché, permettant à l'utilisateur de savoir s'il a fait le bon pronostic.

Estimation de pi (Monte-Carlo) :

Le code Python fourni utilise la méthode de Monte-Carlo pour estimer la valeur de π. Cette méthode repose sur la génération aléatoire de points dans un carré unitaire et la détermination du nombre de points situés à l'intérieur d'un cercle unitaire inscrit dans le carré. En augmentant le nombre de points générés, on peut obtenir une meilleure approximation de π.
Le code est divisé en deux parties principales. La première partie, la méthode mono-processus, effectue un nombre donné d'essais pour estimer π. Elle utilise une boucle pour générer aléatoirement les coordonnées x et y de chaque point, puis vérifie si le point est situé à l'intérieur du cercle unitaire en comparant la somme de leurs carrés à 1. Le nombre de points situés à l'intérieur du cercle est compté et stocké dans une variable partagée appelée "hits".
La deuxième partie du code, la méthode multi-processus, vise à accélérer le calcul en répartissant le travail sur plusieurs processus. Elle crée un nombre de processus égal au nombre de cœurs de calcul disponibles sur la machine. Chaque processus exécute la même fonction que la méthode mono-processus, mais avec un nombre réduit d'essais. Les résultats partiels, c'est-à-dire le nombre de hits, sont ajoutés à la variable partagée "hits". Une fois que tous les processus ont terminé leur travail, la valeur estimée de π est calculée en utilisant le nombre total de hits.
En exécutant ce code, on obtient une estimation précise de π en utilisant la méthode de Monte-Carlo avec une grande efficacité grâce à la programmation multi-processus. La méthode multi-processus permet de diviser le travail entre plusieurs cœurs de calcul, ce qui accélère le calcul global et réduit le temps d'exécution. Les résultats, comprenant la valeur estimée de π et le temps de calcul, sont affichés à la fin de l'exécution pour comparer les performances des deux méthodes.
En résumé, ce code illustre l'utilisation de la méthode de Monte-Carlo pour estimer π, en utilisant à la fois la programmation mono-processus et multi-processus pour améliorer

l'efficacité du calcul. La programmation multi-processus permet d'exploiter les ressources du processeur de manière plus efficace, ce qui réduit considérablement le temps de calcul nécessaire pour obtenir une estimation précise de π.

Gestion de billes avec sémaphores en Python:

Cet exercice propose une simulation de gestion de billes en utilisant des sémaphores grâce au module multiprocessing de Python. L'objectif est de permettre à plusieurs travailleurs d'accéder simultanément à des billes en les contrôlant. Chaque travailleur fait une demande de billes, effectue une tâche pendant un certain laps de temps, puis rend les billes pour qu'elles puissent être utilisées par d'autres travailleurs.
Le code utilise le module multiprocessing pour créer des processus parallèles qui représentent les travailleurs. Un sémaphore est utilisé pour gérer l'accès à la ressource (les billes) et éviter les conflits d'accès concurrents. Chaque travailleur exécute une fonction qui demande un certain nombre de billes, effectue une tâche simulée, puis rend les billes pour libérer la ressource.
Le code commence par définir les constantes et les variables nécessaires, telles que le nombre maximal de billes disponibles, le nombre de billes requis par chaque travailleur et les objets de sémaphore.
Ensuite, des fonctions sont définies pour demander et rendre les billes. La fonction de demande acquiert le sémaphore, vérifie si le nombre de billes disponibles est suffisant, réduit le nombre de billes disponibles, puis libère le sémaphore. Si le nombre de billes disponibles est insuffisant, le travailleur est mis en attente jusqu'à ce que d'autres travailleurs rendent des billes.
Chaque travailleur exécute une boucle où il affiche le nombre de billes restantes, fait une demande de billes en utilisant la fonction de demande, effectue une tâche simulée, puis rend les billes en utilisant la fonction de rendu. Le contrôleur surveille périodiquement le nombre de billes disponibles et affiche un message d'avertissement en cas d'incohérence.
Dans la partie principale du code, les processus travailleurs sont créés et lancés, ainsi que le processus contrôleur. Une fois que tous les travailleurs ont terminé leur tâche, le processus contrôleur se termine et le programme se termine également.
Comme résumé : l'exécution du code simule la gestion simultanée des ressources. Les travailleurs font des demandes de billes, effectuent des tâches simulées, puis rendent les billes. Le contrôleur surveille le nombre de billes disponibles pour assurer leur cohérence. Le code se termine lorsque tous les travailleurs ont fini leur tâche.

Estimation concurrente de l'aire d'un quart de cercle:

Description de l'exercice : Cet exercice vise à estimer l'aire d'un quart de cercle en utilisant la méthode d'espérance avec la programmation concurrente. On tire N valeurs aléatoires de l'abscisse X d'un point M dans l'intervalle [0, 1]. En utilisant ces valeurs, on calcule la somme S des N valeurs prises par f(X) = √(1 - X^2). La moyenne de ces N valeurs de f(X) fournit une approximation de l'aire du quart de cercle (π/4). Pour optimiser le temps de calcul, on utilise la programmation concurrente en répartissant le travail entre plusieurs processus.
Méthode employée : Le code utilise la programmation concurrente en créant plusieurs processus pour effectuer le calcul de la somme S de manière simultanée. Chaque processus

génère une partie des valeurs aléatoires de l'abscisse X et calcule la somme partielle correspondante. Une fois que tous les processus ont terminé leur travail, les sommes partielles sont combinées pour obtenir la somme totale S.
Explication du code : Le code commence par définir le nombre de processus à utiliser et le nombre de pas N pour le tirage aléatoire des valeurs de l'abscisse X. Ensuite, il crée une file d'attente (queue) pour stocker les sommes partielles calculées par chaque processus.
Ensuite, une boucle est utilisée pour créer les processus et les démarrer. Chaque processus génère une partie des valeurs aléatoires de X et calcule la somme partielle correspondante. Les sommes partielles sont ajoutées à la file d'attente.
Une fois que tous les processus ont terminé leur travail, la somme totale S est calculée en récupérant les sommes partielles de la file d'attente et en les combinant.
Le résultat final, correspondant à l'estimation concurrente de l'aire du quart de cercle, est affiché à l'écran.
Résultats obtenus : L'exécution du code estime l'aire d'un quart de cercle en utilisant la méthode d'espérance avec la programmation concurrente. Le résultat affiché correspond à cette estimation, obtenue en utilisant un nombre spécifié de processus et de pas N. La programmation concurrente permet d'accélérer le calcul en répartissant le travail entre plusieurs processus. Le code se termine après avoir affiché le résultat final.

Gestion concurrente des processus de commande dans un restaurant:
Description du code : Ce code simule la gestion concurrente des processus de commande dans un restaurant. Le processus client génère un certain nombre de commandes aléatoires, puis les envoie à la file d'attente des commandes à traiter. Les processus serveur, qui sont plusieurs, récupèrent les commandes en attente, les préparent et les envoient à la file d'attente des commandes préparées. En parallèle, le processus majord'homme affiche les informations sur les commandes en cours de traitement et les commandes en attente.
Le code utilise des queues pour la communication entre les processus. Il crée également un événement pour signaler la fin des commandes envoyées par le client. Les processus client, serveur et majord'homme sont créés et lancés. Après le traitement de toutes les commandes, un message de fin est affiché.
Le nombre de processus serveur et le nombre de commandes peuvent être modifiés en modifiant les variables nb_process_serveur et nb_commandes.
Résultat obtenu : L'exécution du code simule la gestion concurrente des processus de commande dans un restaurant. Les commandes sont générées, préparées et affichées en temps réel. Les commandes en attente sont également affichées. Une fois toutes les commandes traitées, un message de fin est affiché.

Multi-tâche pression et température :
Le code crée deux processus distincts, l'un pour la gestion de la pression et l'autre pour la gestion de la température. Chaque processus exécute une fonction qui met à jour la valeur respective en fonction des mesures et des actions requises. Les processus communiquent entre eux à l'aide de variables partagées pour coordonner leurs actions.
Explication du code : Le code commence par initialiser les variables nécessaires pour la pression et la température. Ensuite, les processus pour la gestion de la pression et de la température sont créés à l'aide de la bibliothèque multiprocessing.

Le processus de gestion de la pression exécute une fonction qui surveille les mesures de pression et effectue les ajustements nécessaires en fonction des seuils prédéfinis. Il met à jour la valeur de la pression dans la variable partagée.
Le processus de gestion de la température exécute une fonction similaire, surveillant les mesures de température et ajustant la valeur en fonction des seuils spécifiés. Il met également à jour la valeur de la température dans la variable partagée.
Les deux processus s'exécutent de manière concurrente et communiquent entre eux via les variables partagées. Par exemple, si la pression atteint un seuil critique, le processus de gestion de la pression peut envoyer un signal au processus de gestion de la température pour qu'il effectue des ajustements en conséquence.
Résultats obtenus : L'exécution du code permet de gérer simultanément la pression et la température dans un système. Les processus se coordonnent pour effectuer les ajustements nécessaires en fonction des mesures et des seuils prédéfinis. Les résultats, tels que les valeurs mises à jour de la pression et de la température, peuvent être affichés à des fins de surveillance ou de contrôle.

Faites des calculs (V1) :

Description (V1): Ce code implémente une simulation de traitement de demandes mathématiques par plusieurs opérateurs. Chaque demande est générée par un processus distinct, et les résultats des calculs sont stockés dans une file d'attente partagée. Le code utilise la bibliothèque multiprocessing de Python pour gérer les processus.

Méthode employée : Le code crée un processus "demandeur" qui génère aléatoirement un certain nombre de demandes mathématiques. Chaque demande est une expression composée de deux nombres et un opérateur (+, -, *, /). Les demandes sont ajoutées à une file d'attente partagée.

Ensuite, plusieurs processus "calculateur" sont créés, correspondant au nombre d'opérateurs spécifié. Chaque processus "calculateur" récupère une demande de la file d'attente, calcule le résultat de l'expression et le stocke dans une autre file d'attente partagée.

Une fois que tous les processus "calculateur" ont terminé, le processus principal récupère les résultats de la file d'attente des résultats et les affiche.

Explication du code : Le code commence par définir une fonction "demandeur" qui génère aléatoirement des demandes mathématiques et les ajoute à la file d'attente. Ensuite, une fonction "calculateur" est définie pour chaque processus "calculateur". Cette fonction récupère une demande de la file d'attente, calcule le résultat et l'ajoute à la file d'attente des résultats.

Dans le code principal, une queue est créée pour la file d'attente des demandes et une autre queue pour la file d'attente des résultats. Le processus "demandeur" est créé et démarré pour générer les demandes et les ajouter à la file d'attente. Ensuite, plusieurs processus "calculateur" sont créés, démarrés et attendus.

Une fois que tous les processus "calculateur" ont terminé, le processus principal récupère les résultats de la file d'attente des résultats et les affiche.

Résultats obtenus : L'exécution du code génère des demandes mathématiques aléatoires, les traite simultanément par plusieurs opérateurs et affiche les résultats des calculs. Les demandes et les résultats sont correctement gérés grâce à l'utilisation de la bibliothèque multiprocessing.

Faites des calculs (V2/2.2) :

Description (V2): Ce code simule un système de demandeurs et d'opérateurs où les demandeurs envoient des demandes mathématiques aux opérateurs, qui calculent les résultats et les renvoient aux demandeurs. Chaque demandeur et opérateur est représenté par un processus distinct pour une exécution parallèle réaliste. Le code utilise la bibliothèque multiprocessing de Python pour gérer les processus et des sémaphores pour éviter les conflits d'accès aux files d'attente.

Méthode employée : Le code crée des processus pour chaque demandeur et opérateur en utilisant la bibliothèque multiprocessing. Chaque demandeur génère aléatoirement une demande mathématique et l'envoie à l'opérateur correspondant via une file d'attente partagée. Les opérateurs reçoivent les demandes, calculent les résultats et les renvoient aux demandeurs via la même file d'attente partagée.

Le code utilise également des sémaphores pour protéger l'accès concurrent à la file d'attente. Un sémaphore est acquis avant d'accéder à la file d'attente et est libéré après pour éviter les conflits d'accès.

Explication du code : Le code commence par définir une fonction "demandeur" qui génère aléatoirement une demande mathématique et l'envoie à l'opérateur correspondant via la file d'attente. Cette fonction utilise un sémaphore pour éviter les conflits d'accès à la file d'attente. Une fois la demande envoyée, le demandeur attend un court laps de temps pour simuler le temps de calcul de l'opérateur. Ensuite, le demandeur reçoit le résultat de l'opérateur via la file d'attente.

Une autre fonction "calculateur" est définie pour chaque opérateur. Cette fonction reçoit une demande de la file d'attente, effectue le calcul et renvoie le résultat à travers la même file d'attente. Un sémaphore est utilisé pour éviter les conflits d'accès à la file d'attente.

Dans le code principal, une liste de files d'attente est créée pour chaque demande-opérateur pair. Ensuite, des processus sont créés pour chaque demandeur et opérateur. Chaque processus est associé à une file d'attente et à un sémaphore correspondant. Les processus sont démarrés et attendus pour terminer leur exécution.

Résultats obtenus : L'exécution du code simule un système de traitement de demandes mathématiques avec des demandeurs et des opérateurs. Les demandes sont correctement transmises aux opérateurs, les calculs sont effectués et les résultats sont renvoyés aux demandeurs. Les sémaphores garantissent que l'accès à la file d'attente est protégé contre les conflits d'accès concurrents.