PHILOSOPHERS

IDÉE GÉNÉRALE:

Commençons par expliquer l'idée générale. Tout d'abord, nous devons imaginer une table ronde, autour de laquelle sont assis X philosophes, chacun d'eux apportant une fourchette et la plaçant devant lui. À ce stade, nous savons qu'un philosophe peut faire trois choses : manger, dormir ou penser, mais pour manger, il doit choisir deux fourchettes (celle devant lui et une autre à sa droite ou à sa gauche, dans ma solution, il choisit celle de sa droite, les deux fonctionnent — mise en œuvre différente).

Prenant en considération que le maximum de philosophes qu'on peut donner en programme est 200, j'ai aussi défini ça dans un macro.
#define MAX PHILOSOPHERS 200: une macro est une constante ou une expression définie par le préprocesseur du compilateur.
Le préprocesseur du compilateur remplace chaque occurrence de `MAX PHILOSOPHERS` par la valeur `200` dans le code avant la compilation. Cela se fait avant même que le code soit compilé.

DÉBUT D'IMPLÉMENTATION

D'abord j'ai traité les arguments après et j'ai creer les threads,
Chaque thread philo exécutera la fonction de routine philo et l'observateur exécutera la fonction de surveillance .

CRÉATION DE THREAD

Dans la création de thread j'ai utilisé la fonction pthread _creat de libpthread.h, chaque thread a sa propre pile (variable locale)mais partage la mémoire globale de processus (variable globale,mémoire dynamique).
après on utilise la fonction pthread_detach: qui est appelé soit à partir de thread lui même , ou d'un autre thread et indique qu 'on voulait pas la valeur de retour ou la possibilité d'attendre qu' il se termine
il ya aussi pthread_join : qui est appelé d'après un autre thread (généralement le thread qui a creer)pour attendre qu 'un thread se termine et obtienne sa valeure de retour.

Si on créer simplement des threads avec `pthread_create` et que tu n'utilises ni `pthread_detach` ni `pthread_join`, les threads créés continueront à s'exécuter, mais leurs ressources (comme la mémoire utilisée pour leur pile)

ne seront pas automatiquement libérées après leur fin. Cela peut entraîner des fuites de mémoire si le programme crée de nombreux threads.

Dans mon programme il y a une simulation où chaque philosophe agit indépendamment, je ne veux pas bloquer le programme en attendant la fin de chaque thread avec `pthread_join`. Je veux simplement que chaque thread fonctionne de manière autonome.

Sans `pthread_detach` : Les ressources du thread ne seront pas libérées automatiquement à sa terminaison, ce qui peut poser des problèmes de gestion de mémoire dans des programmes qui créent beaucoup de threads.

CONDITION:

les philosophes peuvent soit manger , dormir ou pensait, ils peuvent pas faire ces actions en même temps.

Thread :

Un thread est une unité d'exécution qui permet à un programme d'effectuer plusieurs tâches en parallèle.

Data Race :

Une data race survient lorsque plusieurs threads accèdent à une même variable en même temps, sans protection, ce qui peut causer des erreurs imprévisibles.

Mutex :

Un mutex est un mécanisme de synchronisation qui permet de contrôler l'accès à une ressource partagée en ne permettant qu'à un seul thread de l'utiliser à la fois.

TIME IN MILLISECOND:

Dans le sujet ils ont demandé le temps en milliseconde :

j'ai utilisé la fonction time in milliseconde de librairie `<sys/time.h>.`

La fonction utilise `gettimeofday(&tv, NULL)` pour obtenir le temps actuel dans une structure `timeval`, qui contient deux champs :

- `tv_sec` : le nombre de secondes écoulées depuis le 1er janvier 1970 (l'**Epoch**).
- `tv_usec` : le nombre de microsecondes écoulées au sein de la seconde en cours.

ROUTINE :

Chaque philosophe va suivre une routine qui se répète tant que la simulation est en cours. Cette routine est constituée de trois actions principales : penser, manger, et dormir. Ces actions sont exécutées dans cet ordre pour chaque philosophe.

1. Penser :

- Le philosophe commence par penser. Penser est une activité qui ne nécessite pas de fourches, et il peut donc le faire librement.
- Dès que le philosophe commence à penser, un message est imprimé pour indiquer que le philosophe pense : « X pense » (où X est le numéro du philosophe).

2. Prendre les fourches et manger :

- Après avoir pensé, le philosophe a besoin de manger. Pour cela, il doit d'abord prendre les deux fourches disponibles à sa gauche et à sa droite.
- **Prendre la fourche droite :**
  - Le philosophe commence par prendre la fourche à sa droite. Cela se fait en verrouillant un mutex associé à cette fourche pour s'assurer qu'aucun autre philosophe ne peut l'utiliser en même temps.
  - Une fois la fourche droite prise, un message est imprimé pour indiquer que le philosophe a pris la fourche droite : « X a pris la fourche droite ».
- **Prendre la fourche gauche :**
  - Ensuite, le philosophe prend la fourche à sa gauche de la même manière, en verrouillant le mutex associé.
  - Une fois la fourche gauche prise, un message est imprimé pour indiquer que le philosophe a pris la fourche gauche : « X a pris la fourche gauche ».
- **Manger :**
  - Maintenant que le philosophe a les deux fourches, il peut commencer à manger.

    - Pendant qu'il mange, le temps du repas est simulé en utilisant une fonction appelée `ft_usleep`, qui met le programme en pause pendant une durée déterminée (spécifiée par l'utilisateur).
    - Avant de relâcher les fourches, le programme met à jour une variable indiquant le moment où le philosophe a mangé pour la dernière fois. Cette information est essentielle pour un moniteur qui surveille si un philosophe est mort de faim.
    - Pendant que le philosophe mange, un message est imprimé : « X mange »

MONITEURE

Vérifier si un philosophe est mort. Ensuite, le mutex est reverrouillé pour la prochaine vérification.

`usleep(100)` : Entre chaque vérification, la fonction fait une petite pause avec `usleep(100)` (100 microsecondes). Cela permet de "donner du temps" aux autres threads pour effectuer leurs actions (comme manger, dormir, ou vérifier la mort). Sans cette pause, la boucle pourrait être trop rapide, consommant inutilement du temps processeur, et risquerait d'empêcher les autres threads de progresser correctement.

- **Relâcher les fourches :**
    - Une fois le repas terminé, le philosophe doit déposer les deux fourches.
    - Il commence par déverrouiller le mutex de la fourche gauche, puis celui de la fourche droite, libérant ainsi les fourches pour qu'elles puissent être utilisées par d'autres philosophes.

3. Dormir :

- Après avoir mangé, le philosophe va dormir pour une période déterminée.
- Le temps de sommeil est simulé avec `ft_usleep`, comme pour le repas.
- Un message est imprimé pour indiquer que le philosophe dort : « X dort ».

4. Répétition du cycle :

- Après avoir dormi, le philosophe reprend son cycle en recommençant à penser, puis en mangeant, et ainsi de suite.
- Ce cycle se répète jusqu'à ce qu'une condition d'arrêt soit remplie (par exemple, si un philosophe meurt ou si un certain nombre de repas a été atteint).

Synthèse :

- Le cycle des actions pour chaque philosophe est de penser, puis de manger (en prenant les fourches une à une), puis de dormir.
- Chaque action est accompagnée de l'impression d'un message pour suivre l'état du philosophe.
- L'utilisation de mutex permet de gérer l'accès concurrent aux fourches, garantissant qu'un philosophe ne puisse pas prendre une fourche déjà utilisée par un autre.

Détruire les mutexes:

- La dernière étape consiste à détruire tous les mutex que vous avez initialisés, sinon ils ne fonctionnent pas. Dans cette étape, nous allons libérer toutes les données que nous avons allouées si nous avons choisi de les allouer (ce que nous n'avons pas fait).aussi on oublie pas de free tout le sources .

`pthread_mutex_destroy` **est essentiel pour s'assurer que les mutex sont proprement détruits et que les ressources associées sont libérées** avant de libérer la mémoire qui les contient avec `free`. C'est une bonne pratique pour éviter les fuites de mémoire et assurer la stabilité du programme.